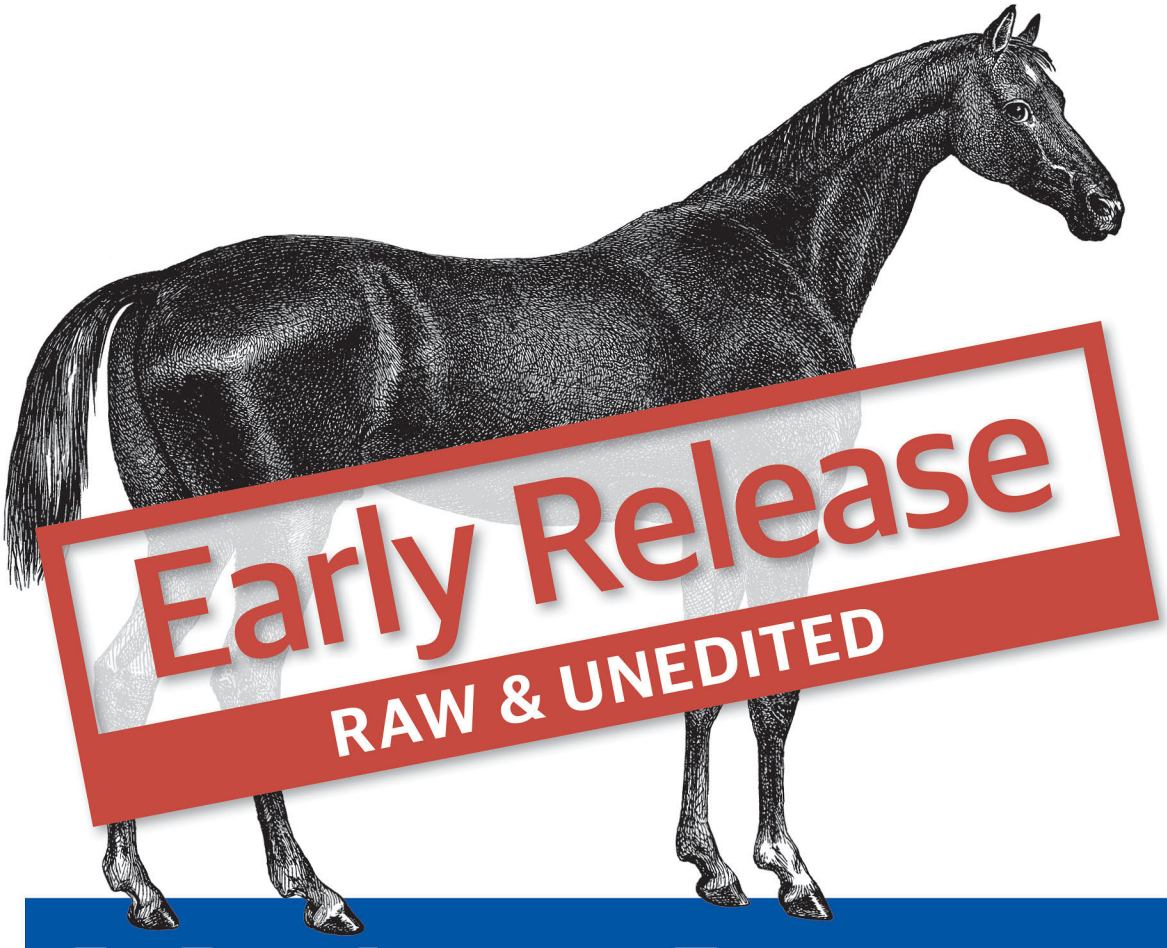
O'REILLY®



Modern Linux Administration

HOW TO BECOME A CUTTING-EDGE LINUX ADMINISTRATOR

Sam R. Alapati

Modern Linux Administration
by Sam R. Alapati

Printed in the United States of America.

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

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Brian Anderson
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition
2016-10-13: First Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491935910* for release details.

978-1-491-93591-0

[FILL IN]

Table of Contents

Preface

Who Should Read This Book

The quintessential reader for this book is someone who currently works as a Linux systems administrator, or wants to become one, having already acquired basic Linux admin skills. However, the books will be useful for all of the following:

- Developers who need to come to terms with systems concept such as scaling, as well as the fundamentals of important concepts that belong to the operations world – networking, cloud architectures, site reliability engineering, web performance, and so on.
- Enterprise architects who are either currently handling , or are in the process of creating new projects dealing with scaling of web services, Docker containerization, virtualization, big data, cloud architectures.
- Site reliability engineers (SREs), backend engineers and distributed applcition developers who are tasked with optimizing their applications as well as scaling their sites, in addition to managing and troubleshooting the new technologies increasingly found in modern systems operations.

In terms of Linux administration knowledge and background, I don't teach the basics of Linux system administration in this book. I expect the readers to know how to administer a basic Linux server and be able to perform tasks such as creating storage, managing users and permissions, understand basic Linux networking, managing files and directories, managing processes, troubleshooting server issues, taking backups, and restoring servers.

The overarching goal of this book is to introduce the reader to the myriad tools and technologies that a Linux administrator ought to know today to earn his or her keep. I do provide occasional examples, but this book is by no means a "how-to" reference for any of these technologies and software. As you can imagine, each of the technologies I discuss over the 16 chapters in this book requires one or more books dedicated

to that technology alone, for you to really learn that topic. There's no code or step-by-step instructions for the numerous newer Linux administration related technologies I discuss in this book, with a handful of exceptions. My goal is to show you want you need to know in order to understand, evaluate, and prepare to work with bleeding-edge Linux based technologies in both development and production environments.

Why I Wrote This Book

Let's say you want to learn all about the new containerization trend in application deployment and want to use Docker to make your applications portable. Just trying to come to grips with the wide range of technologies pertaining to Docker is going to make anybody's head spin – here's a (partial) list of technologies associated with just Docker containers:

- Docker project
- Docker Hub Registry
- Docker Images and Dockerfiles
- CoreOS and Atomic Host
- Cockpit
- Kubernates
- Swarm
- Compose
- Machine
- Mesos
- Zookeeper
- Consul
- Eureka
- Smartstack
- OpenShift

And all this just to learn how to work with Docker!

No wonder a lot of people are baffled as to how to get a good handle on the new technologies, which are sometimes referred to ass DevOps (however you may define it!), but really involves a new way of thinking and working with new cutting edge technologies. Many of these technologies were expressly designed to cope with the newer trends in application management such as the use of microservices, and newer ways of doing business such as cloud based environments, and new ways of data analysis such as the use of Big Data for example.

Over the past decade or so, there have been fundamental changes in how Linux system administrators have started approaching their work. Earlier, Linux admins were typically heavy on esoteric knowledge about the internals of the Linux server itself, such as rebuilding the kernel, for example. Other areas of expertise that marked one as a good Linux administrator were things such as proficiency in shell scripting, awk & sed, and Perl & Python.

Today, the emphasis has shifted quite a bit – you still need to know all that a Linux admin was expected to know years ago, but the focus today is more on your understanding of networking concepts such as DNS and routing, scaling of web applications and web services, web performance and monitoring, cloud based environments, big data and so on, all of which have become required skills for Linux administrators over the past decade.

In addition to all the new technologies and new architectures, Linux system administrators have to be proficient in new ways of doing business – such as using the newfangled configuration management tools, centralized version control depositories, continuous development (CI) and continuous deployment (CD), just to mention a few technologies and strategies that are part of today's Linux environments

As a practicing administrator for many years, and someone who needs to understand which of the technologies out of the zillion new things out there really matter to me, it's struck me that there's a lack of a single book that serves as a guide for me to navigate this exciting but complex new world. If you were to walk into an interview to get hired as a Linux administrator today, how do you prepare for it? What are you really expected to know? How do all these new technologies related to each other? Where do I start? I had these types of concerns for a long time, and I believe that there are many people that understand that changes are afoot and don't want to be left behind, but don't know how and where to begin.

My main goal in this book is to explain what a Linux administrator (or a developer/architect who uses Linux systems) needs to understand about currently popular technologies. My fundamental thesis is that traditional systems administration as we know it won't cut it in today's technologically complex systems dominated by web applications, big data, and cloud-based systems. To this end, I explain the key technologies and trends that are in vogue today (and should hold steady for a few years at least), and the concepts that underlie those technologies.

There's a bewildering array of modern technologies and tools out there and you're expected to really know how and where to employ these tools. Often, professionals seeking to venture out into the modern systems administration areas aren't quite sure where exactly they ought to start, and how the various tools and technologies are related. This book seeks to provide sufficient background and motivation for all the key tools and concepts that are in use today in the systems administration area, so you can go forth and acquire those skill sets.

A Word on the New Technologies that are critical for Linux Administrators Today

In order to be able to write a book such as this, with its wide-ranged and ambitious scope, I've had to make several decisions in each chapter as to which technologies I should discuss in each of the areas I chose to cover in the book. So, how did I pick the topics that I wanted to focus on? I chose to reverse engineer the topic selection process, meaning that I looked at what organizations are looking for today in a Linux administrator when they seek to hire one. And the following is what I found.

Expertise in areas such as infrastructure automation and configuration management (Chef, Puppet, Ansible, SaltStack), version control (Git and Perforce), big data (Hadoop), cloud architectures (Amazon Web Services, OpenStack), monitoring and reporting (Nagios and Ganglia), new types of web servers (Nginx), load balancing (Keepalived and HAProxy), databases (MySQL, MongoDB, Cassandra), caching (Memcached and Redis), Virtualization (kvm), containers (Docker), server deployment (Cobbler, Foreman, Vagrant), source code management (Git/Perforce), version control management (Mercurial and Subversion), Continuous integration and delivery (Jenkins and Hudson), log management (Logstash/ElasticSearch/Kibana), metrics management (Graphite, Cacti and Splunk) .

Look up any job advertisement for a Linux administrator (or Devops administrator) today and you'll find all the technologies I listed among the required skillsets. Most of the jobs listed require you to have a sound background and experience with basic Linux system administration – that's always assumed - plus they need many of the technologies I listed here.

So, the topics I cover and the technologies I introduce and explain are based on what a Linux administrator is expected to know today to work as one. My goal is to explain the purpose and the role of each technology, and provide a conceptual explanation of each technology and enough background and motivation for you to get started on the path to mastering those technologies. This book can thus serve as your "road map" for traversing this exciting (but intimidating) new world full of new concepts and new software, which together have already transformed the traditional role of a system administrator.

Navigating This Book

This book is organized roughly as follows:

- Chapter 1 explains the key trends in modern systems administration, such as virtualization, containerization, version control systems, continuous deployment and delivery, big data, and many other newer areas that you ought to be familiar with, to succeed as a system administrator or architect today. I strive to drive

home the point that in order to survive and flourish as a system administrator in today's highly sophisticated Linux based application environments, you must embrace the new ways of doing business, which includes a huge number of new technologies, as well as new ways of thinking. No longer is the Linux system administrator an island until himself (or herself)! In this exciting new world, you'll be working very closely with developers and architects – so, you must know the language the other speaks, as well as accept that the other groups such as developers will be increasingly performing tasks that were once upon a long time used to be belong to the exclusive province of the Linux administrators. Tell me, in the old days, did any developer carry a production pager? Many do so today.

- Chapter 2 provides a quick and through introduction to several key areas of networking, including the TCP/IP network protocol, DNS, DHCP, SSH and SSL, subnetting and routing, and load balancing. The chapter concludes with a review of newer networking concepts such as Software Defined Networking (SDN). Networking is much more important now than before, due to its critical role in cloud environments and containerization.
- Chapter 3 is a far ranging chapter dealing with the scaling of web applications and provides an introduction to web services, modern databases, and new types of web servers. You'll learn about web services and microservices and the differences between the two architectures. The chapter introduces you to concepts such as APIs, REST, SOAP and JSON, all of which play a critical role in modern web applications, which are a key part of any systems environment today. Service discovery and service registration are important topics in today's container heavy environments and you'll get an introduction to these topics here. The chapter also introduces you to modern web application servers such as Nginx, caching databases such as Redis and NoSQL databases (MongoDB).
- Chapter 4 discusses traditional virtualization and explains the different types of hypervisors. The chapter also introduces containers and explains the key ideas behind containerization, such as namespaces. SELinux and Cgroups (control groups), thus helping you get ready for the next chapter, which is all about Docker.
- Chapter 5 is one of the most important chapters in the book since it strives to provide you a through introduction to Docker containers. You'll learn about the role containerization plays in supporting application deployment and portability. You'll learn the basics of creating and managing Docker containers. The chapter explains the important and quite complex topic of Docker networking, both in the context of a single container as well as networking among a bunch of containers. The chapter discusses exciting technologies such as Kubernates, which helps orchestrate groups of containers, as well as how to use Flannel to set up IP address within a Kubernates cluster. I also show how to use Cockpit, a Web-based

container management tool, to manage containers running in multiple hosts in your own cloud. New slimmed down operating systems such as CoreOs and Red Hat Atomic Host are increasingly popular in containerized environments and therefore, I explain these types of "container operating systems" as well in this chapter.

- Chapter 6 shows how to automate server creation with the help of tools such as PXE servers, and automatic provisioning with Razor, Cobbler and Foreman. You'll learn how Vagrant helps you easily automate the spinning up of development environments.
- Chapter 7 explains the principles behind modern configuration management tools, and shows how popular tools such as Puppet and Chef work. In addition, you'll learn about two very popular orchestration frameworks – Ansible and Saltstack.
- Chapter 8 discusses two main topics – revision control and source code management. You'll learn about using Git and GitHub for revision control, as well as other revision control tools such as Mercurial, Subversion and Perforce.
- Chapter 9 is about two key modern application development concepts – continuous integration (CI) and continuous delivery (CD). The chapter explains how to employ tools such as Hudson, Jenkins, and Travis for CD and CI.
- Chapter 10 has two main parts. The first part is about centralized log management with the ELK (Elasticsearch, Logstash, and Kibana) stack. Performing trend analyses and gathering metrics with tools such as Graphite, Cacti, Splunk, and DataDog is the focus of the second part of the chapter.
- Chapter 11 shows how to use the popular OpenStack software to create an enterprise Infrastructure-–as-a-Service. You'll learn the architecture and concepts relating to the OpenStack cloud, and how it integrates with PaaS (Platform-as-a-Service) solutions such as Red Hat OpenShift and CloudFoundry.
- Chapter 12 is about using Nagios for monitoring and alerts and also explains the concepts and architecture that underlie Ganglia, an excellent way to gather system performance metrics. I also introduce two related tools – Sensu for monitoring and Zabbix for log management.
- Chapter 13 provides you a quick overview of Amazon Web Services (AWS) and the Google Cloud Platform, two very successful commercial cloud platforms.
- Chapter 14 consists of two main parts: the first part is about managing new types of databases such as MongoDB and Cassandra. The second part of the chapter explains the role of the Linux administrator in supporting big data environments powered by Hadoop. Hadoop is increasingly becoming popular and you need to know the concepts that underlie Hadoop, as well as the architecture of Hadoop 2,

the current version. The chapter shows how to install and configure Hadoop at a high level, as well how to use various tools to manage Hadoop storage (HDFS).

- Chapter 15 deals with security and compliance concerns in a modern systems environment. The chapter explains the unique security concerns of cloud environments, and how to secure big data such as Hadoop's data. You'll learn about topics such as identity and access management in AWS, virtual private networks, and security groups. The chapter closes by discussing Docker security, and how to make concessions to traditional security best practices in a containerized environment, and how to use super privileged containers.
- Chapter 16 is somewhat of a mixed bag! This final chapter is mostly about software reliability engineering (SRE) and it does by explaining various performance related topics such as enhancing Web Server performance, tuning databases and JVMs (Java Virtual Machines), and tuning the network. You'll learn about web site performance optimization using both RUM (real user monitoring) and through generating synthetic performance statistics.

If you're like us, you don't read books from front to back. If you're really like us, you usually don't read the Preface at all! Here are some basic guidelines as to how you may approach the book:

- Read Chapter 1 in order to understand the scope of the book and the lay of the land, so to speak. This chapter provides the motivation for the discussion of all the technologies and concepts in the remaining chapters.
- Quickly glance through Chapter 2, if you think you need a refresher course in essential networking concepts for a Linux administrator. If your networking chops are good, skip most of Chapter 2, except the very last part, which deals with modern networking concepts such as software defined networks (SDN).
- You can read the rest of the chapters in any order you like, depending on your interest and needs – there are really no linkages among the chapters of the book!
- Remember that conceptual overview of the various tools and software and explanation of the technical architectures are the real focus of the book – if you need to drill deep into the installation and configuration of the various tools, you'll need to read the documentation for that tool (or a book on that topic).

I hope you enjoy each of the chapters as much as I've enjoyed writing the chapters!

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

PROD: Please reach out to author to find out if they will be uploading code examples to oreilly.com or their own site (e.g., GitHub). If there is no code download, delete this whole section. If there is, when you email digidist with the link, let them know what you filled in for title_title (should be as close to book title as possible, i.e., learning_python_2e). This info will determine where digidist loads the files.
Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/oreillymedia/title_title*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc. 1005 Gravenstein Highway North Sebastopol, CA 95472 800-998-9938 (in the United States or Canada) 707-829-0515 (international or local) 707-829-0104 (fax)
We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/<catalog page>*.

Don't forget to update the link above.
To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Fill in...

CHAPTER 1

Modern Linux System Administration

Linux (and other) system administration has changed tremendously since the advent of internet based web applications and the proliferation of Big Data based systems, and the rush to cloud based systems. A quick perusal of job postings will reveal that organizations are looking for administrators who can handle the seismic changes in IT systems over the past decade. To be successful in this milieu, you need to understand how to work with the new computing paradigms such as cloud based systems, continuous integration and delivery, microservices, modern web application architectures, software based networks, big data, virtualization and containerization.

Old line systems administration isn't obsolete by any means, but as organizations keep moving to the public cloud, there's less need for traditional system administration skills. Today, cloud administrators will take care of the mundane system administration tasks – one need not spend as much time tweaking the Linux kernel as the old Linux ninjas used to, maybe you'll never have to muck with the kernel in most cases, since that's all done by the teams that manage the server farms in the cloud. "The times are a changing", and it's important to move along with the changing times and learn what makes the modern system administrator tick. For example, if you don't know the Ruby programming language, you're at a huge disadvantage today, since many important tools that are in vogue today (and will be tomorrow as well!) are scripted with Ruby.

As our goal is to understand the main concepts and tools involved in modern system administration, this book doesn't presume to explain the fundamentals of traditional system administration. For new system administrators or users, this book shows you what you need to know after you learn the basics of system administration. For the experienced users, this book shows what you need to know to stay relevant in today's world of system administration.

The speed of innovation and the heightened competition due to the ease of starting up new web based businesses has been the primary catalyst behind many of the changes in system administration. In order to survive and flourish in today's environment, organizations need to make changes incredibly fast. The fast pace means that new software and enhancements to existing software don't have the luxury of time as in the past.

The giants of Web based businesses today, such as Netflix, Facebook and the rest all can make very swift changes to their software as part of their routine operations – each of these companies introduces hundreds and even thousands of changes to their software every single day.

The proliferation of cloud based computing means that most system administrators may never even get to set foot in a data center! Traditional system administration involved ordering hardware, and "racking and stacking" that hardware in a data center. Now, you increase your capacity by issuing an API call or by merely clicking a button on the cloud provider's web page.

Devops is a term that has come increasingly to the forefront in the past few years. Devops is meant to be a way for operational and development teams to work together to speed up the creation and deployment of software. While I don't explicitly address this book to devops professionals, many, if not all of the topics that I discuss in this book are highly relevant to devops professionals. Sysadmins have always worked with devops, through legend has it that the relationship was somewhat acrimonious, with sysadmins being accused of undue zealotry in maintaining their fiefdoms. Today, there's absolutely no chance of such a vast schism between the two groups: either the two groups swim together, or they both sink!

If you're indeed a devops person who's interested in the development side, I'd like to clarify that while I do discuss in depth several new concepts and tools geared towards development, the book is squarely aimed at systems administrators, or those who want to learn about the systems admin side of the equation.

The main thrust of the book is to discuss and explain the key principles and the processes underlying the way modern businesses are architecting and supporting robust highly scalable infrastructures. Sometimes I do show how to install and get going with a tool, but the focus is really on the role the tools play, and how to integrate them into your work as an effective Linux systems administrator.

The best way to benefit from the book is to absorb the main conceptual principles underlying modern systems administration – any tools I discuss in the book are there mostly to illustrate the concepts. Progress is very rapid in this area and new techniques and new tools are introduced all the time. Tools which are popular today for performing certain tasks may be easily supplanted by better tools with short notice. Thus, focusing on the conceptual side of things will help you in the long run, by

showing you how to solve major problems confronted by organizations in developing software and managing web sites, scaling and performance, etc.

Motivation for the new System Administration Strategies and Tools

Several important goals are at the root of most of the newer developments in the software building process and therefore, in systems administration, such as:

- Speed to market (reduce the software cycle time)
- Better software (increase the software quality)
- Cost aware architectures (lower the costs of deploying and maintaining infrastructure)

Modern system administration concepts and strategies that I enumerate throughout this book, such as rapid infrastructure deployment, continuous integration of applications, automated testing and push–button deployments are all a means to achieving these key goals.

Traditional IT practices are conservative and risk averse, often shying away from any changes that might increase the volatility of their operations. Today, Companies like Etsy and Facebook deploy to production numerous times every single day. Netflix has its Chaos Monkey tool that lets it randomly terminate production instances or introduce latency into the application. Having realized that frequent changes bring more benefits than pain, these and other organizations have endeavored in recent years to make their systems antifragile.

Strong testing and QA pipelines and continuous deployment strategies are what enable modern companies to reduce the risk of individual changes. Amazon for example can automatically rollback the software changes in case a deployment doesn't quite pan out. Obviously the company's doing something right, since less than 0.001% of Amazon's deployment result in an outage, although on average it deploys a new change almost every 10 seconds.

Problems with Traditional Systems Administration

Traditional systems administration concepts date back to several decades, predating the advent of major technological innovations such as the internet, cloud computing, newer networking models, and many others. While the guts of administering systems remains effectively the same as always, the job requirements and what management expects from system administrators have slowly but irrevocably changed over the past few years.

Let's review the changes in some areas of traditional systems administration to learn why one ought to change their basic approach to systems administration in many ways compared to the traditional way they administered systems, and how modern tools and techniques are transforming the very nature of the system administrator's role in a modern IT environment.

Monitoring

Modern systems management requires monitoring, just as the traditional systems administration did. However, traditional monitoring was mostly limited to tracking the uptime of servers and whether key services were running on it. Whereas in the past, one focused more on system metrics, today, application or service metrics have come to play an equal or even larger role in ascertaining the health and well-being of systems – keeping the end user happy is today's corporate mantra in all areas of business, including the area of systems administration.

Faster and Frequent Deployments

In today's fast paced web environments, deployments aren't something that are massive and infrequent. It's actually the opposite - deployments are small and quite frequent. Following is a sample of the production environment for Amazon (circa 2014): 11.6 seconds: Mean time between deployments (weekday) 1,079 : Max # of deployments in a single hour 10,000 : Mean # of hosts simultaneously receiving a deployment 30,000 : Max # of hosts simultaneously receiving a deployment

The Image Sprawl problem

While using "golden images" is definitely a superior approach to traditional deployments which are always done from scratch, they do tend to exacerbate the problem of image sprawl. Image sprawl is where multiple images are in deployment, usually in different versions. Images become unwieldy and management becomes chaotic. As the number of images grows, you'll find yourself performing regular manual changes, which tend to lead to deviation from the gold standard. A gold standard in this context refers to a known set of good configuration. Configuration management, the primary objective underlying gold standard usage, becomes hard over time.

Agile Development Methodologies and the System Administrator

Agile Operations is the counterpart to agile development practices, which involve strategies like Kanban and scrum, along with frequent, small code rollouts. The high frequency of code changes means that the operations teams can't be in relative isolation from the development teams, as in the days past. The rigid barriers between the two teams have been gradually coming down due to the high degree of cooperation

and interaction between development and operations that the agile development methodologies require.

Cloud environments

Systems administration practices that work well for a company's data center aren't really usable in toto when you move to a cloud environment. Storage and networking both are fundamentally different in the cloud, especially an external one. System administrators are expected to understand how to work with both public clouds such as AWS, Azure and Rackspace, as well as know how to set up their own private cloud based environments using something like OpenCloud.

Impact of Big Data

Traditional warehouses can't scale beyond a certain point, regardless of how much hardware and processing capacity you throw at them. The advent of the Web and the consequent deluge of data required a different paradigm, and distributed processing turned out to be the best approach to solving problems posed by big data. Hadoop is here to stay as a platform for storing and analyzing big data. Administrators must know how to architect and support Hadoop and other big data environments.

Manual Operations without automation

Traditional systems are to the most extent still run with a heavy dose of manual operations. While script based systems administration has been around for many years, most administrators still perform their duties as they did 30 or even 40 years ago – by hand, one operation after the other. Consequently, change management is slow and there are plenty of opportunities to make mistakes.

New trends in system administration and application development include the following:

- Infrastructure automation (infrastructure as code)
- Automated configuration management
- Virtualization and containerization
- Microservices
- Increasing use of NoSQL and caching databases
- Cloud environments– both external and internal
- Big data and distributed architectures
- Continuous deployment and continuous integration

In the following sections, I briefly define and explain the concepts behind the trends in modern system administration. In the following chapters of the book, I discuss most of these concepts and the associated tools in detail.

Automated Infrastructure Management

Configuring the environment in which applications run is just as important as configuring the application itself. If any required messaging systems aren't configured correctly for example, an application won't work correctly. Configuring the operating system, the networks, database, and web application servers is critically important for an application to function optimally.

The most common way of configuring systems is to do so as you go, That is, you first install the software and manually edit the configuration files and settings until the darn software works correctly. No record is made of the prior state of the configurations as you iterate through successive configuration states. This means that you can't revert easily to the last "good" configuration if any changes go bad.

A fully automated process offers the following benefits:

- It keeps the cost (in terms if effort and delays) of creating new environments very low.
- Automating the infrastructure creation (or rebuilding) process means then when someone playing a critical role in a team leaves, nothing really stops working or stymies your attempts to fix it.
- Automating things imposes an upper bound on the time it takes to get back to a fully functioning state of affairs.
- A fully automated system also helps you easily create test environments on the fly – plus, these development environments will be exact replicas of the latest incarnation of the production environment.
- An upgrade to your current system won't automatically lead to an upheaval. You can upgrade to new versions of systems with no or minimal downtime

You can use a tool such as Chef, Puppet (or Ansible) to automate the configuration of the operating system and other components. As explained earlier, all you need to do is to specify which users should have access to what, and which software ought to be installed. Simply store these definitions in a VCS, from where agents will regularly pull the updated configuration and perform the required infrastructure changes. You gain by not having to manually do anything, plus, since everything is flowing through the VCS, the changes are already well documented, providing an effective audit trail for the system changes.

Automating Redundant Configuration Work

Setting up a new infrastructure or using an existing but unwieldy infrastructure setup isn't a trivial task for new system administrators. While the laying out of the infrastructure itself is pretty straightforward, it usually involves steps that are inherently prone to simple errors. And once you set up an infrastructure with built-in errors, a lot of times you're forced to live with those errors as long as that infrastructure is in place.

Redundant work and duplication of tasks occupies the scarce time of administrators. Manual installation and configuration of infrastructure components such as servers and databases isn't really practical in large scale environments that require you to setup hundreds and even thousands of servers and databases.

Configuration management software grew out of the need to eliminate redundant work and duplicated efforts. The configuration tools help automate infrastructure work. Instead of manually installing and configuring applications and servers, you can simply describe what you want to do in a text-based format. For example, to install an Apache Web Server, you use a configuration file with the declarative statement:

```
All web servers must have Apache installed.
```

Yes, as simple as this statement is, that's all you'd have to specify to ensure that all web servers in a specific environment have the Apache web server installed on them.

Automating infrastructure and application deployment requires more than one simple tool. There's some overlap among the different types of automation tools, and I therefore briefly define the various types of tools here:

- Configuration management tools let you specify the state description for servers and ensure that the servers are configured according to your definition, with the right packages, and all the configuration files correctly created.
- Deployment tools: deployment tools generate binaries for the software that an organization creates, and copies the tested artifacts to the target servers and starts up the requested services to support the applications.
- Orchestration tools: orchestration of deployment usually involves deploying to remote servers where you need to deploy the infrastructure components in a specific order.
- Provisioning tools: provisioning is the setting up of new servers, such as spinning up new virtual machine instances in the Amazon AWS cloud.

Some tools are purely meant for one of the four purposes I listed here, such as deployment for example, where a tool such as Jenkins or Hudson performs purely integration and deployment related functions. Most tools perform more than one

function. A tool such as Ansible for example, can perform all four of these things – configuration management, deployment, orchestration, and provisioning very well.

Configuration Management

Configuration management is how you store, retrieve, identify, and modify all artifacts and relationships pertaining to a project. Modern configuration management practices are an evolution of the traditional system administration strategies to manage complex infrastructures.

Until recently, scripting was the main means of automating administrative tasks, including configuring systems. As infrastructure architectures become ever more complex, the number and complexity of the scripts used to manage these environments also grew complex, leading to more ways for scripted procedures to fail. In 1993, the first modern configuration management (CM) system, CFEngine, was started to provide a way to manage UNIX workstations. In 2005 Puppet was introduced and soon become the leader in the market until Chef was introduced in 2009. Most CM systems share the same basic features:

- Automation of the infrastructure (infrastructure as code)
- Ruby (or a similar scripting language) as the configuration language
- Extensibility, customizability, and the capability to integrate with various other tools
- Use of modular, reusable components
- Use of thick clients and thin servers – the configuration tools perform most of the configuration work on the node which is being configured, rather than on the server that hosts the tools
- Use of declarative statements to describe the desired state of the systems being configured

Good configuration management means that you:

- Can reproduce your environments (OS versions, patch levels, network configuration, software and the deployed applications) as well as the configuration of the environment
- Easily make incremental changes to any individual environment component and deploy those changes to your environment
- Identify any changes made to the environment and be able to track the time when the changes were made, as well as the identity of the persons who made the changes.

- Easily make necessary changes and also obtain the required information, so as to decrease the software cycle time

when you handle server configuration like software, naturally you can take advantage of a source code management system such as Git and Subversion to track all your infrastructure configuration changes

Popular configuration management tools include the following:

- Chef
- Puppet
- Ansible
- Capistrano
- SaltStack

Infrastructure as Code

The term infrastructure as code is synonymous with configuration management. Tools such as Chef transform infrastructure into code. Other CM tools such as Chef, Puppet, Ansible and Saltstack also use the infrastructure as code approach to configure the infrastructure. All these tools are automation platforms that let you configure and manage an infrastructure, both on-premises as well as in the cloud.

Infrastructure as code is really a simple concept that lets your infrastructure reap the benefits of automation by making an infrastructure versionable, repeatable, and easily testable. CM tools let you fully automate your infrastructure developments as well as automatically scale infrastructure, besides handling infrastructure repairs automatically (self-healing capabilities).

Applications typically lock down their configuration and don't allow untested adhoc changes over time. Why should the administrator do any different? You must learn to treat environmental changes as sacrosanct, and work through a structured and formal build, deploy, and test process the same way as developers treat their application code.

While you can't always build expensive test systems that duplicate fancy production systems, you do need to deploy and configure these test environments the same way as you do the production systems.

Let me illustrate the dramatic difference between the old approach and the configuration tools methodology. Following is a Chef recipe that shows how to change permissions on a directory:

```
directory '/var/www/repo' do
mode '0755'
owner 'www'
group 'www'
end
```

If you were to do the same thing using a scripting language, your script would need to incorporate logic for several things, such as checking for the existence of the directory, and confirming that the owner and group information is correct, etc. With Chef, you don't need to add this logic since it knows what to do in each situation.

Site reliability engineering is a term that is increasingly becoming popular in the system administration world. In conventional shops, if a disaster occurs, you need to rebuild everything from scratch. In fact, almost every company maintains an up-to-date disaster recovery plan and even performs dry runs on a regular basis to test their disaster recovery mechanisms. CM helps you quickly restore services following a disaster, since it automates all deployments. Same goes for scheduled upgrades as well, as CM tools can build out the upgraded systems automatically. Bottom line: shorter outages and shorter downtimes for the end users.

Modern Scripting Languages and Databases

Often system administrators in the current milieu wonder what programming languages they ought to learn. A good strategy would be for administrators to become adept at a couple, or even one scripting language, say Python or Ruby. In addition, they should learn new languages and frameworks because in today's world you'll be dealing with numerous open source tools. In order to work efficiently with these tools and adapt them to your environment and even make enhancements, you need to know how these tools are constructed.

There's really no one scripting language that can serve as a do-it-all language. Some languages are better for handling text and data while others maybe be more efficacious at working with cloud vendor APIs with their specialized libraries.

Essential Programming Skills for the System Administrator

Traditionally system administrators used a heavy dose of shell scripting with Bash, Awk and Sed, to perform routine Linux administration operations such as searching for files, removing old log files and managing users, etc. While all the traditional scripting skills continue to be useful, in the modern Linux administration world, a key reason for using a programming language is to create tools. Also, most of the exciting open source tools such as Chef and Puppet are written in languages such as

Ruby, so it's imperative that you learn the modern languages such as Ruby and GO (Golang), if you want to be an effective modern system administrator.

New technologies such as Linux Containers, Docker. Packer and etcd all use GO for their internal tooling.

The Rise of NoSQL Databases

Traditional infrastructures and applications mostly relied (and a lot of them still do) on relational databases such as Oracle, MySQL, PostgreSQL and Microsoft SQL Server for running both online transaction processing systems as well as for data warehouses and data marts.. In large environments, there are usually dedicated database administrators (DBAs) hired expressly to manage these databases (and these databases do require a lot of care and feeding!), so sysadmins usually got into the picture during installation time and when there was some kind of intractable server, storage, or network related performance issue that was beyond the typical DBA's skill set.

Two of the most common relational databases that are used today on the internet are MySQL and PostGreSQL. In addition to these traditional databases, modern system administrators must also be comfortable working with NoSQL databases. NoSQL databases have become common over the past decade, owing to the need for handling document storage and fast clustered reading

Today, it's very common for organizations, especially small and medium sized companies, to expect their system administrators to help manage the NoSQL databases, instead of looking up to a dedicated DBA to manage the databases. For one thing, the new genre of NoSQL databases requires far less expert database skills (such as SQL, relational modeling etc.). Furthermore, companies use a bunch of different databases for specialized purposes and the size of the databases in most organizations doesn't call for a dedicated DBA for each of these databases. So, managing these databases, or at the least, worrying about their uptime and performance is falling more and more in the lap of the system administrator.

Chapter 3 explains the key concepts behind the NoSQL databases such as MongoDB.

Caching

In addition to NoSQL databases, the use of caching has come to occupy a central place. Modern application stacks contain several points where you can use caching to vertically scale the applications.

Setting up remote caching clusters using open source caching solutions such as Memcached or Redis is pretty straightforward but scaling and managing failure recovery scenarios isn't. You can let Amazon Elastic Cache to set up a hosted cached cluster with Memcached or Redis. Microsoft Azure handles your Redis replication and automatic failover setup in a few easy clicks. Chapter 3 explains caching and the use of external distributed caching servers such as Memcached and Redis. Se

Content Delivery Networks

Content Delivery Networks (CDNs) are a network of distributed servers that deliver web content to users based on the geographical location of the user and the origin of the web pages. CDNs are efficient in speeding up the delivery of content originated by websites with a heavy traffic and a wide reach. CDNs also protect you against large increases in the web traffic. CDNs use content delivery servers that cache the page content of popular web pages. When users access these pages, the CDN redirects the user to a server that's closest to that user.

Chapter 3 explains the concepts underlying a CDN and how it works.

IT Orchestration

IT Orchestration goes beyond configuration, and enable you to reap the benefits of configuration management at scale. Orchestration generally includes things such as the following:

- Configuration
- Zero downtime rolling updates
- Hot fixes Let's say you're configuring multiple instances of a web server with a CM tool, each of them with a slightly different configuration. You may also want all the servers to be aware of each other following a boot up. Orchestration does this for you. An orchestration framework lets you simultaneously execute configuration management across multiple systems.

You can use a CM tool for orchestration, but a more specialized tool such as Ansible is excellent for IT orchestration. If you were to use just a CM tool, you'd need to add on other tools such as Fabric and Capistrano for orchestration purposes. With a dedicated orchestration tool such as Ansible, you won't need the other tools. Ansible also helps you manage cloud instances without modification. Note that while Ansible is really an orchestration tool, since it permits the description of the desired state of configuration, you can also consider it a CM tool.

Often companies use Ansible along with Puppet or Chef as the CM tool. In this case, Ansible orchestrates the execution of configuration management across multiple sys-

tems. Some teams prefer to use just Ansible for both configuration and orchestration of an infrastructure.

Provisioning with Vagrant

It's quite hard to develop and maintain web applications. The technologies, code and the configuration keep changing all the time, making it hard to keep your configuration consistent on all servers (prod, staging, testing, and dev). Often you learn that a configuration for a web server or a message queue is wrong, only after a painful production screw up. Virtualized development environments are a great solution for keeping things straight. You can set up separate virtualized environments for each of your projects. This way, each project has its own customized web and database servers, and you can set up all the dependencies it needs without jeopardizing other projects.

A virtual environment also makes it possible to develop and test applications on a production-like virtual environment. However, virtual environments aren't a piece of cake to set up and require system administrator skillsets. Vagrant to the rescue!

Vagrant is a popular open source infrastructure provisioning tool. Vagrant lets you easily create virtualization environments and configure them using simple plain text files to store the configuration. Vagrant makes your virtual environments portable and shareable with all team members. With Vagrant, you can easily define and create virtual machines that can run on your own system.

```
You can generate virtual machines with Oracle's VirtualBox or VMware as the providers. Once you've
```

```
$ vagrant up
```

Since Vagrant integrates with various hypervisor and cloud providers, you can use it to provision both an on premise virtual infrastructure as well as a full blown cloud infrastructure.

Vagrant and Configuration Management Tools

Earlier, you saw how you can use CM tools such as Chef and Puppet to provision and configure infrastructure components such as web servers and databases. You can use Vagrant with a configuration management tool such as Chef to bring up a complete environment, including a virtual infrastructure. Once you provision the cloud or virtual infrastructure through Vagrant, you can use Chef to deploy and configure the necessary application servers and databases on the virtual (or cloud) servers.

A common use case for Vagrant is the fast creation of a disposable infrastructure and environment for both developers and testers.

Provisioning - Automatic Server Deployment

With the proliferation of server farms and other large scale server deployment patterns, you can rule out old-fashioned manual deployments. PXE tools such as Red Hat Satellite for example, help perform faster deployments, but they still aren't quite adequate, since you need to perform more work after deploying the operating system to make it fully functional.

You need some type of a deployment tool (different from a pure configuration tool such as Puppet or Chef), to help you with the deployment. For virtual servers, VMware offers CloudForms, which helps with server deployment, configuration management (with the help of Puppet and Chef) and server lifecycle management. There are several excellent open source server deployment tools such as the following:

- Cobbler
- Razor
- Crowbar
- Foreman

In Chapter 6, I show how tools such as Razor and Cobbler help with the automatic deployment of servers.

Server (hardware) Virtualization

In the old days, all applications ran directly on an operating system, with each server running a single OS. Application developers and vendors had to create applications separately for each OS platform, with the attendant increase in effort and cost. Hardware virtualization provides a solution to this problem by letting a single server run multiple operating systems or multiple instances of the same OS. This of course lets a single server support multiple virtual machines, each of which appears as a specific operating system, and even as a particular hardware platform.

Most application libraries and server delivery platforms usually were installed at the system level and thus disparate applications would often find conflicts that were resolvable only by using dedicated systems for each application. Virtual machines solve this problem by hosting dedicated operating systems as virtual environments on top of another OS.

Hardware virtualization separates hardware from a single operating system, by allowing multiple OS instances to run simultaneously on the same server. Hardware virtualization simulates physical systems, and is commonly used to increase system density and hike the system utilization factor. Multiple virtual machines can share the

resources of a single physical server, thus making fuller use of the resources you've paid for.

Containerization – the New Virtualization

Virtual machine technology has been around for a good while and folks understand how it works: virtual machines contain a complete OS and run on top of the host OS. You can run multiple virtual machines, each with a different OS, on the same host. Containers share some features with virtualization but aren't the same as virtual machines. Under containerization, both the host and the container share the same kernel, meaning containers are based on the same OS as their host. However, since containers share the kernel with the host, they require fewer system resources.

Container virtualization is newer than virtualization and uses software called a virtualization container that runs on top of the host OS to provide an execution environment for applications. Containers take quite a different approach from that of regular virtualization, which mostly use hypervisors.

The goal of container virtualization isn't to emulate a physical server with the help of the hypervisor. Containers are based on operating system virtualization – all containers share the same OS kernel and their isolation is implemented within that one kernel..All containerized applications share a common OS kernel and this reduces the resource usage since you don't have to run a separate OS for each application that runs inside a container on a host. Processes running inside a container have a small hook into the underlying OS kernel.

Virtualization systems such as those supported by VMware let you run a complete OS kernel and OS on top of a virtualization layer known as a hypervisor. Traditional virtualization provides strong isolation among the virtual machines running on a server, with each hosted kernel in its own memory space and with separate entry points into the host's hardware.

Since containers execute in the same kernel as the host OS and share most of the host OS, their footprint is much smaller than that of hypervisors and guest operating systems under traditional virtualization. Thus, you can run a lot more containers on an OS when compared to the number of hypervisors and guest operating systems on the same OS.

Containers are being adopted at a fast clip since they are seen as a good solution for the problems involved in using normal operating systems, without the inefficiencies introduced by virtualization. New lightweight operating systems such as CoreOS have been designed from the ground up to support the running of containers.

Containers don't have the main advantage provided by hardware virtualization such as the virtualization provided by VMware, which can support disparate operating systems. This means, for example, that you can't run a Windows application inside a Linux container. Containers today are really limited to the Linux operating system only.

Docker as a solution for Image sprawl

Docker is primarily a solution for managing image sprawl. Docker is lightweight and its images are layered, allowing you to easily iterate on the images. While Docker can fix many of the problems inherent in the golden image strategy, it doesn't supplant configuration management tools such as Chef and Puppet. After all, you can use the CM tools to install and configure Docker itself, as well as the Docker containers. Containers require orchestration and deployment support, which the CM tools excel in.

Docker and Containerization

Docker makes it easy to overcome the headaches involved in managing containers. Docker is an application that offers a standard format in which to create and share containers. Docker didn't materialize out of thin air: it extends the contributions made by LXC, cgroups and namespaces to simply the deployment and use of containers.

As mentioned earlier, it was Google that started developing CGroups for Linux and also the use of containers for provisioning infrastructure. In 2008 the LXC project was started, combining the technology behind CGroups, kernel namespaces and chroot, as a big step forward in the development of containers. However, using LXC to run containers meant a lot of expert knowledge and tedious manual configuration.

It was left to Docker, however, to complete the development of containers to the point where companies started adopting them as part of their normal environment, starting around 2012. Docker brought containers from the shadows of IT to the forefront.

Docker did two main things: it extended the LXC technology and it also wrapped it in user friendly ways. This is how Docker became a practical solution for creating and distributing containers. Docker makes it easy to run containers by providing a user friendly interface. In addition, Docker helps you avoid the reinventing of the wheel, by providing numerous prebuilt public container images you can download and get started.

Docker Container Orchestration and Distributed Schedulers

Managing processes on a single server is easy with the help of the Linux kernel and the init system. Managing and deploying a large number of containers isn't a trivial concern. When you deploy containers on multiple hosts, you not only need to worry about the deployment of the containers, but you also need to concern yourself with the complexities of inter-container communications and the management of the container state (running, stopped. failed, etc.). Figuring out where to start up failed servers or applications and determining the right number of containers to run, all are complex issues.

Message Oriented Middleware (Message Buses)

The prevalence of service oriented architectures means that administrators build systems that connect multiple services, in a loosely coupled format. These systems are distributed by definition, and hence need a way for the individual components to communicate efficiently at high speeds. Messaging buses (also called message brokers or MOM (Message Oriented Middleware), are communication platforms that enable applications and services to communicate in distributed architectures. Streams of messages pass through the bus and are acted upon by the services (worker nodes).

Popular message buses include the following:

- RabbitMQ
- ZeroMQ
- Celery

Containers decouple processes from the servers on which the containers run. While this offers you great freedom in assigning processes to the servers in your network, it also brings in more complexity to your operations, since you now need to:

- Find where a certain process is running right now
- Establish network connections among the containers and assign storage to these containers
- Identify process failures or resource exhaustion on the containers
- Determine where the newly started container processes should run

Container orchestration is the attempt to make container scheduling and management more manageable. Distributed schedulers are how you manage the complexity involved in running Docket at scale. You simply define a set of policies as to how the applications should run and let the scheduler figure out where and how many instances of the app should be run. If a server or app fails, the scheduler takes care of

restarting them. All this means that your network becomes a single host, due to the automatic starting and restarting of the apps and the servers by the distributed scheduler.

The bottom line is to run the application somewhere without concerning yourself with the details of how to get it to run somewhere. Zero downtime deployments are possible by launching new application versions along the current version, and by gradually directing work to the new application.

There are several container orchestration and distributed scheduling tools available, as the following sections explain.

FLEET

Fleet works with the system daemon on the servers in order to perform as a distributed init system. Thus, it's best used in operating systems with systemd, such as CoreOS and others. In addition, you need etcd for coordination services.

KUBERNATES

Kubernates, an open source tool initiated by Google, is fast becoming a leading container orchestrator tool. Kubernates helps manage containers across a set of servers. Unlike Fleet, Kubernates demands far fewer requirements to be satisfied by the OS and therefore you can use it across more types of operating systems than Fleet. Kubernates contains features that help with deploying of applications, scheduling, updating, scaling, and maintenance. You can define the desired state of applications and use Kubernate's powerful "auto features" such as auto-placement, auto-restart, and auto-replication to maintain the desired state.

Apache Mesos

Apache Mesos is considered by some as the gold standard for clustered containers. Mesos is much more powerful than Kubernates but requires you to make more decisions to implement it as well! Mesos has been around even before Docker became popular. Mesos is a framework abstraction and lets you run different frameworks such as Hadoop as well as Docker applications (there are projects in place to let even Kubernates run as a Mesos framework) on top of the same cluster of servers. Mesosphere's Marathon framework and the Apache Aurora project are the two frequently used Mesos frameworks that support Docker well.

Apache Mesos is a mature technology used by well-known internet companies to support their large scale deployments. Kubernates and Apache Mesos provide somewhat similar features as what the public cloud vendors themselves offer with their proprietary technology.

Swarm

Docker itself provides a native clustering tool named Swarm, which lets you deploy containers across a large pool. Swarm presents a collection of Docker hosts as a single resource. While Swarm is really lightweight and has fewer capabilities than either Kubernates or Mesos, it's adequate for many purposes. You can use a single Docker Swarm container to create and coordinate container deployment across a large Docker cluster.

Cluster Management and Cluster Operating Systems

Increasingly, distributed services are using clusters for both redundancy as well as the scaling benefits offered by the cluster based architectures. Clusters provide numerous benefits but also bring their own unique problems. Chief among these is the efficient allocation and scheduling of resources to the various services running in the cluster. Cluster operating systems are the answer to this problem, with Apache Mesos being the most well-known of these types of systems. Here is a short list of the most important cluster operating systems.

- Mesos is a kernel/operating system for distributed clusters. It supports both .war files and Docker containers.
- Marathon is a scheduler that helps schedule and run jobs in a Mesos cluster. It also creates a private Platform –as-a-Service on top of the Mesos framework.
- Fleet is for Docker containers
- YARN (Yet Another Resource Negotiator) is the processing component of Apache Hadoop and together with the storage component of Hadoop - HDFS (Hadoop Distributed File System), forms the foundation of Hadoop 2.

Each framework in a cluster has different computing requirements. Companies must therefore run these different frameworks together and share the data and resources among them. Any cluster resource manager in a cluster should be able to support the goals of isolation, scalability, robustness and extensibility.

Mesos, which became a top-level Apache project in 2013, is becoming increasingly popular as a cluster manager that helps improve resource allocation by enabling the dynamic sharing of a cluster's resources among different frameworks. Twitter and Airbnb are among the production users of Mesos.

Mesos does the same for the data center as what the normal operating system kernel does for a single server. It provides a unified view and easy access to of all the cluster's resources. You can use Mesos as the center piece for your data center applications, and make use of its scalable two-phase scheduler, which avoids the typical problems you experience with a monolithic scheduler. Chapter 10 discusses Mesos in detail.

Distributed jobs run across multiple servers. System administrators, for course, use tools to monitor a set of servers and use configuration tools to perform the server configuration updates etc. You can also use CI (Continuous Integration) tools such as Jenkins to manage some general infrastructure management tasks. But the best way to schedule both ad-hoc and scheduled jobs across multiple servers is to use a remote command execution tool. All the following can help you execute commands across multiple hosts:

- Fabric
- Capistrano
- Mcollective

MCollective for example, is a great tool for parallel job execution that you can use to orchestrate changes across a set of servers in near real-time.

There are simple parallel execution tools based on the Parallel Distributed Shell (pdsh), an open source parallel remote command execution utility, or you can even script one yourself. However, these tools are primitive because they loop through the system in order, leading to the time drift issue. They also can't deal with deviations in responses and make it hard to track fatal error messages. Finally you can't integrate these tools with your CM tools. A tool such as Mcollective overcomes all these drawbacks of the plain vanilla parallel execution tools and allows you execute commands parallelly on thousands of servers belonging to different platforms. Chapter 6 shows how MCollective works.

Version Control Systems

Version control is the recording of changes to files so you can recall specific versions of the files at a later time. Although version control systems are mostly used by developers (and increasingly by system administrators), source code isn't the only thing you can version – the strategy of keeping multiple versions of the same document in case you'll need them later on is applicable to any type of document.

Version control systems offer numerous benefits such as the following:

- Take a project (or some of its files) back to a previous state
- Find out which user made the modifications that are giving your team a headache
- Easily recover from mistakes and the loss of files NOTE: Key practices such as continuous integration and automated deployments depend critically on the usage of a central distributed version control repository. Really, with benefits

such as these and very little overhead, what's there to argue about version control systems?

Application development teams have been using version control systems for a number of years now, to track and maintain application code. However, it's important to version your infrastructure, configurations and databases, along with you application code. You must script all your source artifacts.

Just as the application developers do with their application code, you must have a single source of truth for your systems, including servers, databases and web servers. This results in letting you quickly resolve problems and easily recreate known good states of your infrastructure components. You don't need to waste precious time figuring out which versions of your application code should be paired with which environment.

The most commonly used version control tools today are the following:

- Git
- Perforce
- SVN (subversion)

Of the three tools, Git has become enormously popular, for several reasons.

When you store your configuration definitions in a version control system, you can also set up things so that any configuration changes you commit automatically trigger actions to test and validate changes. This is how continuous integration is triggered for infrastructure changes, so the changes are automatically tested and validated. You therefore integrate a CI or CD tool such as Jenkins, or TeamCity with the CVS. Since configuration definitions are inherently modular, you can run implementation actions on only those parts of the infrastructure code that you've changed.

Version control systems (VCSs) are nothing new – most of us are quite familiar with these systems, variously referred to as source control systems, revision control systems or source code management systems. However, early version control systems such as CVS (Concurrent Version System) have been supplanted over the past several years by far more powerful open source tools such as Git, Mercurial and Subversion.

Chapter 7 discusses version control systems, including Git and GitHub.

Continuous Integration and Continuous Deployment

One of the ways in which system administrators can help developers be more productive is by helping developers spend most of their time writing code and creating or enhancing software, instead of running around fixing problems cause by bad

builds and merges. Broadly speaking, administrators can help developers in the following two ways:

- Provide performance metrics and test results to developers so they can evaluate their work
- Help build environments for developers to work in, ideally with the same configuration as the production systems

The terms rollout, releasing, and deploying are often used synonymously to refer to the process of making changes available to internal or external end users.

In traditional application deployment, there are numerous steps between generating code and the product into the hands of the users (internal or external). These steps include:

- Writing new source code or modifying existing code
- Committing the source code
- Building binaries from the modified code
- Performing quality assurance (QA) testing
- Staging the application
- Deploying the application to production

Not only are there several clearly demarcated steps along this process, with different owners and stakeholders, there's also quite a bit of manual work involved in those processes. This has the rather unsettling effect that while you're fixing some bugs in your code, the tedious manual processes you employ for testing the changes and deploying it may introduce their own bugs into the software! Automation of course, is the way around these potential errors and it also speeds up the entire pipeline. Continuous integration is the broad name given to this automating of the software build and deployment process.

the primary purpose behind a CI or CD system is to catch bugs when they're young, that is, fairly early in the development and testing process.

Benefits Offered by CI

Continuous integration offers several benefits, as summarized here:

- Enhanced reliability of the artifacts that you deploy
- Promotion of automation and reduction in the number of manual processes
- Early identification and remediation of software bugs
- More frequent software updates

Steps involved in CI

Continuous integration involves a sequence of steps that must be performed in order, to get to the final stage of deploying the software. In a nutshell, CI involves the following steps or procedures:

- Developers check in code to a common VCS on a regular basis, usually at least once a day. Everything that an application needs to run – source code, database migration scripts, etc. are checked in.
- CI tools such as Jenkins and TeamCity run automated builds whenever there are changes in the VCS.
- The automated builds include the testing of the code checked in by the developers, such as unit tests, code coverage and functional tests.
- A build is classified as a fully integrated build after it passes all tests.
- The CI server provides the tested artifacts (same as binaries or executables) for download. You can alternatively use an external artifact repository such as Artifactory, Nexus or Nuget.
- The CI server provides visibility through a dashboard into the broken builds and test failures, and automates communications to the team members for actions to fix the problems.
- One of the key requirements of CI is to automate deployments to test environments, where you can run automated functional tests by deploying the previously generated tested artifacts.

Continuous Integration and Continuous Deployment

There's often some confusion between the two similar sounding terms Continuous Integration (CI) and Continuous Delivery (CD). Here's how the two terms differ:

- Continuous integration, also referred sometimes as Continuous Staging, involves the continuous building and the acceptance testing of the new builds. The new

builds aren't automatically deployed into production. They're introduced into production manually, after approval of the acceptance testing process.

- Continuous delivery is where new builds are automatically pushed into production after acceptance testing.

In general, organizations use CI when they first start out along the road of continuous testing and deployment. They need to develop an extremely high degree of confidence in their build strategies and pipelines before moving to a CD approach. In any case, since both CI and CD really use the same set of processes and tools, you often are OK regardless of what you call the deployment approach (CI or CD).

Developers unit test the applications when they build or modify applications. Regardless of whether they use Test Driven Development (TDD) strategies, unit testing help verify small chunks of code at a time with tools such as Junit and RSpec, which speed up the writing of unit tests. A CI tool such as Jenkins, Hudson or Travis Continuous Integration (Travis CI) lets you create efficient build pipelines that consist of the build steps for the individual unit tests. In order to be promoted to the staging phase, the source code, after it's built into a binary package, must pass all the unit tests you define in the test pipeline. The compiled code without errors is called a build artifact and is stored in a directory that you specify.

Continuous Application deployment

Application deployment tools let you automate releases. These tools are at the center of continuous delivery, along with the continuous integration frameworks such as Jenkins. Capistrano is a deployment library that's highly popular as an application deployment tool. You can also consider other application deployment automation tools such as:

- Ansible
- Fabric
- Jenkins

Automating deployment speeds up the flow of software deployment. Many organizations have automated their software deployments in a way that enables them to introduce software changes several times every day. Automated deployment reduces the software cycle times by removing the human error component from deployments. The end result is fast and frequent, high quality deployments. Anyone with the appropriate permissions can deploy software as well as its environment by simply clicking a button.

It's quite common for project members to delay acceptance testing until the end of the development process. Developers may be making frequent changes and even

checking them in, and running automated unit tests. However, in most cases there's no end to end testing of the application until the very end of the application. Even the unit tests are of dubious value at times, since they aren't tested in a production-like environment. Instead, the project managers often schedule an elaborate and time consuming integration testing phase when all the development concludes. Developers merge their branches and try to get the application working so the testers can put the app through the paces of acceptance testing.

What if you can spend just a short few minutes after adding new changes to see if the entire app works? Continuous integration (CI) makes this possible. Every time one of the developers commits a change, the complete application is built and a suite of automated tests is run against the updated complete application. If the change broke the app, the development team needs to fix it right away. The end result is that at any given time, the entire application will function as it's designed.

So, how is the "continuous" part of CI defined? Continuous in this context simply means that every time your team changes something and commits the change to your version control system. A continuous integration tool such as Jenkins integrates with a VCS such as Git, which lets you automatically submit code to Jenkins for compiling and testing executions soon as the developers commit new code to the VCS. Later, the developers and others can check the stages of the automated test and build cycles using the job history provided by Jenkins as well as other console output.

Continuous Integration results in faster delivery of software as your app is in a known functioning state at all times. When a committed change messes the app up, you'll know immediately and can fix it immediately as well, without having to wait for the lengthy integration phase of testing at the very end of development. It's always cheaper in terms of effort and time to catch and fix bugs at an early stage.

Tools for Implementing Continuous Integration

Although CI is really more a set of practices than a specific tool, there are several excellent open source tools such as the following:

- Hudson: has a large set of plugins that allow you to integrate it with most tools in the build and deployment environment.
- CruiseControl: this is a simple, easy to use tool.
- Jenkins: the most popular CI tool out there today with a huge community of users and numerous plugins for almost anything you may want to do with respect to CI/CD.

In addition, you can check out the following commercial CI servers, some of which have free versions for small teams.

- TeamCity (JetBrains): contains numerous out-of-the box features to let you get started easily with CI
- Go (ThoughtWorks): Is an open source tool that descends from one of the earlier CI server named CruiseControl. Delivery Pipelines are the strong feature of Go, and the tools excels at visualization and configuring the pipelines.
- Bamboo (Atlassian)

In chapter 8, which deals with continuous integration, I explain the concepts behind the popular CI tools Jenkins and Hudson.

Checking into the trunk or mainline means that you aren't checking into branches. CI is by definition impossible if you aren't checking into the trunk, since any code you check into a branch isn't integrated with the rest of the existing application code or the changes being made by the other developers.

Applications use both unit testing and acceptance testing on a regular basis to test software changes. Unit tests are for testing the behavior of small portions of the application and don't work their way through the entire system such as the database or the network. Acceptance tests, on the other hand, are serious business: they test the application to verify its full functionality as well as its availability and capacity requirements.

How did companies perform integration before they started adopting the new CI tools? Well, most teams used the nightly build mechanism to compile and integrate the code base every day. Over time, automated testing was added to the nightly build processes. Further along the road cane the process of rolling builds, where the builds were run continuously instead of scheduling a nightly batch process. Continuous builds are just what they sound like: as each build completes, the next build starts, using the application version stored in the VCS.

Log Management, Monitoring and Metrics

Traditional log management has involved poring over boring server logs with the intention of divining the root cause of system failures or performance issues. However, the advent of the web means that you now have other types of logs to deal with as well. Companies have lots of information that's trapped by the web logs of users, but most users aren't really ready to mine these humongous troves of data.

Traditional systems administrators might think it's odd to consider logs as the fount of business intelligence, but that's exactly what they've become in today's web based world. Logs today don't mean just old fashioned server and platform logs – they mean so much more. Log management in the contemporary sense mostly refers to the

management of logs of user actions on a company's websites, including clickstreams etc. These logs yield significant insights into user behavior, on which so much of a business's success depends. Thus, it's the business managers and product owners that are really the consumers of the products of the log management tools, not the system administrators.

The term logs is defined quite broadly in the context of log management, and not limited to server and web server logs. From the viewpoint of Logstash, any data with a time stamp is a log.

If you want to assess the stability and performance of your system, you need the system logs. A distributed system can produce humongous amounts of logs. You definitely need to use tools for collecting the logs as well as for visualizing the data.

There are great log management tools out there today that you can use to manage and profit from logs. Although originally tools such as Logstash were meant to aggregate, index and search server log files, they are increasingly being used for more powerful purposes. Logstash combined with Elasticsearch and Kibana (ELK) is a very popular open source solution for log analysis. ELK is increasingly being used as a real time analytics platform. It implements the Collect, Ship+Transform, Store and Display patterns. Here's what the three tools in the ELK stack do:

- ElasticSearch: a real-time distributed search and analytics engine optimized for indexing and based on Lucene, an open source search engine known for its powerful search capabilities
- Logstash: a log collection, analyzing and storing framework, used for shipping and cleaning logs
- Kibana: a dashboard for viewing and analyzing log data

Effective Metrics

Graphite is a popular open source tool for storing metrics. Graphite helps you build dashboards lightening quick for just about any metric that you can think of. Graphite is highly flexible, scalable, and is extremely easy to use.

Graphite lets you perform various types of actions with metrics, by providing a timestamp and value for the metrics. You can graph the metric and set alarms for it. Graphite offers an API that's quite easy to use. Developers can add their self-service metrics to the dashboards.

Proactive Monitoring

You need to have two entirely different types of monitoring for your systems. The first type of monitoring is more or less an extension of traditional server monitoring. Here, tools such as Nagios and New Relic Server provide you visibility into key areas bearing on system performance, such as capacity, CPU and memory usage, so you can fix problems when they rear their ugly head.

The second and more critical type of monitoring, especially for those organizations who are running complex web applications, are the application performance monitoring tools. Tools such as New Relic APM enable code-level identification and remediation of performance issues.

The goal in both types of monitoring is to let everybody view the available application and server performance data – so they can contribute to better decisions to improve/fix things.

Tools for Monitoring

There are several popular monitoring tools, including Nagios, Ganglia, statsd and Graphite. Let me briefly describe the Nagios and Ganglia tools here – I describe these and tools in detail in Chapter 7.

Nagios is a system and networking monitoring application that helps you monitor web applications, as well as resource utilization such as disk, memory and CPU usage. Nagios is very popular as an open-source monitoring tool. Nagios doesn't include any monitoring scripts when you install it. It's simply a plug-in scheduler and execution program. You turn Nagios into a true monitoring system by using various plug-ins that check and report on various resources. Nagios uses exit codes from the various plug-ins to determine subsequent actions. When a plug-in script exits with a non-zero exit code, Nagios generates and sends alert notifications to you.

Ganglia is a distributed monitoring system with powerful graphical capabilities to display the data it collects. While Nagios shows you if an application or server is failing preconfigured health checks, Ganglia shows you the current and historical picture as to the resource usage and performance of applications and hosts. As with Nagios, Ganglia is highly configurable and lets you add custom metrics to the Ganglia graphs. Its ability to visualize resource usage trends helps with your capacity planning exercises. Nagios and Ganglia work very well together. If Nagios emits alerts about a host or application, Ganglia helps you immediately find out if CPU or resource utilization, and system load have anything to do with the alerts sent by Nagios.

Service Metrics

System administrators regularly track system metrics such as OS and web server performance statistics. However, service metrics are very important too. Among other

things, service metrics reveal how your customers are using your services, and which areas of a service can benefit from enhancements. Apache Hadoop has numerous built-in counters that help you understand how efficient MapReduce code is. Similarly, Codehale's Metric x Library provides counters, timers or gauges to for JVM. It also lets you send the metrics to Graphite or another aggregating and reporting system.

Synthetic Monitoring

While system metrics have their uses, it's hard to figure out how a service or an application will react when it's confronted by an unusual set of circumstances. Organizations sometimes insert fake events into their job queues, called synthetic transactions, to see how the system behaves. Synthetic monitoring often yields more meaningful metrics than low-level traditional metrics. Often, synthetic monitoring is used to measure availability of the applications and real-user monitoring is used to monitor performance. Chapter 16 delves into application performance monitoring and Real User Monitoring (RUM) and synthetic application monitoring.

Cloud Computing

In the past few years, especially over the past 5 years, there has been an explosive increase in the outsourcing of hardware and platform services, through the adoption of cloud computing. The reasons for the headlong rush of companies to cloud based computing aren't hard to figure out. When you move to the cloud, you don't need to spend as much to get a new infrastructure in place and hire and train the teams to support it. Applications such as ERM (Enterprise Risk Management) that are notoriously difficult to implement successfully, can be had at a moment's notice by subscribing to a cloud based application. Enhanced performance, scalability and reliability and speed of implementation are some of the main drivers of the move to cloud-based computing.

Platform-as-a-Service means that vendors can support the entire stack required for running mission critical applications, such as the web servers, databases, load balancers etc. You can monitor and manage the cloud based infrastructure through web based interfaces

Cloud based providers such as Amazon's AWS and the Rackspace Cloud have wowed both system administrators and developers with their ability to provide huge amounts of on-demand computing power at the mere push of a button! Using tools such as knife, you can spin up and spin down server capacity as your demand fluctuates. You can use the configuration tools not only with the public cloud providers but also with private cloud platforms such as VMware's vSphere.

However, for many organizations, especially small to middle sized ones, it makes a lot of sense to simply outsource the entire task of running the infrastructure and building a new data center to a service such as Amazon Web Services (AWS). AWS (as well as Microsoft's Azure and Rackspace Cloud) is a highly optimized and efficient cloud infrastructure and if the cost analysis works for you, it may be the best way to move to a cloud-based environment. After all, if you want to go the route of something such as AWS, all you need is a valid credit card and you could be setting up a cloud environment and running a web application or a big data query in a matter of a few hours!

Let me make sure and point out that cloud computing isn't an unmixed blessing for everybody. Cloud computing comes with its own shortcomings, chief of which are concerns such as the inability of service providers to offer true SLAs (service level agreements), data integration between private and public cloud based data, the inability to audit the provider's systems or applications, the lack of customization, and inherent security issues due to the multi tenancy model of public clouds.

Amazon Web Services (AWS) is a truly amazing set of services that enable you to get on the cloud with minimal effort and pain. AWS includes several exciting components such as auto scaling and elastic load balancing. I discuss AWS in detail in Chapter 10.

Even when you aren't using a cloud computing vendor such as AWS, Microsoft Azure or Google App engine, you're likely to be virtualizing your on premise infrastructure through private clouds using something like OpenStack. OpenStack is hot right now, so let me introduce you to it in the following section.

Open Stack – the Open Source Cloud Platform

OpenStack is a popular open source platform that enables you to build an IaaS cloud (Infrastructure as a Service). OpenStack is designed to meet the needs of both private and public cloud providers and can support small to very large sized environments. OpenStack is free Linux based software that provides an orchestration layer for building a cloud data center. It provides a provisioning portal that allows you to provision and manage servers, storage and networks.

The goal of OpenStack is to provide an open standard, easy to use, highly reliable cloud computing platform for organizations so they can build either a public or a private cloud.

Software Defined Networking

Traditional network administration involves administrators manually configuring and maintaining complex physical network hardware and connectivity. Software defined networking (SDN) is a fast evolving area in networking. SDN aims to apply

the virtualization concepts currently used at the storage and processing areas to networking. By combining virtualization and distributed architectures, SDN aims to remove the need to manage physical network devices on an individual basis. SDN allows administrators to manage network services in an abstract fashion and facilitates the automating of network tasks.

SDN is becoming increasingly popular, and modern cloud computing platforms such as Open Cloud rely on it. Cloud computing regards SDN as an essential and key foundation. OpenFlow and Frenetic/Pyretic are some of the new tools associated with SDN. Chapter 2 discusses SDN and Network Flow Virtualization (NFV) in detail.

Microservices, Service Registration and Service Discovery

Microservices are a fast growing architecture for distributed applications. Microservice architectures involve the breaking up of an application into tiny service apps, which each of the apps performing a specialized task. You build up applications by using microservices as your building blocks.

You can deploy microservices independent of each other since the services are loosely coupled. Each microservice performs a single task that represents a small part of the business capability. Since you can scale the smaller blocks per your requirements, fast-growing sites find it extremely convenient to adopt a microservice based architecture. Microservices help deliver software faster and also adapt to changes using newer technologies.

Right now, many organizations use Node.js, a cross-platform runtime environment for developing server-side web applications, to create the tiny web services that comprise the heart of a microservice based architecture. Following are some of the common tools and frameworks used as part of creating and managing microservices.

- Node.js
- Zookeeper (or etcd or consul)
- Doozer
- Serf
- Skydns/skydock

Some folks use the two-pizza rule to as a guideline for determining if a service really qualifies as a microservice. The Two-Pizza rule states that if you can't feed the team that's building the microservice with two pizzas, the microservice is too big!

Microservices are just what they sound like – they're small services that are geared towards a narrow functionality. Instead of writing one long monolithic piece of code, in a microservice based approach, you try to create independent services that work together to achieve specific objectives.

If you're finding that it's taking a very long time and a large amount of effort to maintain code for an app, the app may very well be ready for breaking up into a set of smaller services. As to how small a microservice ought to be, there's no hard and fast rule. Some define a microservice in terms of the time it takes to write the application – so, for example, one may say that that any service that you can write within two weeks qualifies as a microservice.

A PaaS such as Cloud Foundry is ideal for supporting microservice architectures, since it includes various types of databases, message brokers, data caches and other services that you need to manage.

The key thing is that the services are independent of each other and are completely autonomous, and therefore, changes in the individual services won't affect the entire application. Each of the services collaborates with other services through an application programming interface (API). Microservices in general have the following features:

- Loosely coupled architecture: microservices are deployable on their own without a need to coordinate the deployment of a service with other microservices
- Bounded Context: any one microservice is unaware of the underlying implementation of the other microservices
- Language Neutrality: all microservices don't have to be written in the same programming language. You write each microservice in the language that's best for it. Language neutral APIs such as REST are used for communications among the microservices
- Scalability: you can scale up an application that's bottlenecked without having to scale the entire application

One may be wondering as to the difference between microservices and a service-oriented architecture (SOA). Both SOA and microservices are service architectures that deal with distributed set of services communicating over the network. The focus of SOA is mostly on reusability and discovery, whereas the microservices focus is on replacing a monolithic application with an agile, and more effective incremental functionally approach. I think microservices are really the next stage up for the principles behind SOA. SOA hasn't always worked well in practice due to various reasons.

Microservices are a more practical and realistic approach to achieving the same goals as SOA does.

Benefits of Microservices

Following is a brief list of the key benefits offered by microservice based architectures.

- Heterogeneous Technologies: you don't have to settle for a common and mediocre technology for an entire system. Microservices let you choose the best of breed approach to technologies for various functionalities provided by an application.
- Speed: it's much faster to rewrite or modify a tiny part of the application rather than make sure that the entire application is modified, tested and approved for production deployment.
- Resilient systems: you can easily isolate the impact of the failure of small service components.
- Scaling: you can scale only some services that actually need scaling and leave the other parts of the system alone.
- Easy Application Modifications and Upgrades: with microservices, you can easily replace and remove older services since rewriting new ones is almost trivial.

Both service registration and service discovery play a key role in the management of microservices, so let me briefly explain these two concepts in the following sections.

Service Discovery

Communication among the microservice applications is a major issue when you use the microservice architecture. Server Registration and Service Discovery are the solutions for this issue. As the number of microservices proliferates, both you and the users need to know where to find the services. Service discovery is the way to keep track of microservices so you can monitor them and know where to find them.

There are multiple approaches to discovering services, the simplest strategy being the use of the Domain Name Service (DNS) to identify services. You can simply associate a service with the IP address of the host that runs the service. Of course this means that you'd need to update the DNS entries when you deploy services. You could also use a different DNS server for different environments. However, in a highly dynamic microservice environment, hosts keep changing often, so you'll be stuck with having to frequently update the DNS entries.

The difficulties in handling dynamic service discovery through DNS have led to other approaches, which involve the concept of service registration.

Service Registration

Service registration is the identification and tracking of services by having services register themselves with a central registry that offers a look up service to help you find the services. There are several options for implementing service registration, as I explain in the following sections.

Zookeeper

Zookeeper is a well-known coordination service for distributed environments, and happens to be a critical component of highly available Hadoop driven big data systems. Zookeeper offers a hierarchical namespace where you can store your service information. You could set up monitors and watches so you and the clients are alerted when there are any service related changes.

Etcd

Etcd is a distributed key-value store that provides shared configuration and service discovery for clusters. Etcd is configured to run on each node in a cluster and handles the master election during the loss of the current master. In a CoreOS cluster, for example, application containers running in a cluster read and write data into etcd. The data include things such as the database connection details and cache settings.

Consul

Consul is more sophisticated than Zookeeper when it comes to service discovery capabilities. It also exposes an HTTP interface for discovering and registering services and also provides a DNS server. Consul also has excellent configuration management capabilities.

Chapter 5 discusses service discovery and service registration in more detail, in the context of using Docker containers.

CHAPTER 2

Networking Essentials for a System Administrator

Networking is complex – and my goal is to simplify it a bit so you can be an effective system administrator. Networking has always been a key component of a Linux system administrator's skill set. However, now more than ever, it has become really one of the, if not the most important skill sets you need to master. The reason is that today, with heavy internet driven workloads and the proliferation of cloud based systems, networking has become the fulcrum of everything. When dealing with a cloud environment such as Amazon's AWS, you'll find that most of the complex work involves the setting up of the cloud DNS, internet gateways, virtual private networks and so on.

As a system administrator, one of your key tasks is to ensure that your system is able to accept and send data, both from within your own corporate network as well as from anywhere on the internet. It's therefore essential that you understand the key components of a network and grasp how data is moved through the network. You must also understand the important protocols that are in use today to move data across networks.

The TCP/IP (Transmission Control Protocol/Internet Protocol) protocols are the two key protocols that stand at the center of modern networking. This chapter builds on basics such as packets and frames and leads up to a thorough explanation of the TCP/IP protocol. In addition, I also talk about several key networking concepts such as the domain naming service (DNS), dynamic host protocol (DHCP), and similar useful networking concepts.

What the internet is

Before we delve into the intricacies of modern networking, it probably is a good idea to discuss the Internet and find out how it works. This gives us the opportunity to define various key concepts of a network, such as the following:

- Packets
- Headers
- Routers
- Protocols

The internet is the largest network in the world, and connects billions of computing devices across the world. These devices are called hosts or end systems. The systems are called hosts since they host applications such as web server or an E-mail server, and they're referred to as end system since they sit at the edge of the Internet.

End systems access the Internet via Internet Service Providers (ISPs). These end systems are connected together via a network of communication links and packets switches. When a host needs to send data to another host, it segments the long messages (such as email messages, a JPEG image or a MP3 audio file) into smaller chunks of data called packets, and adds header bytes to the segments. The resulting packages of data (packets) are then sent through the network to the destination host, where they are reassembled so as to get the original data.

Packets

Network packets are the smallest chunk of data transmitted by a network – each of the packets contains the data we are transmitting across the network as well as some control data that helps the networking software and protocols determine where the packet should go.

A frame is synonymous with a packet in common usage, but it's actually different from a packet – a frame is the space in which the packet goes. Each frame has a preamble and post-amble that tells hardware where the frame begins and ends. The packet is the data contained in a frame.

Each type of network protocol uses a header to describe the data it's transmitting from one host to another. Packets are much small in size compared to the actual data – for example in a packet that's 1500 bytes in size, the headers for TCP, IP and Ethernet together take up only 54 bytes, leaving 1446 bytes for the payload.

When a network card on a host gets a packet, it verifies that it's indeed supposed to accept it by looking up the destination address in the packet header. Next, it generates an interrupt to the operating system. The operating system will then request the

device driver or the NIC to process the packet. The device driver examines the packet to find the protocol it's using, and places the packet where the protocol's software or the stack can find it. Most likely, the packet will be placed in a queue and the stack grabs the packets from the queue in the order they arrived.

As the packet passes through the network layers, the driver strips appropriate headers at each of the layers. For example, IP strips the IP headers and TCP strips the TCP headers. Finally, the packet will be left with just the data (payload) that it needs to deliver to an application.

Packet Switches

A packet switch grabs the packets arriving on its incoming communication links and forwards the packets to its outgoing communication links. The two most commonly used packet switches are routers and link-layer switches. i Routers are more common in the network core and the link–layer switches are usually employed in access networks. An access network is one that physically connects an end system to the very first router (also called an edge router) on a path to other end system. Access networks could be a home network, a mobile network, an enterprise network, or a regional, national, or global ISP. Thus, a company's' devices are linked by an Ethernet switch which then uses the company's routers to access the company's ISP and through it, to the Internet.

The path travelled by a packet through all the communication links and the packet switches between the source and destination end systems is known a route through the networks.

Applications and APIs

While you can look at the Internet as a set of hardware and software components that use a set of protocols, routers and switches to move around vast amount of packets across the world, what we're really interested in is how the Internet helps applications we all use, Thus, it's probably a good idea to view the Internet as a linked infrastructure that provides various services needed by applications that run on the end systems. Here, by applications, I'm referring to things such as E-Mail, instant messaging, Web surfing, social networks, video streaming, Voice-over-IP (VoIP), distributed gaming, and so on.

Network Protocols

Network protocols are at the heart of the Internet, so let's delve into network protocols a bit. Network protocols are similar in a way to the normal protocols we all follow in a civilized world. Protocols in the networking context are sets of rules and procedures that allow different devices and systems to communicate with each other. In the OSI Reference Model, a protocol is a set of rules that governs communication

among entities at the same layer. TCP/IP is often referred to a protocol – it's actually a protocol suite, since it's really a set of protocols.

A network protocol helps hardware or software components of a device such as a computer exchange message and take actions. More formally, we can say the following: a protocol defines the format and order of messages exchanged between two systems, as well as the actions taken during the transmission and the receipt (or non-receipt) of the messages.

A hardware-implemented protocol for example, controls the flow of the actual bits on the links ("writes') between two network interface cards. Similarly, a software-implemented protocol in a router specifies a packet's route from the source to the destination system. When a router forwards a packet that comes into its communication link, it forwards that packet to one of its outgoing communication links. How does the router know to which link it should forward the packet? The exact method depends on the type of computer network you're dealing with.

In the Internet, every end system has an IP address that identifies it. The end system sending a packet to a destination end system includes the destination's IP address in the packet's header portion. Each router has a forwarding table that contains mappings of destination addresses to that router's outbound communication links. When packets arrive, the router examines their addresses and searches tis forwarding table, using the packet's destination addresses to find the correct outbound communication link. The router then forwards the packet to this link.

The Internet uses a set of special routing protocols that are used to automatically set the forwarding tables. The routing protocols use algorithms such as those that employ the shortest path between the router and a destination – they then compute the shortest path and configure the forward tables in that router.

You can use the traceroute program (www.traceroute.org) to trace a route for accessing any host on the internet.

Network Messages and Message Formatting

So, networks represent a set of connected servers that exchange information. How does this information actually flow through the network? Networks send and receive data in the form of structures called messages. When information needs to be sent through a network, it's sent by breaking the data into these small chunks (messages).

There are several types of network messages, some of which are the following:

- Packets: Messages sent by protocols operating at the network layer of the OSI model are called packets, or IP packets, to be more accurate.
- Datagrams: this is more or less synonymous with a packet and refers to the network layer as well. Often this term is used for messages sent at higher OSI levels.
- Frames: these are messages that are sent at the lower levels of the OSI model, most commonly the data layer. Frames take higher-level packets or datagrams and add the header information needed at the lower levels.

Each of the networking protocols uses messages and each protocol uses a different formatting method to specify the structure of the message it carries. For example a message connecting a web server to a browser will be different from a message that connects Ethernet cards.

While the formatting of the message depends on the specific protocol sending the messages, all messages contain headers and a footer, along with a data element which represents the actual data being transmitted in a message. The OSI model refers to the message handled by a protocol as its PDU (Protocol Data Unit) and the data part as its SDU (Service Data Unit). A lower layer protocol's SDU then is the PDU of a higher-layer protocol.

With this quick introduction to the Internet, let's move on to learn the important elements of modern day networking.

Networking Essentials – Theory and Practice

A network is a group of connected servers that use specialized hardware and software to exchange information. The size of a network can range from really tiny affairs such as a home network, to massive, complex infrastructures, such as the networks managed by Google, for example. The biggest network of course, is something you're all familiar with – the internet itself.

Networks can be classified into classes such as Local Area Networks (LANs), Wide Area Networks (WANs), and Metropolitan Area Networks (MANs). A subnetwork or subnet is a portion of a network - a network that's part of a large network. A segment is a small section of a network and usually smaller than a subnetwork. An internetwork (or Internet) is a large network formed by connecting many smaller networks.

Breaking up the real work into layers – the OSI Model

Networking involves several technologies, which interact in a complex and sophisticated manner to perform their job, which is to move data along the network. To make sense of networking, it's essential that you start at a conceptual level, by going to the fundamental networking model, known as the OSI Reference Model (OSI).

The OSI model provides a basic framework that explains how data moves through a network, by organizing the components of the network into several interlinked networking layers. The OSI model is a great learning tool for understanding networks – and you'll frequently find references to the various layers of the OSI when dealing with various aspects of networking.

Many of you may have come across the OSI model as part of a Computer Science college course, or during a network training class, and therefore think that it's something theoretical, with little or no practical use. Networking models such as the OSI seem, let's face it, boring! What possible benefit could there be to learning about a theoretical model such as this? After all, the OSI model was conceived a very long time ago, when a LAN (local area network) was what a network mostly meant. The model doesn't adapt very well to even WAN technologies, let alone the internet.

The network layers at various network layers together are called the protocol stack.

Protocol Layering

Networks and network protocols, including the hardware and software that supports the protocols, are organized into multiple layers so as to provide an efficient structure to the network design. Each of the many network protocols belongs to a specific network layer. Each layer offers a specific set of services to the layer above it and consumes services provided by the layer beneath it, and therefore we call this a service model of layering. As an example, a layer named n will consume a service such as an unreliable message delivery provided by a layer below it, named n-1. In its turn, layer n will provide the service of a reliable message delivery, made possible by adding its functionality that detects and retransmits lost messages.

As mentioned earlier, protocol layering makes it simpler to update network components in a specific layer. However, they do carry the drawback that several of the network layers duplicate the functionality of a lower layer.

A networking model such as OSI is a framework that shows the multitude of tasks involved in making a network function, by splitting the work into a series of connected layers. Each of the layers has hardware and software running on it, whose job it is to interact with the hardware and software on other layers.

The OSI model, as mentioned earlier, consists of seven distinct layers, with the lowest layer being where the low level software and hardware are implemented. The layers divide up the work involved in implementing a network. The highest layer, Layer 7, is the application layer and this layer deals with user applications and operating system

software. As you proceed from the bottom most layer (Layer 7) to the upper most layer (Layer 1, application), you move up along increasing levels of abstraction. The lower layers deal with both hardware and software, and by the time you get to the top layers, they're dealing exclusively with software. The lower levels do all the work so the upper level layers, especially Layer 7, can serve application users.

It's easier to understand the layers of the model by grouping the seven layers into just two simple parts:

- Lower layers (Layers 1,2, 3, and 4): these are the physical, data link, network and transport layers and are responsible for formatting, packaging, addressing, routing, and the transmitting data over the network. These layers aren't concerned with the meaning of the data – they just move it.

There's no hard and fast rule of how you classify the seven OCI layers into higher and lower layers. For example, I chose to place the transport layer into the set of lower layers, since it primarily provides services to the higher layers by moving data. However, this layer is often implemented as software and so some folks may want to group it with the higher layer.

- Upper Layers (Layers 5,6, and 7): these are the session, presentation and application layers – they concern themselves with software application related issues. The upper layers interact with the users and implement the applications that run over the network. The upper layers aren't concerned with the low level details of how data moves over the network, instead depending on the lower layers to do that for them.

The TCP/IP Protocol Suite

TCP/IP (transaction communications protocol/internet protocol) is a set of networking communication protocols, and is the king of modern networking. It's also the underlying language of the internet, and it's thus important that you understand this protocol suite in some detail. In the TCP/IP protocol suite, the TCP protocol provides network layer functionality and the IP protocol the transport layer functionality. TCP/IP is ubiquitous – if you're using the HTTP protocol, which all of us do when we use the internet, you're using TCP/IP, since the HTTP protocol is part of the TCP/IP suite of protocols. Among the many reasons for its widespread use is the fact that the TCP/IP protocols are highly scalable, and are universally accepted and used across the internet.

There are two IP protocols – IPV4 and IPv6. IPV6 was created because IPV4 was going to run out of IP addresses. Although IPV4 did run out of IP addresses, it's still going to be around. IPV6 isn't really used for most end customers of Internet Service Providers (ISPs).

When you send out a packet it first goes to your default gateway. Routers keep forwarding the packet until it reaching its destination. IP provides simplicity and efficiency to message transmission but it doesn't offer delivery guarantees. Here's where TCP comes in – while IP delivers and routes packets efficiently, TCP guarantees the delivery and the order of packet transmission.

TCP protocols are stateful, meaning that they manage the state of the connection. UDP is a stateless protocol and this doesn't involve any handshaking. UDP is a faster protocol as a result, since there's no mechanism in the form of state to control how data is transmitted, track the data transmission, and retransmit the data if there are any errors during the initial transmission. TCP effortlessly handles all of these aspects of message transmission, as a result of setting up a stateful session following the initial handshake between clients and servers.

In Chapter 3, you'll learn about the wonderful new HTTP/2 protocol that'll eventually replace the current HTTP/1.1 protocol – one of its best features is that it uses a single TCP connection to perform multiple transactions.

The Internet Protocol Stack

The Internet Protocol stack, also called the TCP/IP network protocol stack, uses just four network layers to represent the work done by the top 6 layers of the OCI Model. The seventh layer in the OCI model, the Physical Layer, isn't part of TCP/IP, since the interface between the TCP/IP protocol stack and the underlying network hardware occurs at the Data Link Layer (Layer 2 in the OCI model).

Following is a simple description of the four TCP/IP layers and how they correspond to the OCI layers.

- Network Interface Layer: this is the lowest layer in the TCP/IP model and where the TCP/IP protocols from its upper layer interface with the local network. On most TCP/IP networks there's no Network Interface Layer since it's not necessary when you run TCP/IP over Ethernet.
- Internet (also called network) Layer: corresponds to the network layer in the OCI model and is responsible for data packaging, delivery, and routing. t's in this layer that the well-known IP protocol, the guts of TCP/IP, works. Routing protocols such as RIP (Routing Information Protocol) and BGP (Border Gateway Protocol)

that determine the routes to be followed by datagrams between source and destination end systems operate in the network layer as well.

The internet and the transport layer are the guts of the TCP/IP network layer based architecture – they contain almost all the key protocols that implement TCP/IP networking.

- Host-to-Host Transport Layer: Usually called just the Transport Layer, this layer provides the end-to-end communications over the internet. It allows logical connections to be made between network devices and ensures that transport-layer packets (called segments) are sent and received, and resent when necessary. Key transport protocols are TCP and UDP. NOTE: Real time applications such as video conferences and Internet based phone services are slowed down by TCP's congestion control mechanism and hence often run their applications over UDP rather than TCP.
- Application Layer: The application layer is where the application-layer protocols reside. This layer encompasses layers 5-7 of the OCI model and includes well known protocols such as HTTP, FTP, SMTP, DHCP and DNS. Over the past several years, the rise of the internet and the widespread use of the TCP/IP protocol has led to an obscuring of the original seven layers of the OSI model, although the latter is still relevant. TCP/IP protocols have basically supplanted the OCI model.

The Network Layer and TCP/IP Networking

Of all the TCP/IP (or Internet) network layers, the network layer is probably the most complex, so let's spend some time learning how the network layer functions and the services that this layer provides.

The job of the network layer is seemingly simple: it moves packets from the source host to the destination host. The two key functions provided by the network layer for our purposes are forwarding and routing. While forwarding deals with the transferring of a packet from an incoming communication link to an outgoing communications link within the same router, routing involves all routers within a network.

It's the interactions among the routers through various routing protocols that determine the end–to-end path that a network packet follows as it vends its way from a source host to the destination host.

The Internet's network layer is a minimalist network-layer service layer: it doesn't provide a guaranteed delivery service which is one of the many service types a network layer can provide, instead, it provides a best-effort service, where the packets aren't guaranteed to be received in the same order in which they're sent, and the timing between packets isn't guaranteed to be preserved – even the delivery of the transmitted packets isn't guaranteed!

Routing and forwarding are often used as synonyms, but there are crucial differences between these two network layer functions, as explained in this section.

The Internet's network layer consists of three major protocols (or sets of protocols):

- The IP Protocol: You've learned about the IP protocol earlier in this chapter. The IP protocol determines the datagram (a network-layer packet) format, addressing and packet handling conventions.
- Routing protocols: these determine the best path for network packets to traverse. Routing protocols determine the values in a router's forwarding table.
- The Internet Control Message Protocol (ICMP): this is the network layer's error reporting and router signaling protocol. This protocol reports datagram errors and responds to requests for some types of network related information.

The Forwarding Function

When a network packet arrives at a router, its input communication link receives the packet and the router must then move the packet to the correct output link – this is called forwarding the packet. Note that the terms forwarding and switching are often used as synonyms. All routers use forwarding tables.

A router forwards a network packet by using the header value (such as the destination address of the packet) of an arriving packet to look up the corresponding values stored in the forwarding table. The corresponding values in the forwarding table for a specific header show the router's outgoing link interface to which the router must forward the packet.

Routing algorithms determine the values that should be added to the forwarding tables. Routers use the routing protocol messages that they receive to configure their forwarding tables.

A router in this case is a type of a packet switch. A packet switch transfers packets from input link interfaces to output link interfaces. Since routers are network layer (layer 3) devices, they base their forwarding decisions on the values in the fields per-

taining to the network layer. Routers are store-and-forward packet switches that forward network packets using the network layer addresses. Switches, on the other hand, while they're also store-and-forward packet switches, are very different from routers since they forward packets using MAC addresses.

A router has four main components: input ports, output ports, a switching fabric that connects the input/output ports and actually switches (forwards) packets from input ports to output ports and finally, a routing processor. The input/output ports and the switching fabric are usually implemented in hardware. It's these three compoents that perform the forwarding function, sometimes together referred to as the forwarding data plane. This layer needs to perform very fast, and the speed (in nanoseconds) at which it should function is too fast for a software implementation.

Forwarding and Addressing Using the IP Protocol

Forwarding and addressing in the Internet are determined by the IP protocol in the network layer. Today, there are two IP protocols in use, the most common being IPV4 (IP protocol version 4). IPV6 (IP protocol version 6), a solution introduced to overcome the limited number of IP addresses made available by the IPV4 protocol, will be the IP protocol to which everybody will be moving to over time. However, this isn't going to happen anytime soon, and IPV4 is going to stick around for a while.

An IPV4 address is simple, and is also called a dotted quad number, such as 192.68.1.10, for example. It's called dotted quad because it's made up of four decimal numbers ranging from 0-255, with each number separated by a period.

In what follows, I explain IPv4 and the Internet's address assignment strategy, called Classless Interdoman Routing (CIDR).

CIDR – The IPv4 Addressing Strategy

This section deals with the details of IPv4 network addressing. Typically hosts have a single link into the network and IP uses this link to send its datagrams. The boundary between the physical network link and the host is called an interface. The boundary between a router and its input/output links is also called an interface. Since IP requires each host and router interface to have its own IP address, an IP address then is really associated with an interface than with the actual host server or the router.

An IP address is four bytes (32 bits) long, and has a hierarchical structure. There are a total of about 4 billion possible IP addresses. An IP address looks like the following: 121.6.104.88, where each period separates the four bytes, which are expressed in the dotted- decimal notation from 0-255. Each byte in the address is written in the decimal form and is separated from the other bytes in the IP address by a period (dot). Let's say you've an IP address such as 192.36.224.12. In this address, 192 is the decimal equivalent of the first 8 bits of the address, and so on. An IP address is hierarchical since as you go from the left to the right portions of the address, you get more and

more specific information about the hosts located in the Internet. That is, you'll get a more specific sense of the network where the host with this IP address lives.

Hosts and Networks

Each component of a network is assigned a network address. Usually the network address is 32 bits long. The network address is always split between the network component and the hosts that belong to that network. In the Internet, every host and every router must have a unique IP address. The first several bits of the network address of a host or interface are taken up by the network component (actually, the subnet, as you'll see shortly) and the rest are used to enumerate the hosts in a network.

In the Internet lingo, a subnet is also called an IP network, or just a network.

As I mentioned earlier, the first several bits in the network address are assigned to the network. An octet is 8 bits which means the number before the dot in a dotted decimal IP address notation, as in 192.168.1.42. In this IP address, the first octet is 192 and there are three other octets following this.

Networks are organized by size into various classes. The first octet of the IP address determines the class to which the network address belongs to. Here are the four network classes and the octet ranges they're assigned in a network address:

```
Class         Octet Range
A               0-126
B               128-192.167
C               192.169-223
```

Following is what you need to remember regarding the address ranges:

- The 127.0.01 address is special: it's a loopback address that every host (that uses IP) uses to refer to itself.
- The following are all private IP address blocks that you can't allocate to anyone on the Internet, and are meant to be used for internal networks only.
- All IP addresses in the 10.0.0 network
- The networks ranging from 172.16 to 172.31
- The network 192.168

Subnetting and Netmasks

For performance and management reasons, it's common to create subnetworks under an organization's main network. Subnetting refers to the creation of the subnetwork. If your corporate network is 10.0.0.0, you can create smaller class C networks under it such as 10.1.1.0, 10.1.2.0 and so on.

Each of the subnets will have a 24-bit network component and an 8-bit host component in its 32 bit network address. The first 8 bits are designated for the organization's main network and the last 8 bits are for designating the hosts in the network (the host component of an IP address is set to all zeroes. This makes it easy for you to tell which part of the network address corresponds to the network itself and which addresses correspond to hosts). In this example, the 16 bits allocated to the subnet allow you to create potentially 65,534 sub networks (calculated by raising 2 to the power of 8).

To really grasp network addressing, you must understand binary numbering, which I explain in the following section.

Binary Numbers

In normal counting, you use decimal numbers, where each digit ranges from 0-9. In a number such as 1234, for example, the ones place has 4 as its value. The next number, 3, is in the tens place, the number 2 is a higher power of 10 (100) and the 1 is a yet higher power of 100 (1000). In a binary number, on the other hand, the values can be either 0 or 1. Thus, 1111 is a binary number, which has 1's in all places. You start at the right with a value of 1, and each digit to its left side is a higher power of two. Since binary numbers have just two possible values for each digit, (0 or1), they're also referred as base 2 numbers.

Here's how binary numbers correspond to decimal numbers – make sure you calculate the value of the binary numbers by computing and adding the values of the digits from right to left.

```
** Decimal:   0       1        2       3       4        5       6       7       8        9
** Binary:    0       1       10      11     100      101     110     111    1000 1001     1010
```

To calculate the value of a binary number, you add the values for each binary digit place where you see a 1 – a 0 means there's no corresponding binary value. The binary number 111 evaluates to 7 since it has a value for the 4, 2 and 1 places. Similarly the binary number 1101 evaluates to the decimal number 13 (8+4+0+1).

The Netmask

The netmask serves to let the IP stack know which portion of a network address belongs to the network and which part to the hosts. Depending on this, the stack

learns whether a destination address is local (on the LAN), or it needs to be sent to a router for forwarding outside the LAN.

According to the definition of a netmask, the bits in the network address that are zero belong to the host. Let's say you're looking at the network address 192.168.1.42 with the netmask of 255.255.255.0. For easier understanding, let's show the binary representation of the IP addresses, which are in their dotted decimal form:

```
Decimal                                                  Binary
192.168.1.42                    11000000    00101000    10000001    00101010
255.255.255.0                   11111111    11111111    11111111    00000000
```

Given the IP address (192.168.1.42) and its netmask (255.255.255.0), how do you tell which part of the IP address is the network and which part, the host? Remember that the bits that are zero in the netmask belong to the host. Thus, the three octets (255.255.255) are allocated to the network and the last octet (0) represents the host.

Given an IP address and its netmask, you use the bitwise AND operation to figure out the network address. Bitwise AND operations are simple: if both bits in the corresponding place are 1, the result of the AND is a 1. If either bit or both bits evaluate to 0, the result of the AND operations is a zero. If you then perform the bitwise AND operations on the IP address 192.168.1.42 and its netmask of 255.255.255.0 using their binary representation, here's what you get:

```
11000000    10101000    00000001    00101010
11111111    11111111    11111111    00000000
-----------------------------------------------
11000000    10101000    00000001    00000000
```

The resulting bit pattern after performing the bitwise AND operation (11000000 10101000 00000001 00000000) is the same as 192.168.1.0 in the dotted decimal format. It's painful to write out the complete netmask each time you want to refer to it. Luckily, there's a shortcut – you simply write out the network address followed by a slash (/) and the number of bits in the netmask. Thus, the network address 192.168.`.`0 with a netmask of 255.255.255.0 will translate to 192.168.1.0/24. This is so since there are 24 bits on the network address portion of the netmask (remember that all bits that are non-zeros belong to the network and the zeros go to the host).

Let's say you're dealing with the Internet and the IP protocol. In this case, if you've three host interfaces and a router interface connecting to them, the four interfaces form a subnet, often simply called a network in the Internet terminology. The address assigned to this subnet, say, is 224.1.1.0/24. You can have additional subnets such as 224.1.2.0/24 and 224.1.3.0/24 as well.

The /24 notation in the addressing scheme is called a subnet mask, which means that the leftmost 24 bits (out of the total 0f 32 bits) of the network address define the subnet address. In this case, the subnet 224.1.1.0/24 consists of the three host interfaces with the addresses 224.1.1.1, 224.1.1.2, 224.1.1.3 and a router interface with the

address 224.1.1.4. Additional hosts you attach to the subnet 224.1.1.0/24 would all be required to have addresses of the form 224.1.1.xxx.

The Internet address assignment strategy is called Classless Interdomain Routing (CIDR), and is a way to generalize subnet addressing that I described earlier. Under CIDR, a 32-bit IP address has the dotted-decimal form a.b.c.d/x and is divided into two parts, with the x showing the number of bits ion the network portion (first part) of the IP address. The x number of leading bits are also called the network prefix (or just prefix) of the OP address.

Routers outside your company's network will consider only the x leading prefix bits (remember that there are a total of 32 bits in an IP address). So, when a router in the Internet forwards a datagram to a host inside your network, it's concerned only with the leading x bits of the IP address, which means that the routers need to maintain a much shorter forwarding table than otherwise. For your company, the router has to maintain a single entry of the form a.b.c.d/x – that's it! Let's say you're dealing with the subnet 224.1.1.0/24. Since x is 24 here, there are 8 bits left in the IP address. These 8 bits are considered when forwarding packets to routers inside your company.

How You Get Your IP Addresses

When your organization asks for IP addresses, it's assigned a block of contiguous IP addresses, which means a set of address with a common prefix and all your company's IP addresses will thus share a common prefix. All addresses are managed by the Internet Corporation for Assigned Names and Numbers (ICANN).

The non-profit ICANN not only allocates the IP addresses but also manages the DNS root servers and assigns domain names. To get a block of IP addresses for one of your organization's subnets, your network admin needs to contact your Internet Service Provider (ISP). Let's say the ISP was assigned the address block 200.23.16.0/20. The ISP will break this block into chunks and assign each chunk of addresses to a different organization. Let's say our ISP wants to break its address block into 8 blocks for assigning to various organizations. This is how the breakup would look like:

```
ISP's  block          200.23.16.0/20         11001000  00010111  00010000  00000000
Company 1             200.23.16.0/23         11001000  00010111  00010000  00000000
Company 2             200.23.18.0/23         11001000  00010111  00010000  00000000
…
Company 8             200.23.30.0/20         11001000  00010111  00010000  00000000
```

Note that I've underlined the subnet portion of each organization's subnet.

Configuring IP Addresses with the Dynamic Host Configuration Protocol (DHCP)

Once your organization is assigned a block of IP addresses, it must assign IP addresses out of that pool of addresses to all hosts and router interfaces in your network. If you've just a handful of servers for which you need to configure IP addresses,

it's easy to do so manually. The sys admin manually configures the IP addresses with a network management tool. However, if you need to configure IP addresses for a large number of servers, you're better off using the Dynamic Host Configuration Protocol (DHCP), which allows hosts to be automatically assigned an IP address).

DNS and DHCP are related in many ways. DHCP is a way to provide a network device with an IP address, and whereas DNS maps IP addresses to hostnames. You can integrate DHCP and DNS, whereby when DHCP hands a network device an IP address, DNS automatically updated to reflect the device name.

There's a client and a server DHCP that you need to configure for using DHCP to handle IP address assignment. You can configure DHCP so it assigns either a temporary IP address which is different each time a host connects to the network, or have it assign the same IP address each time. When you start the DHCP client, it requests the network for an IP address and the DHCP server responds to that request by issuing an IP address to help complete the client's network configuration. NOTE: Since DHCP automates the network related aspects of connecting the hosts, it's called a plug-and-play protocol.

The response that a DHCP server issues consists of the IP address that it assigns to a host, and may also include the address of the local DNS server, the address of the first-hop router (known as the default gateway), and a network mask. Client use this information to configure their local settings. The IP addresses granted by the DHCP server have a lease attached to them. Unless the clients renew the lease on the address, the address goes back into the server's address pool to satisfy requests from other clients.

Internet IP addresses are network-layer addresses. Link layer addresses, are also called physical addresses, or MAC addresses. The Address Resolution Protocol (ARP) translates between these two types of addresses, IP addresses are in the dotted decimal notation, and MAC addresses are shown in the hexadecimal notation

Network Address Translation (NAT)

Network Address Translation (NAT) is a way to map your entire network to a single IP address. You use NAT when the number of IP addresses assigned to you is fewer than the total number of hosts that you want to provide Internet access for. Let's say you've exhausted the entire block Of IP addresses assigned to you by your ISP. You can't get a larger block of IP addresses allocated, because your ISP has no contiguous portions of your current address range available. NAT comes to your rescue in situations like this.

Using NAT means that you take advantage of the reserved IP address blocks and set up your internal network to use one or more of these network blocks. The reserved network blocks are:

```
10.0.0.0/8              (10.0.0.0 - 10.255.255.255)
172.16.0.0/12           (172.16.0.0 - 172.31.255.255)
192.168.0.0./16         (192.168.0.0 - 192.168.255.255)
```

Address spaces such as 10.0.0./8 (and the other two listed here) are portions of the IP address space reserved for private networks, or a realm with private addresses , such as home network, for example. A realm with private addresses on a network whose addresses have relevance only to devices living on that network

A system requires a minimum of two network interfaces to perform NAT translation. One of these interfaces will be to the Internet and the other to your internal network. NAT translates all requests from the internal network so they appear to be coming from your NAT system.

How NAT Works

NAT uses IP address and port forwarding to perform its job. When a client in your network contacts a host on the Internet, it is required to send IP packets for that host. These IP packets contain the address information required to enable the packet to reach their destination hosts. NAT uses the following bits of information from this information set:

- The IP address of the source (ex. 192.184.1.36)
- The TCP (o UDP) port of the source (2448)

When the packets from your internal host pass through the NAT gateway, NAT modifies the packets so they appear as if they were sent by the NAT gateway, and not the clients. In our case, NAT will make the following changes:

- Replaces the IP address of the source with the external IP address of the gateway (ex. 198.52.100.1)
- Replaces the source port with a random unused port on the gateway (ex.52244)

The NAT gateway records the address/port changes in a "state" table so it can reverse the changes it made for packets being returned to the internal hosts from the Internet. To the Internet, the NAT-enabled router looks as a single device with a single IP address. All Internet traffic leaving your organization's router to the Internet will share the same source IP address of 192.52.100.1 and all traffic entering that router

from the Internet will have the same destination address of 192.52.100.1. So, NAT in effect hides your internal network from the rest of the world!

It's important to remember that neither the internal hosts, nor the Internet host are aware that NAT is performing this "behind the scenes" address translation – the internal host thinks the NAT system is just an Internet gateway and the Internet host thinks the packets come directly from the NAT server. So, the Internet host sends its replies to the packets to the NAT gateway's external IP (198.52.100.1) at the translation port 52244.

The NAT gateway receives a large number of these replies destined for various internal hosts. How then does it know to which host it ought to send a specific reply packet? That's where the "state" table comes in. The state table is formally called a NAT translation table, and includes the IP addresses and port numbers of the internal hosts as entries in the table. NAT looks up the state table to see which established connection matches the reply packets. When it finds a unique IP/port combination match, telling it that the packets belong to a connection initiated by the internal host 192.184.1.36, it reverses the changes it made originally for the outgoing network packets, and will forward the reply packets coming from the Internet host to the correct host.

The New IP Protocol – Ipv6

About 20 years ago, efforts began to develop a new IP protocol to replace the IPV4 protocol, since the 32-bit IP address space (maximum of 4 billion IP addresses) was starting to get used up fast. Why is this a big deal? If there aren't any IPv4 addresses to allocate, new networks can't join the internet! The new IP protocol IPv6 that'll eventually replace IPv4 is designed with a much larger IP address space, as well as several other operational improvements compared to IPv4.

Besides containing a larger IP address pace, theIPv6 datagram has several changes, including the following:

- Expanded addressing capability: the size of the IP address is increased from 32 to 128 bits to ensure that we never run out of IP addresses. Earlier, you had only unicast and multicast addresses. TPv6 adds the anycast address, which lets a datagram be delivered to any member of a group of hosts.
- Streamline Headers: Several IPv4 header fields have been removed or made optional, making the streamlined shorter header field (40 bytes) allow much fast processing of an IP datagram.
- Handling fragmentation and reassembly: IPv6 removes the time consuming operations of datagram fragmentation and reassembly from the routers and places it on the end systems (source and destination), thus speeding up IP for-

warding inside the network. Other costly operations in IPv4 such as performing header checksums have been eliminated as well.

While new IPv6 capable systems are backward compatible and can send and receive IPv4 datagrams, currently deployed IPv4 systems can't handle IPv6 datagrams. The big question right now is how to migrate all the current IPv4 based systems to IPv6. Ipv6 adoption is growing fast, but it's still going to take a while to switch completely over to IPv6 from IPv4. Changing a network layer protocol such as IP is much harder than changing protocols at the applcition layer such as HTTP and P2P file sharing, for example.

The IPsec Protocol

IPv4, as well as its replacement, IPv6, weren't designed with security in mind. IPsec is a popular secure network layer protocol that's backward compatible with both IPv4 and IPv6. The IPsec protocol is commonly deployed in Virtual Private Networks (VPNs).

You can use IPsec to perform secure private communications in the unsecure public Internet. Many organizations use IPsec to create virtual private networks (VPNs) that run over the basically insecure public Internet.

Routing Essentials and Routing Management

It's quite easy for two hosts within the same LAN to talk to each other – each host needs to send an ARP (address resolution protocol) message and get the MAC address of the other host and that's it. If the second host is part of a different LAN, then you need the two LANs to talk to each other, and you use a router to make this possible.

The router is a network device that sits between two networks and redirects the network packets to the correct destination. If a host is connected to a network with multiple subnets, it needs a router or a gateway to communicate with the hosts on the network. While hosts know the destination of a packet, they don't know the best path to the destination. Either the router knows where the packet ought to go, or it knows of another router than can make that determination.

The router's job is to know the topology of networks plugged into it. When you want to communicate with a server from a different LAN, you don't talk to the other LAN directly. You do specify the destination IP of the host you want to send data to, but also specify the destination MAC address for the router. So the router will get your packets first and check the destination IP and send it to the other LAN, since it knows that the destination IP isn't part of your LAN. And it does the opposite for packets coming from distant LANs to your LAN.

Earlier, I explained how the network layer consists of two major functions: forwarding and routing. When a router receives a packet, it indexes a forwarding table and finds the link interface to which it must send the packet. Routers are where packet forwarding decisions are made in a network. Routing algorithms are used by the router to configure the forwarding tables. Routing is the determining if the best paths (route) from a sender to the receiver, through a network consisting of other routers.

Routing Algorithms

Most hosts are connected to a single router, which is known as the default router (or first-hop router) of a host. The default router of the host that sends a packet is called the source router, and the default router on the receiving host is called the destination router. The purpose of routing algorithms is to find the most efficient path from a source router to the destination router, It's routing algorithms that are at the are at the heart of routing in modern networks, While you might think that the algorithms with lowest cost path is the best, it isn't always so, as you'll learn soon.

Routing algorithms use graphs to formulate a routing problem. The goal of a routing algorithm is to find the least-cost path between a source and a destination. If all links in a graph have the same financial costs and link speeds, then the least-cost path is the same as the shortest path.

There are numerous routing algorithms out there, and you can classify the algorithms according to several criteria:

- Global versus decentralized routing algorithms: Global routing algorithms compute least-cost paths between routers using complete knowledge about the network, such as information pertaining to all the node connectivity between the nodes and all the link costs. Since each node maintains a vector of estimates of the costs of traversing to all other nodes, the decentralized routing algorithm is called a distance-vector (DV) routing algorithm. The link-state and distance-vector algorithms are the two major routing algorithms used in today's Internet.

In the LS algorithm, each node in a network broadcasts to all the other nodes, letting them only the cost of only those links that are directly connected to it. The DV algorithm, on the other hand, means that each node speaks to just its directly connected neighbors, to whom it provides least-cost estimates to traverse from itself to all the nodes it knows about in the network.

- Static versus dynamic algorithms: a static routing algorithm means the routes change mostly because you manually edit a router's forwarding table. A dynamic

routing algorithm changes routing paths periodically, or due to a change in the link costs or a change in the network topology.

Currently used Internet routing algorithms such as RIP, OSPF, and BGP are load-insensitive,

- Load-sensitive versus load-insensitive routing algorithms: In a load-sensitive routing algorithm, the current level of congestion in a link is taken into account - congested links may have a high cost and the algorithm will try to find routes around these congested links. A load-insensitive algorithm doesn't with the current (or past) level of congestion in a link as part of its computation of the link's cost.

Hierarchical Routing

Routers today all use either the LS or the DV algorithm to find the best path for routing the datagrams. With millions of routers in the Interne, the overhead of computing, storing and transmitting routing information among these routers quickly becomes very expensive. There's a different issue as well with maintaining a bunch of routers, all independent from each other – organizations want to administer their routers and their network as they wish, without exposing their network architecture to the entire world. You thus need way to organize routers into a hierarchy to lower the cost of managing the routing information, while allowing companies to maintain autonomous groups of routers (belonging to just their own network).

```
An administrative system (AS) is a way to organize routers so a group of routers is under the admi
```

An intra-AS routing protocol determines routing within an AS and an inter-AS routing protocol determines routing behavior between two ASs.

All routers within an AS run an identical routing algorithm such as the LS algorithm for example, and have complete information about all the other routers in that AS. The algorithm running in an AS is called an intra-autonomous system routing protocol. The routers that belong to a specific AS use the AS's intra-AS routing protocol to forward packets along an optimal path to other routers within the same AS. Note that different ASs may run different intra-AS routing protocols (LS or DV). The concept of an administrative system (AS) of groups of related routers under the same admin-

istrative control and using the same routing protocol among themselves is a crucial feature of how routing works in the Internet today.

One or more routers in each AS act as a gateway router to connect an AS to another AS. If the AS has but a single link that connects it to other ASs, the single gateway router receives the packets from each router and forwards the packet to other AS. However, if there are multiple links and hence multiple gateway routers, each router needs to configure its forwarding table to manage the external AS destinations. The inter-AS routing protocol helps get information from neighboring ASs that they're "reachable", and send this information to all routers internal to an AS. An inter-AS routing protocol is for communications between two ASs, and therefore both ASs must run the same protocol – most commonly, the BGP4 protocol is used for this purpose.

Now that you know what an autonomous system of routers is, IT'S time to learn about the key Routing protocols of which there are three today: RIP, OSPF and BGP.

Common Routing Protocols

As I'd explained earlier, routing protocols determine the path taken by IP datagrams between a source and a destination host. Let me quickly summarize the three important routing protocols that are used in the Internet – in the following discussion, the RIP and the OSPF routing protocols are intra-AS routing protocols, also called interior gateway protocols:

- Routing Information Protocol (RIP): This is the one of the oldest routing protocols, and is used extensively in the Internet, typically in lower layer ISPs and enterprise networks. RIP is a distance-vector protocol that's quite similar to the DIV protocol described earlier. A hop in the RIP terminology refers to the number of subnets traversed along the shortest-path. RIP is limited to the use of autonomous systems that are shorter than 15 hops in diameter. Under RIP each router maintains an RIP table, also called a routing table. The routing table contains the router's estimates of its distance from various other routers, as well as the router's forwarding table. In Linux systems, RIP is implemented as an applcition layer process, and the process named routed ("route dee") executes RIP and maintains the routing information and works with other routed process running on nearby routers.
- Open Shortest Path First (OSPF): OSPF was designed to replace RIP and contains several advanced features such as enhanced security through authentication, and support for a hierarchical autonomous system within a single routing domain. OSPF is a link-state protocol that uses a least-cost path algorithm and also the flooding of link-state information,
- Border Gateway Protocol (BGP): Unlike RIP and OSPF, BGP is an inter-AS routing protocol. It's also known as BGP4. BGP provides an AS a way to obtain

reachability information regarding neighboring ASs and determine the best route to subnets based on reachability and AS policies. In this protocol, routers exchange information over semi-permanent TCP connections.

Static Routing

How does a router know about the networks that are plugged into it? A routing table contains the network information. You can manually edit the routing table, and the routing information you configure here will stay fixed until someone edits the table manually – this routing table is consequently called a static routing table.

A routing table consists of three things:

- Network addresses
- Netmask
- Destination interface

When a network packet gets to the router with a routing table such as this, it'll apply each of the netmasks in the routing table to each destination IP address. If the network address it thus computes matches any of the network addresses in the routing table, the router will forward the packet to that network interface.

For example, let's say the router gets a packet with the destination address of 192.168.2.233. The router finds that the first entry in the routing table doesn't match this destination address. However, the 2nd entry does, as the network address that you get when you apply the netmask 255.255.255.0 to the destination address 192.168.2.233 is 192.168.2.0. The router forwards the packet to Interface 2. If a packet's destination address doesn't match any of our three routes, the packet will automatically match the default address and will be sent to the destination interface for that address - usually this is a gateway to the internet.

A Linux host is aware of multiple standard routes, such as the following:

- The loopback route, which points to the loopback device
- A route to the local area network (LAN), which is used to direct network packets directly to the hosts in the same LAN
- The default route, which is used for packets that are to be sent to destinations outside the LAN.

You use the traditional route command to manage routes. The route command modifies the routing table. Therefore, it's a good idea to first copy the current routing table before issuing the route command! The following command shows how to set the default route for a host with a default gateway at 192.168.1.2:

```
# route add default gw 192.168.1.2 eth1
```

This command adds a default route and sets the gateway to 192.168.1.2. It also sets the interface to this connection through eth1 (a network adapter). Once you issue this route command, all incoming traffic (other than any that were stipulated through other routes) will be sent via your web server, which is 192.168.1.2. Any routes that you add won't survive a system restart, so if you want to make the new route permanent, add the route command with the word up preceding it, to the /etc/network/interfaces file, as shown here:

```
# up route add default gw 192.168.1.2 eth0
```

In order to see if your changes stuck, you don't need to restart the server – just restart the network, as shown here:

```
# /etc/init.d/networking restart
```

On the other hand, if you changed your mind about the new route, you can remove it with the route del command as shown here:

```
# route del default gw 192.168.2 eth0
```

Using the newer ip command instead, you'd add a default route to a host as follows:

-# ip route add default via 192.168.1.1

You can view the routing table using one of the following commands:

- netstat
- route
- ip route

Dynamic Routing

Static routing is OK when dealing with small networks. As the network size grows and you need to deal with a large number of subnets, the overhead involved in manually managing the routing tables becomes prohibitive. Dynamic routing is the solution for managing routing in large networks. Under dynamic routing, each router needs to know only about the networks close to it. It then transmits information about these nearby networks to all the other routers connected to it.

Dynamic routing can be based on one of the three routing protocols you learned about earlier in this chapter: the Routing Information Protocol (RIP), the Open Shortest Path First (OSPF) protocol, and the Border Gateway Protocol (BGP) RIP is simple and is fine for small networks due to its simplicity. This protocol chooses the best route based on the fewest number of routers it needs to traverse – also called the number of hops.

The problem with RIP is that it doesn't take into account network congestion on the routes. OSPF, on the other hand, bases its decision not on the number of hops but how quickly a router can communicate with another router. In doing this, it automatically takes into account any congestion along the route. OSPF is more complex and expensive than RIP and BGP, and is suitable for larger networks.

Network Performance – Bandwidth and Latency

We usually use terms such as "network speed" and "how fast" the network is, when talking about network efficiency. The speed or "fastness" of a network can be really better understood through examining two related but different concepts – bandwidth and latency. Bandwidth is the amount of data a connection can move within a specific amount of time, and is measured in bits per second. Latency is the time it takes for a request to receive the response it's seeking. Latency and bandwidth are related, but they're quite different things – one refers to the capacity of the network and the other, to its speed – obviously, the interaction between the two determines the 'speed" or "fastness" of the network.

The Hypertext Transfer Protocol (HTTP)

Of all the protocols that are supported by the application layer of the network, HTTP is probably the most important, in light of its importance in our Internet dominated world. Well, I do understand that everyone knows what HTTP is! However, I start from the basics so I can explain the critical role played by it in how web applications work. This background will be very helpful during our discussion of load balancing web applications in the next chapter, as well as in the discussion of performance tuning of web applications in Chapter 16.

The Web is the most famous and most popular Internet application. The HyperText Transfer Protocol (HTTP) is the Web's application-layer protocol, and is the foundation of the Web. HTTP is implemented in a pair of client and server programs that communicate through HTTP messages. The HTTP protocol defines the message structure and how exactly the client and the server exchange the messages.

HTTP is a request-response protocol that enables clients and services to communicate with each other. The client can be a web browser and the server can be an application that runs on the server hosting a website. When a client submits a HTTP request to a server, the server returns a response that contains the request status, and most likely, the requested data as well.

Using the HTTP/2 Protocol for Enhanced Performance

HTTP/2, the evolving alternative to the current HTTP protocol (HTTP/1.1) lets you create simpler, faster, and more robust web applications. If you want to provide the same features through HTTP/1.1, you'll need to depend on multiple workarounds, all of which leads to a much more complex and less robust application.

HTTP/2 offers several new optimizations without altering the application semantics of HTTP. Familiar concepts such as HTTP methods, URIs, HTTP headers and status codes remain the same. You can use HTTP/2 instead of HTTP/1.1 in your current applications without having to modify the applications. The most important changes introduced in HTTP/2 are in the framing (within the network transport layer - please see Chapter 2 for a detailed explanation) layer. HTTP/2 provides its optimizations through changing the way data is framed (formatted) and transported between the requester and the responder.

To really cut through the thicket, what HTTP/2 does is to break up the traditional HTTP protocol communications into smaller binary encoded frames. These frames are then mapped to messages that belong to specific streams, which can be multiplexed within a single TCP connection by interleaving the independent frames and reassembling them at the other end. This ability to breakdown/interleave/reassemble messages is the biggest optimization offered by HTTP/2.

While a developer can still deliver a working application with traditional HTTP, the newer protocol HTTP/2 offers vastly superior performance. HTTP/2 reduces latency by the following techniques:

- Using compression to reduce the protocol overhead: In HTTP/1.1, the HTTP headers that contain metadata about the resources being transferred are sent as plain text and add significant overhead (about 500-800 KB on average). HTTP/2 reduces the overhead and improves performance by compressing the request/response metadata using the efficient JHPACK compression format.
- Enabling full request and response multiplexing: In HTTP/1.1, if clients want to improve performance by making parallel requests, they must use multiple TCP connections. HTTP/2 removes this limitation by allowing full request and response multiplexing. All HTTP/2 connections are persistent and just a single connection is required per origin. Overhead is reduced significantly compared to HTTP/1.1, because an overwhelming majority of active connections carry just a single HTTP transaction.
- Facilitating response prioritization: HTTP/2 breaks up the HTTP messages into individual frames and allows each stream to have a different weight. Using these weights, clients can construct and communicate a "prioritization tree" that shows its preferences for receiving the response to tis web requests. Servers use the pri-

oritization information to prioritize the processing of streams through appropriate resource allocation.

- Facilitating server push: HTTP/2 lets a server push additional resources to clients on top of the response to the original request. Server initiated push workflows enhance performance since the server uses its knowledge of the resources required by the client and sends them out to the client instead of waiting for the client to request those resources.

Switching to HTTP2

Before the whole world can switch to HTTP/2 and partake of its superior performance capabilities, millions of servers need to be updated to use binary framing. All the clients that access these web servers must also update their browsers and networking libraries. While all modern browsers have enabled HTTP/2 support with minimal intervention for most users, server upgrades to HTTP/2 aren't really trivial in terms of the time and effort necessary to convert them over to HTTP/2.

As of October 2015, although only 5% of all SSL certificates in Netcraft's SSL survey were on web servers that supported SPDY or HTTP/2, 29% of the most SSL sites in within the top 1000 most popular sites do support SPDY/HTML/2. Also, 8% of the top million sites do support the new protocol. This makes sense since the busiest web sites gain the most by optimizing connections with HTTP/2. Widespread use of HTTP/2 is still several years away since browser vendors currently support HTTP/2 only over encrypted TLS (Transport Layer Security) connections. This means that a large proportion of non-HTTPS sites will continue using HTTP 1.1 well into (many years) the future.

Network Load Balancing

Load balancing is the distribution of a system's load over more than one system, so it can be handled concurrently and faster than if just one system handled all the work. Database servers, web servers and others use load balancing architectures all the time, and so do networks.

A load balancer is either software or a piece of hardware that can distribute traffic arriving at an IP address over multiple servers. Load balancers strive to spread the load evenly among the servers and allow you to dynamically add and remove the servers. The servers stay hidden behind the load balancer and since clients can see just the load balancer, you can add web servers anytime, without disrupting your services.

Benefits of Using a Network Load Balancer

A network load balancer offers the following benefits.

- Security: Routing all traffic between the users and the web servers through a load balancer hides the data center from the users.
- Transparent server maintenance: You can take servers out the load balancer pool without affecting clients (after making sure the active connections are completed). This helps you perform rolling upgrades of your infrastructure without incurring expensive downtime.
- Easily add servers to the pool: when you add new servers to the pool, the load balancer immediately starts sending it connections without any delay, as is the case in DNS load balancing. When we discuss auto-scaling in the context of Amazon or Open Stack in Chapter 9, you'll see how a load balancer can let you scale automatically without any adverse impact on the users.
- Easily manage server failures: you can easily yank web servers with performance issues out of the load balancer pool, to keep connections from being sent out to that server.
- Reduce web server resource usage: if your compliance requirements allow it, you can use Secure Sockets Layer (SSL) offloading (also called SSL termination), to lower the web server resource usage. By doing this, you let the load balancer handle all SSL encryption/decryption. This also means that you can avoid running the SSL based web servers, and have all the web servers processing just HTTP connections and not HTTPS connections, which arrive over SSL.

Load Balancing with DNS

Domain Name Service (DNS) based load balancing (also called the poor person's load balancing) is very easy to set up – all you need to do is provide multiple IP addresses for a domain. That is, you need to add multiple A records for the same domain. When a client attempts a connection to the domain, DNS will hand the client a different IP address sequentially, in a round robin fashion.

Let me illustrate how to set up DNS load balancing with a simple example that uses three IP addresses.

```
example.com      IN          A      126.126.126.130
example.com      IN          A      126.126.126.131
example.com      IN          A      126.126.126.132
Or you can satisfy the domain name just once:
example.com      IN          A      126.126.126.130
                 IN          A      126.126.126.131
                 IN          A      126.126.126.132
```

You can test your DNS load balancing configuration by issuing the nslookup command:

```
# nslookup example.com 127.0.0.1
Server:             127.0.0.1
Address:            127.0.0.1#53
Name:               example.comAddress:            126.126.126.130
Name:               example.com
Address:            126.126.126.131
Name:               example.com
Address:            126.126.126.132
```

DNS load balancing is easy to configure, simple to understand, and very easy to debug as well. However, there are some serious drawbacks as well to DNS based load balancing. One or more of the hosts could end up with a lopsided distribution of the load. The load balancer also keeps sending connection requests to a server that's no longer up.

DNS load balancing was more common a while ago, when load balancers were much more expensive than now. While DNS load balancing is good for a quick spin to learn about load balancing, you need a much more mature and robust load balancing mechanism in a real world production system with heavy internet traffic.

Enterprise Load Balancers

When a client resolves a domain name to an IP address through DNS, the client needs to connect to that IP address to request a web page (or a web service endpoint). To enhance the scalability of your applications, it's a good idea to let a load balancer act as the entry point to your data center, by letting users connect to the load balancer directly instead of to your data center. This will help you scale up easily and be in a position to make infrastructure changes that are transparent to your customers.

DNS load balancing, while simple and cheap (especially in the past when commercial load balancers where much more expensive), isn't really load balancing in the real sense, although it does distribute traffic among a set of servers. DNS has several issues, as summarized here:

- DNS isn't transparent to users. If you remove a server, it may lead to problems because the users may have cached its IP address
- Even when you add a new server, users may not use it since they may have cached your DNS records and so, will keep connecting to the old server(s)
- DNS makes it difficult to recover from failures

For all these reasons, DNS isn't a really viable proposition for production environments.

There are many types of load balancers from which you can choose, with the following three types being the most popular options:

- Software based load balancing
- Hardware based load balancing
- Using a Hosted Sever for load balancing

In the following sections, I describe the three load balancing options.

Chapter 9 discusses AWS and ELB in detail.

Software Based Load Balancing

If you don't want to use a load balancer service such as ELB, you can use one of the many open-source load balancers that are available. Open source load balancers are software based. Two of the most popular software-based load balancing options today are HAProxy, a TCP/HHTP load balancer, and the NGINX web server. In the following sections, I explain these two options.

Using HAProxy for Load Balancing

Since HAProxy is session-aware, you can use it with web applications that use sessions, such as forums, for example. High Availability Proxy (HAProxy) is a pure load balancer, and you configure it in two ways:

- You can configure HAProxy as a layer 4 (see chapter 2 for an explanation of network layers) load balancer. In this case, HAProxy uses just the TCP/IP headers to distribute traffic across the servers in the load balancer pool. This means that HAProxy can distribute traffic for all protocols, and not just for HTTP (or HTTPS), which enables it to distribute traffic for services such as message queues and databases as well, in addition to web servers.
- When configured as a layer 7 load balancer, HAProxy contains enhanced load balancing capabilities such as SSL termination for example, but consumes more resources to inspect HTTP related information. It's suitable for web sites crawling under high loads while requiring the persistence of Layer7 processing.

HAProxy offers three types of services:

- Load balancing
- High availability

- Proxying for TCP and HTTP-based applications

The high availability part of HAProxy is possible because HAProxy, besides performing load balancing by distributing requests among multiple web servers, can also check the health of those servers. If a server goes down, HAProxy will automatically redirect traffic to the other web servers. On top of this, you can use it to help the servers monitor one another, and automatically switch a slave node to the master role if the master node fails.

Using Nginx for Load Balancing

NGINX, the new hotshot web server, also performs load balancing functions. NGINX is more than a pure load balancer – it also contains reverse HTTP proxy capabilities. This means it can cache HTTP responses from the web servers. NGINX is a great candidate when you need a reverse proxy as well as a load balancer

HAProxy versus Nginx

HAProxy offers some benefits when compared to NGINX, as a load balancer:

- HAProxy as its name indicates, provides high availability support and is more resilient, and is easier to recover from failures.
- HAProxy is in many ways simpler to configure
- HAProxy performs better than NGINX, especially when you configure it as a layer 4 load balancer.

NGINX's reverse HTTP proxy capabilities enable it to offer the following benefits:

- Caching capabilities that reduce resource usage of your web services layer
- Easy scaling of the web services layer by adding more servers to the NGINX pool

Scaling the Load Balancer

For many small to medium sized applications, you can run a single HAProxy or NGINX service to handle the workload. For high availability, you may want to configure a hot standby as well on the side. While both NGINX and HAProxy can handle thousands of requests per second and a large number of concurrent connections (tens of thousands), ultimately they both have a hard capacity limit.

When you do reach the capacity limit of your software based load balancer, it's time to scale out the load balancers horizontally, by creating multiple load balancers. When you deploy multiple load balancers, each of them will receive some of the arriving traffic, using a round-robin DNS to apportion the traffic among the load balancers.

While there are several drawbacks to using round-robin DNS in assigning traffic to web servers, it's a harmless way to distribute work among the load balancers themselves, since they are more stable entities in the sense that you don't often replace them, or move them around.

Note that you can use both HAProxy and NGINX even in a hosted environment, if for any reason you want to use your own load balancing setup and not the provider's load balancing service.

Hardware Load Balancing

Hardware load balancers are meant for use within your own data center, and are useful when hosting heavily used websites. Hardware load balancers are the real deal: unlike software load balancers, which you can only scale horizontally (by adding more of them), a hardware load balancer is easier to scale vertically. Typically, a hardware load balancer can handle hundreds of thousands, or even millions of simultaneous connections. These load balancers provide high throughput and a very low latency, all of which dramatically increase the power of your network. Big-IP from F5 and Netscaler from Citrix are the leading hardware based load balancers.

A data center network connects its internal hosts with each other, and also connects the data center to the Internet. The hosts do all the heavy lifting, by serving web pages, storing email and performing computations, and so on. The hosts are called blades and they resemble pizza boxes stacked in a tray, and are usually generic commodity servers, Hosts are stacked in racks, with each rack containing anywhere from 20-40 blades. On top of the rack is the main switch called the Top of Rack (ToR) switch that connects the hosts with each other as well as with other switches in that data center. Each host has its own data center specific internal IP address. Hosts usually use a 1 Gbps Ethernet connection to the ToR switch, although 10 Gbps connections have become common of late.

Each application running in a data center is associated with a public IP address to which external clients send their requests and receive their responses from. When you use load balancing, the external requests are directed first to the load balancer. It's the load balancer's job to balance the workload by distributing requests among the hosts.

A large data center may use multiple load balancers, each dedicated to a set of cloud based applications. A load balancer such as this is often called a "laeyr-4 switch" since it makes its load balancing decisions based on the destination port number ad destination IP address in the packets – both of these entities belong to the network layer (layer 4). In addition to balancing the workload, a load balancer also provides you security benefits, since it provides a NAT-like function by translation the public external IP addresses into the internal IP address of the hosts.

Hardware load balancers are expensive and since they're pretty sophisticated pieces of hardware, require appropriately trained personnel to manage them. However, for non-hosted, heavily trafficked web sites, hardware load balancing is the best approach.

Using a Hosted Load Balancer Service

Managing load balancers gets harder as the size of your infrastructure grows. As with other infrastructure services, it's a good idea to consider using a hosted load balancer service such as the Elastic Load Balancer (ELB) service offered by Amazon AWS. You can benefit from a third-party service such as ELB regardless of whether you host your own applications or host them on Amazon AWS (or Azure).

If you're using a hosted service such as Amazon Web Service (AWS), it's better to use the host's load balancer such as AWS's Elastic Load Balancing (ELB) service than to set up your own. Everything is done for you by AWS and all you have to do is to configure ELB to work with the set of instances that you choose from AWS's dashboard. A load balancing service such as ELB has several major advantages:

- It's simple to set up, as everything is done for you by the service provider (you can do the minimal configuration from your side via a console such as the AWS console)
- Ability to automate load balancer configuration changes
- It's easy to scale up using auto-scaling, with the click of a button, so to speak
- Connection draining features that let you take web servers out the pool without disconnections – when you take a server out, it waits for existing users to disconnect
- A load balancing service such as ELB allows SSL termination
- No upfront infrastructure expenses, and you only pay for what you use
- Easy failure management, as failed web servers are easily replaced

The load balancers I've been describing thus far are public facing load balancers, but Amazon and other cloud providers also let you configure internal load balancers as well. Internal load balancers sit between your front-end web servers and your internal services, and let you reap all the benefits of load balancing inside your stack.

Modern Networking Technologies

The widespread use of cloud provider supported data centers as well as large enterprise data centers driven by big data has led a large number of interconnected servers, with an overwhelming portion of the network traffic occurring within the data center network itself. Current computing trends such as the increasing popularity of big

data, cloud computing, and mobile network traffic have contributed to a tremendous increase in the demand for network resources. In the following section, I briefly discuss two important concepts that let you quantify a network's performance: quality of service (QoS) and quality of experience (QoE).

Quality of Service (QoS)

QoS is a set of measurable performance characteristics of a network service, which a service provider can guarantee via a service level agreement (SLA). QoS commonly includes the following performance properties:

- Throughput: this is measured in bytes per second or bits per second for a given connection or traffic flow
- Delay: also called latency, this represents the average or maximum delay
- Packet Jitter: the maximum allowable network jitter (jitter is the variation in the delay of received packets due to network congestion, improper queuing, or configuration errors)
- Error rate: usually the maximum error rate in terms of bits delivered in error
- Packet loss: defined by the fraction of packets not dropped
- Availability: expressed as a percentage of time
- Priority: a network can also assign priorities for various network traffic flows to determine how the network handles the flows

When you move to a cloud environment, putting a QoS agreement in place means that the business applications get some type of minimal performance guarantees. QoS is a major determinant of resource allocation in a cloud environment.

Quality of Experience (QoE)

While QoS is a measurable set of performance properties, QoE is more intuitive and subjective, being the impression of quality felt and reported by users. It's how end users perceive the network performance and the quality of service, regardless of the actual QoS metrics. By capturing the end user's perception of network quality, QoE provides a different, and in many ways, a more usable measure of quality. However, while it's fairly easy to measure QoS, QoE is not so amenable to quantification and it's hard to come up with accurate measures for this component.

Both QoS and QoE depend to a great extent on network routing and congestion control, which I discuss in the following section.

Cloud Networking

Obviously one uses the Internet as part of a cloud setup with an external provider, but that's usually only a small part of the networking infrastructure needed to support cloud based operations. Often you also need high performance, highly reliable networking to be established between the cloud provider and the subscriber. Here, at least some traffic between the organization and the provider bypasses the internet, using dedicated private network facilities managed by the cloud service provider. In general, cloud networking includes the network infrastructure required to access a cloud, including connections over the internet. The linking of company data centers to the cloud also includes the setting up of firewalls and other network security mechanism to enforce security policies.

Routing and Network Congestion Control

Routing and network congestion are two key aspects of a network that have a major bearing on how efficiently a network transmits network packets. Efficient transmission of network traffic has a direct bearing on both QoS and QoE.

The internet's main job is to move packets from one place to another, using alternate paths or routes. The routing function is essential since there's often more than one path for a packet to travel to its destination. Routers usually employ an algorithm based on performance, such as least cost routing, which minimizes the number of hops through the route. Cost could be based on throughout or time delay. The router's job is to accept network packets and forward them to other routers or to their final destination, using what are called forwarding tables.

Routers base their routing decisions on various routing protocols in order to forward packets through an interconnected set of networks. An autonomous system is a set of routers and networks managed by an organization, with the routers exchanging information through a common networking protocol. Within an autonomous system, a shared routing protocol called an interior router protocol (also referred to as an interior gateway protocol) passes the routing information between the various routers. OSPF is one such protocol.

A protocol for passing routing information between routers belonging to different autonomous systems is called an exterior router protocol (or an exterior gateway protocol), with BGP (Border Gateway Protocol) being an example of this type of protocol.

A router's control function includes activities such as executing routing protocols and maintaining routing tables and handling congestion control policies. Network congestion occurs when internet traffic exceeds the capacity or if the network doesn't manage the traffic in an efficient fashion. When a router is overwhelmed by incoming network packets, it can either discard the newly arrived packets (not good!), or manage the traffic using flow control, which means neighboring routers will be forced to share the excess demand. As the network load increases, the queue lengths at various nodes get longer and nodes start dropping packets. This of course leads to the sources retransmitting these packets, and in the extreme case, the capacity of the network in effect eventually diminishes to almost nothing.

Two new network innovations that many service and application providers are adopting these days to combat network congestion are software-defined networking (SDN) and network functions virtualization (NFV).

Software-Defined Networking

Thus far in this chapter, I've been busy describing protocols, network layers, packets and frames, but the buzz these days in networking is all about something called software-define networking (SDN). Although there's still more hype regarding SDN than actual deployments, it's clear that traditional data centers are slowly metamorphosing into software based data centers. Software-defined networking is a way of organizing the functionality of computer networks. With SDN, you can virtualize your network, which enables you to exert greater control over it and also provides support in traffic engineering.

SDN is the outcome of a search initiated around the year 2004 for a better and modern network management paradigm, and all indications are that it's slowly and quietly replacing traditional networking as it provides a way to meet the increasing demands made on networks by new trends such as cloud, big data, mobility and social networking.

Limitations of current networks

Networking models today are the product of architectural and protocol decisions made around 40 years ago. Networks were expected to be static entities and their topologies weren't expected to change significantly over time. Until the advent of virtualization, each of the applications was associated with a specific physical server which had a fixed location in the network. Virtualization has changed all of this. Applications are now usually distributed across multiple virtual machines, all of which can be moved around to optimize or balance the server workload.

Traditional networking uses the concept of subnetted network segments and routing, along with addressing schemes and namespaces. The migrating of VMs across a network causes the physical end points of a network flow to change and creates chal-

lenges for traditional networking. In addition to the problems introduced by virtualization, networks are used today to deliver various types of services such as voice, video and data. Traditional legacy networks are static in nature, and can't dynamically keep up with the fast changing network traffic and application demands.

Networks today are built by connecting large numbers of complex network routers which accept data packets from applications and forward them to the next router on the path to the packet's final destination. The router's operating system controls the packet forwarding function and is designed to work with the vendor's hardware platform. Specialized routing protocols use the operating system and the proprietary packet-forwarding hardware to send the packets along to their destinations.

There are inherent problems with the traditional networking model. In order to change the network behavior, you need to configure each of the routers and issue commands in the proprietary language of the router vendor. In this closed environment, it's difficult for routers to interact easily with the rest of the network components.

The three "Planes' in Networking

Computer networks use three distinct planes to perform their tasks – the data plane, the control plane and the management plane. Let me explain these three planes briefly:

- Data Plane: processes the packets received by a network by using the packet header information to determine whether the packet should be forwarded or dropped. If it decides to forward the packets, it must determine the host and port to which it should send the packet.
- Control plane: this plane computes the forwarding state used by the data plane to forward network packets, by calculating the state using distributed or centralized algorithms.
- Management plane: this plane coordinates the interactions between the data and the control planes.

While the data plane uses abstractions such as TCP/UDP and IP, thus creating a reliable transport layer out of a set of unreliable transport layers, the control plane has no such abstractions. There's no unifying abstraction underlying the control plane, with a bunch of control plane mechanisms created to achieve different goals. For example, a routing control plane may be used to provide access control lists or firewalls, by influencing the routing of packets through controlling the calculation of the forwarding state. Each of the control planes solves a specific narrow problem, but they all can't be used together.

The limited functionality of the control planes means there's no one solution that contains all the control plane functionality. Why do we need to worry about the fact that the different "small bore" control planes can't talk to each other? Well, this type of architecture has an impact on the bottom line.

In a legacy network, when you have a large amount of data that needs to be moved, by default the fastest network link is employed, although it's the most expensive. Since the applications can't really communicate with the network due to the way the control plane is designed in traditional networks, it never finds out about the availability of slower but cheaper alternate network links.

In a traditional network, it's not just the routers that perform the data, control and management functions in an integrated fashion – all the other network devices, such as a network bridge, packet switch etc. perform these functions in the same fashion as a router.

SDN constitutes the set of instructions that govern the control plane in a modern network. SDN's approach is quite different from that of today's distributed network routing protocols. SDN essentially involves the simple computation of a function. It computes functions on an abstract view of the physical network layer, ignoring the actual physical infrastructure of the network. This allows the network engineers to manage network traffic by ignoring the physical network design. The networking operating system takes the SDN generated function and distributes it to all the network switches.

Server virtualization is a big reason for the advent of SDN. While virtualization allows you to easily migrate servers across virtual machines for load balancing or for high availability purposes, it doesn't play too well with traditional networking. For example, the virtual LAN (VLAN) that's used by a virtual machine must be assigned to the same switch port as the host server. Since you may often move virtual machines it means you need to reconfigure the VLAN every time you move the VMs around. Since traditional switches perform both the control and data plane functions, it's hard to modify network resources and profiles with these switches. The need to provide a rapid network response to the use of virtual servers has been a big motivating force in the move to SDN.

In traditional networks, the control plane needs to compute the forwarding state of network packets, consistent with the low level hardware and software and the entire network topology. In addition it must ensure that it inserts the forwarding state into all the physical forwarding boxes in a network. A new protocol means that the control plane has to redo all of the work, which isn't very efficient.

Under SDN, the traditional three plane functionality of the router is broken up. Router hardware has less work to do in this architecture: all it needs to do is provide

the data plane functionality – that's it! A software application running on a separate platform and connected to the router provides the control and management planes.

Performing all the routing functions within the router itself means that all routers must implement the same routing and control protocols. In SDN, a central controller will perform the tasks such as routing, naming, declaration of policies and security.

```
The SDN control plane consists of one or more SNC controllers, which define data flows that occur
```

Defining Functions for the Network Control Plane

When designing the network control plane, you need three types of functions:

- A forwarding function
- A network state function
- A configuration function

I describe the three functions briefly in the following sections.

The Forwarding Function

The packet-forwarding function needs to be implemented differently from how the network switch is implemented. This function abstracts the details of low level hardware and software involved in the creation of the network switch. The switch should be able to use any underlying low level mechanism and the software that runs in the switch shouldn't impact the forwarding function.

The OpenFlow interface, which is a set of application programming interfaces (APIs), is one way to link the control plane software and the network switch.

The Networking Function

The goal of the network state function is the presenting of a graph-like global network view, with necessary network information such as network delay and the recent loss rate. This global network view functionality is provided by a network operating system (NOS), and the controlling software uses the network graph to make decisions. The global network view is continuously updated with the changes occurring with the network switches. The NOS has the capability to update the switches to control the packet forwarding.

In a traditional network design, network switches are responsible for routing packets of user information among themselves. The switches use the information to update their view of the global network, and modify how they forward the network packets.

Legacy network control planes implement peer-based distributed routing algorithms, such as the shortest path first (SPF) algorithm. The problem is that the algorithms are

complex, and the distributed decisions by the switches are based on only a partial knowledge of the network's global state.

In an SDN based network, there's a general purpose software algorithm that runs on the NOS servers and determines the network topology by polling the network switches. Based on the responses from the switches, the software creates a global network view. The global network view acts as the basis for various types of control programs such as routing or traffic control programs. Under SDN, instead of distributed decision making based on imperfect network information, a centralized control program based on the global network view is used to determine how switches ought to forward the packets.

With the centralized decision making through the control program, redesigning the network to route packets differently is easy – all you need to do is modify the control program. You don't have to create a brand new distributed routing algorithm, as is the case in a traditional network.

The Configuration Function

The control program doesn't concern itself about how the actual routing behavior is implemented. That is, the control program doesn't have to configure the routing tables on all network nodes to implement a routing algorithm. A virtualization layer is added to the SDN model to translate the control program's routing instructions into configuration commands for the switches in the physical network. This virtualization layer updates the global network view to reflect the routing decisions made by the control program. The layer hands the updated global network view to the NOS, which actually configures all the switches in the underlying physical network. It maps the packet-routing control commands and maps them to the physical switches. The OpenFlow interface facilitates the interaction between the NOS and the switches.

Since routers are left with just the data plane functionality, they don't need to be complex things any longer. You can now make do with basic packet-forwarding hardware based routers to communicate with the NOS through the OpenFlow (or a similar) protocol. Also, since under SDN the networking intelligence has been separated and moved to the SDN layers, switches can be mere commodity hardware as well.

Probably the most important benefit of SDN is the fact that by enabling network virtualization, SDN lets you migrate easily between your current network and the cloud. SDN makes it easy to export your virtual topology to the cloud and ignore the physical design of the network. Network teams can create a network policy statement using their network topology and replicate this policy statement in the cloud. Under SDN, the control plane is not on hardware but exists in software. You can easily perform large scale simulation of the control plane and test new network designs before moving into production.

In traditional networks, the control plane is part of a proprietary switch or a router box, whereas in SDN it's a program, with software determining how to forward network packets. Instead of designing a network, you'll be programming networks based on the abstracted view of the physical network. The software you use to program the network will be independent of the hardware used by the network. Troubleshooting SDN based networks is also simpler since you can simulate the programs once you identify the source of a network related problem such as incorrectly forwarded network packets, for example.

Network Functions Virtualization

In today's computing world, storage and servers have already been virtualized. Network functions virtualization (NFV) allows the virtualization of the physical networks. Once it virtualizes the network, NFV enables software applications that use the network to reconfigure the network as they see fit, thus letting the network provide the best service possible to the applications.

NFV breaks away the set of essential network functions such as routing, firewalling, NAT and intrusion detection from vendor controlled proprietary platforms and implements these functions in software, using virtualization. If NFV sounds very similar to SDN, it's no mystery, as they share several features and objectives. Both SDN and NFV believe in:

- Using software instead of hardware to provide network functions
- Using commodity hardware instead of proprietary network platforms
- Using standardized APIs

The key thing to remember is that both SDN and NFV decouple or break up components of the traditional network architecture. While SDN decouples the data and control planes of network traffic control, NFV removes the network functions from proprietary hardware through virtualization and implements the network functions as software. You can use either SDN or NFV by itself, but of course, employing both together will get you the most benefits. If you use SDN by itself, you'll be implementing the control plane functionality on a dedicated SDN platform. If you use NFV as well, you can implement the network data plane functionality on a VM and the control plane functionality on either a dedicated SDN platform or an SDN VM.

The OpenFlow Protocol

The OpenFlow protocol is a new network protocol that has been created to enable software-defined networks. The OpenFlow protocol provides structure for the messaging between the control and data planes and provides external applications access to the forwarding plane of a network switch or router.

Traditional networks tend to stay static over time as a result of their complexity. In order to move a device to a different location on the network, the network administrators must modify multiple switches, routers, firewalls and updated ACLs, VLANs and other protocol based mechanisms that work on the device and the link levels. The OpenFlow protocol was designed to solve the problems posed by legacy network protocols. In SDN architectures OpenFlow is the interface between the control and forwarding layers and lets you directly access the network devices such as switches and routers in the forward plane. Standard network protocols can't offer this type of functionality, and thus OpenFlow is a necessary ingredient to move the control part out of networking switches and into centralized control software.

With this review of networking behind us, let's turn to how modern web applications are architected, and how you achieve scalability in a world of modern web applications, web services and microservices.

CHAPTER 3

Scalability, Web Applications, Web Services, and Microservices

Supporting web sites and web applications is often a key function of Linux system administrators. An overwhelming majority of the world's web sites and web applications run on Linux. In the old days, when it came to supporting web based applications, all an administrator had to know was how to set up a web server such as the Apache HTTP server, and a few things about DNS and how to connect the web front end to the backend databases.

Over the past several years, there have been truly revolutionary changes on many fronts, changes which require you to be conversant with a lot more technologies that have come to play a critical role in driving web applications. The rise of web based applications and the consequent concurrency issues it gave rise to due to massive user bases have called for innovations in virtually all areas of the traditional web application architecture. In addition to newer application architectures, there are also vastly more moving pieces in a typical web application today than in the applications from the previous generations. This chapter has two major goals:

- Explain the concept of scalability and how you can enhance it using modern approaches, in all areas of a web application, such as the front end and back end web servers, databases, caching servers, etc.
- Introduce you to several modern innovations such as new application architectures, NoSQL databases, modern caching concepts, asynchronous processing models, high availability, and newer, more efficient web servers

Scalability is the ability of applications to handle a large number of users, data, transactions or requests without a slowdown in response time. Modern application architectures utilize several strategies to achieve scaling for each of the components of the

application stack. Throughout this chapter, our focus is squarely on scaling using modern techniques.

Web applications are often deluged with large numbers of simultaneous users, making concurrency a tricky issue. Over the past 10 years or so, the need for higher scalability has driven several significant developments in web architectures, with the adoption of newer concepts such as web services and microservices. This chapter provides an introduction to these architectures.

Traditionally, web applications have used the open source database MySQL and the open source Apache web server to power them. While the established user base of MySQL and the Apache Web server is still very high, the pursuit of scalability and concurrency has led to newer database and web servers that have been explicitly designed to provide scalability.

Modern web servers designed with high concurrency in mind, such as the NGINX web server and modern databases that are based on a non-SQL approach to data processing such as MongoDB and Cassandra have come to become major players in today's application architectures. This chapter explains the way these modern tools work, and how they provide the benefits that make them significant components in the modern many application architectures.

This chapter discusses several concepts that are used by developers, but administrators need to know about these crucial concepts so they can better serve their clients, both within and outside their organization. For example, learning about popular programming concepts such as Model-View-Controller (MVC) architecture and the reasons for the popularity of the MEAN stack for building web applications helps understand what the developers in your organization are doing, and why. Similarly, understanding the basics of web performance helps you provide more efficient services for your external customers (Chapter 16 discusses web performance optimization in detail)

Two of the most popular web application frameworks today are what are generally referred to as the MEAN Stack and Ruby on rails. MEAN stands for a framework that uses the following components together to drive web applications: Mongodb, Express, Angular JS and Node.js. Ruby on Rails is a very popular framework for developing web applications, and relies on the principle of "convention over configuration".

This chapter explains the Model-View-Controller (MVC) architecture that underlies modern web applications. Following this, it explains both the MEAN application stack as well as Ruby on Rails, which is a highly popular modern web application framework. You'll also learn about the new paradigm in web applications, the Single Page Application (SPA).

Microservices are increasingly becoming the norm, more or less supplanting the previously popular concept of web services. You'll learn about both web services and microservices in this chapter, and also find out the key differences between these two approaches.

Scaling and Common Datacenter Infrastructures

Before a client request even arrives at a company's data center, it usually traverses through third party entities such as a GeoDNS or a CDN service. Once it arrives at the data center, web traffic is handled by multiple technologies, some of which fall under the front end type and others the back end architecture components. Let's list the typical components of front end and back end technologies.

While managing a data center with the multiple layers shown here does add to the complexity, it helps you scale your environment. For example, load balancers reduce the load on the web application servers in the front. Similarly, backend web services need to deal with a lower amount of traffic since the application level caches and the message queues handle some the load.

While I list a fairly large number of components for a data center, by no means do you need all of these in your environment. If your applications are simple, they may not need all the complex layers, such as frontend cache server, message servers, search servers and backend cache servers, for example. The objective here in listing these components is to make you aware of the potential complexity of a data center architecture.

Scaling web applications (and your entire computing environment for that matter) means knowing how to scale all the components of a typical architecture, including both the front line and backend technological components of a data center.

Front End and Back End Web Technologies

You probably are familiar with the commonly used terms "front end" and "back end" when dealing with web applications. The front end in this context refers to the application code, HTTP and web servers, load balancers, reverse proxies and Content Delivery Networks (CDNs). The backend layers include web services, object caches, message queues and databases. I discuss how you can enhance scalability and performance by using several strategies at the front end as well as the back end of your applications. Here's a listing of the main technologies I explain in this chapter:

- Load Balancers
- Web servers (web application servers)
- Caching servers (from caching servers)
- Reverse Proxy Servers

- Content Delivery Networks (CDNs)
- Web services and Microservices
- Object caches
- Messaging (queuing) services

The Front End Technologies

When dealing with web applications, front-end technologies are responsible for rendering the user interface and handling the user connections. Typically, the front-end technologies are the following:

- Reverse proxies
- Content Delivery Networks (CDNs)
- Hardware (or software) load balancers
- Domain Naming Service (DNS)
- Front cache servers (optional)
- Front-end web application servers

```
The web application layer is the presentation layer and its job is to simply handle user
```

The Back End Technologies

Back end technologies include the following:

- Web services: Web services constitute the heart of the application and contain the business logic. These services use either REST or SOAP (I explain both of these in this chapter) to communicate with the front-end applications. If you keep the web services stateless, scaling them involves adding more servers to run the web services.
- Cache servers: these are used both by the front-end application services as well as the backend web services to reduce latency by storing partially computed results.
- Message queues: both front end applications servers and the web service machines can send messages to the queues. The message queues help postpone some processing to later stages and also delegate work to worker machines which process the message queues. The queue worker machines perform work such as taking care of asynchronous notifications and order fulfillment which typically require significant time to complete.
- Databases: a data center can contain both traditional relational databases such as MySQL and several big data stores such as Hadoop, in addition to NoSQL database such as MongoDB and Cassandra. Regardless of the type of database, scala-

bility is a running theme common to all databases, since data sizes are usually arge, regardless of the type of data that's stored and processed.

- Other servers: These can include batch processing servers, search servers, file servers and lock servers.
- Third-party services: these include services from both CDN providers and the providers of various business services such PayPal and SalesForce, for example.

Scalability of Applications

We all know in general what scalability means – it's the ability to handle an increasing number of users and their requests without deterioration in the performance of the applications. Scalability contains the following two major dimensions:

- Ability to handle higher concurrency: as applications become more popular, the number of users simultaneously accessing the web sites starts rising, bringing to the fore a special set of problems, such as an increasing number of open connections and active threads, not to speak of higher I/O for the storage subsystem as well as the network, and a higher load on CPU. Latency and response time become crucial measures of the ability of a system to handle high concurrency usage patterns.
- Processing More Data: As a website becomes more popular, more users will be making requests and your system will be handling increasingly larger amounts of data. Processing data involves disk I/O as well as network I/O, and therefore the system has to keep up with the higher data loads without slowing down the speed of processing.

Scalability and performance are intertwined, and present two sides of the same coin. A system that can't process things fast enough can't scale very well. There are two major approaches to scaling applications – vertical scaling and horizontal scaling, as I explain in the following sections.

Scaling Vertically

Vertical scaling is usually the first response to meeting higher scalability requirements. Under vertical scaling, you seek to upgrade the processing power of your systems without reconfiguring your applications or the components of your application architecture. When it comes to upgrading the processing power of servers, the most common approaches are to increase the RAM allocated to the servers, or sometimes, even switch to different servers with more and/or faster CPUs. The following are all ways in which you can scale up.

- Increase RAM to reduce the disk I/O – computations in memory are much faster as compared to computations on disk, so for most database dependent applications, allocating more RAM leads to dramatic improvements in performance.
- Get faster network interfaces: you can often gain throughput by upgrading your network adapters.
- Move to more powerful servers: you can move your databases and web servers to servers with more CPUs and virtual cores. The CPUs themselves can also be faster than the CPUs you're currently using.
- Improve the disk I/O: One of the first things you can do here is to see if you can invest in speedier hard disk drives (higher RPMs). Web applications in particular predominantly use random reads and writes, which are much faster when you use solid-state drives (SSDs). If it makes sense from the cost point of view (while they were extremely expensive to begin with, SSD costs have been continuously dropping), SSDs could dramatically speed up disk I/O.

While vertical scaling can and does enhance scalability in most cases, and offer the benefit that it doesn't require you to make any changes to your application architectures, it has inherent limitations over the long run. Scaling up is usually extremely expensive and not always cost effective. You'll find that you get a smaller payoff by spending extra money on buying these muscular infrastructure components (ultra-fast disk drives, faster CPUs, and more powerful RAID controllers, for example) with higher specs.

Even assuming that you can afford the increasing cost (the cost increases nonlinearly) of the heftier hardware and other components to support vertical scaling, you'll eventually hit a performance wall due to the sheer fact that there are hard limits on how much memory a system can handle, and how fast disk drives can spin.

Content Delivery Networks (CDNs) also help significantly improve scalability by taking some load off your web servers for providing static web content. I discuss CDNs in the following sections.

Scaling Horizontally

While vertical scaling takes the approach of bulking up the existing infrastructure so it can process more things faster, horizontal scaling enhances your infrastructure's processing capacity by adding more servers to your infrastructure. You can add more database servers or more web servers, for example, to support a growing volume of business.

The principle of horizontal scaling underlies the massive infrastructures built up by behemoths such as Amazon and Google, which run data centers with hundreds of

thousands of commodity servers, as well as the increasing popularity of big data processing models such as Hadoop and Mesos.

Content Delivery Networks

One of the best ways to offload some of an application's web traffic is to employ a third-party content delivery network (CDN) service, such as Akamai, CloudFare, Rackspace (Rackspace Cloud Files Service), and Amazon's Elastic Compute Cloud (EC2). Currently Akamai is the worldwide market leader. Content delivery networks are hosted services using a distributed network of caching servers that work somewhat similar to the way caching proxies work, and are typically used for distributing an application's static files such as images, CSS, JavaScript, PDF documents and videos.

Users that need to download the static content connect directly to the CDN provider instead of your web server. The CDN will serve the requested static content to the users, and if it doesn't have the requested content, it gets that content from your web servers and caches it, and so it can send it along to the users directly for all subsequent requests.

CDNs are highly cost effective for most organizations. They do things at a fraction of what you'd have to spend if you were to set up complex and hard to maintain network infrastructures across the globe to cache the content in your own private infrastructure.

While CDNs are primarily meant for providing static content, they aren't limited to it. CDNs can also serve dynamic content for your websites, offloading more work from your own data center to the CDN provider to process content. As Chapter 9 explains, you can configure Amazon's CloudFront service to provide both static and dynamic content. Note that in addition to simply serving static web content, CDNs can also work as a HTTP proxy, as I explain later.

If your needs aren't big, you can set up a simple Apache server to deliver your static files so you can take needless load off your web application servers. You can use the single server CDN setup until your needs require you to scale up, and that's when you can splurge for a third-party CDN provider. In addition to scalability, third-party CDNs also offer faster performance since a globally distributed CDN can serve its content from the nearest server.

Since CDNs use caching to do their work, they are similar to proxy servers for caching purposes. In fact, CDNs use HTTP headers to cache content, in the same way as proxy servers. Large web applications benefit immensely from a CDN since CDNs not only reduce your network bandwidth usage, but also decrease the network latency by serving content to users from a the nearest cache server from their distributed network of cache servers.

As mentioned earlier, you can configure a CDN to cache both static and dynamic content. For example, Amazon CloudFront can serve both static and dynamic content for your applications. Typically, however, web applications use CDNs to cache static files such as images, CSS, PDF documents and JavaScript.

How large websites scale

When scaling for large audiences, organizations use multiple data centers. Spreading the date centers across the world enhances user experience and also provides higher availability. Two of the most common strategies used to service a global audience are the use of a GeoDNS and the use of edge-cache servers.

GEODNS

Normal DNS servers resolve domain names to IP addresses. A GeoDNS takes this concept a bit further: they serve IP addresses (of the web servers or most commonly, the IP addresses of the load balancers) based on the location of the client, allowing clients to connect to web servers running closer to where they're located. Obviously, minimizing network latency is the goal in using a GeoDNS strategy.

Edge-cache Servers

Large companies can also reduce network latency by hosting several edge-cache servers across the world. Edge-cache servers are HTTP cache servers that are situated closer to the clients, enabling them to (partially) cache their HTTP content. The edge-cache servers handle all requests from a client browser and can serve entire web pages if they have it in the cache, or they can use cache fragments of HTTP responses to assemble complete web pages by sending background requests to the company's web servers.

Both GeoDNS and edge-cache servers can be used together with CDNs. For example a company's Asian customers will resolve the company's domain name to an IP address of an Asian edge server. The customers will be served results cached by an edge-cache server or from one of the company's own web application servers. The company's CDN provider loads static files such as JavaScript files from the nearest data center of the CDN provider in Asia. The bottom line is to keep latency low, and cut the costs in achieving that goal.

Scaling Web Applications

Scalability of both the front end and the backend of web applications is crucially affected by the concept of state. To understand how this is so, you need to know the difference between stateful and stateless services:

- A stateful service holds data relating to users and their sessions. Data in this context could relate to user session data, local files or locks.
- A stateless service (or server) doesn't hold any data. Data in this case captures the state and stateless services let external services such as a database handle the maintenance of a client's state. Since they don't have any data, all the service instances are identical.

Stateless services don't retain state or any knowledge relating to users between HTTP requests made by the users. Since a stateless service stores the state in an external shared storage (such as a database), all stateless servers are virtually the same - there's no difference among them.

Having to maintain any type of state has a negative impact on your system's ability to scale. For example, keeping web servers (both front-end and backend web service servers) stateless lets you scale the servers simply by adding more servers. State has relevance both in the front-end layer as well as on the backend, as the following sections explain.

Managing state at the front end

For front end servers such as web servers, there are two main types of state that need to be managed – HTTP sessions and file storage. Let's review how web applications handle these two types of state.

HTTP Sessions and State

While the HTTP protocol itself is stateless, web applications create sessions on top of HTTP to keep track of user activity. There are several ways to track session state, as explained here:

- Using session cookies: Web applications usually implement their sessions with cookies, and the web server adds new session cookie headers to each response to a client's HTTP request, enabling it to track requests that belong to specific users.
- Using an external data store: Using cookies is a simple and speedy approach when data is small, but if the session's scope gets large, cookies can slow down the web requests. A better strategy may be to store the session data in an external data store such as Memcached, Redis or Cassandra. If the web application is based on a Java type JVM-based language, you can store the session data in an object clustering technology such as Terracotta instead. The web server remains stateless as far as the HTTP session is concerned, by not storing any session data

between web requests. Instead, the web application grabs the session identifier from the web requests and loads the pertinent session data from the data store.

- Implementing "sticky sessions" through a load balancer: In this strategy, the load balancer takes over the responsibility for routing session cookies to the appropriate servers, by inspecting the request headers. The load balancer uses its own cookies to track the assignments of the user sessions to various web servers. Note that under this strategy the web servers do store the local state by storing session data.

File Storage and State

In addition to the state of the applications, file storage is another important type of web application state handled by web servers. If you're wondering how file storage plays a role in managing state, it's useful to remind yourself that typically, users can both upload content to the web servers and also download various files generated by the web applications.

Instead of reinventing the wheel, you can set up your own file storage and delivery system using open-source data stores and related tools to scale your file storage system. This lets you take advantage of scalability features such as partitioning and replication that are built into these data stores. For example, GridFS, which is an extension that's part of MongoDB, can split large files and store them inside MongoDB as MongoDB documents. Similarly, Netflix's Astyana Chunked Object Store takes advantage of another open source data store, Cassandra, by using its partitioning, failover and redundancy features, and by building file storage features on top of the data model.

Auto Scaling

Auto scaling is the automatic adding and removing of processing power, such as adding or removing of web servers based on changing workloads. While scalability in this chapter mostly deals with scaling out to efficiently address increasing workloads, auto scaling is the automation of an infrastructure's capacity by automatically adding to or removing servers from the infrastructure based on the workloads. Auto scaling is really more relevant in a hosted environment such as Amazon AWS, where, by using Amazon's auto scaling feature one can considerably reduce their hosting costs. For a client that's running a web application by hosting it on AWS, handling a holiday rush is a breeze with auto scaling – and once the holiday rush business is taken care of, you won't need the beefed up infrastructure, so the system automatically scales itself down. Just as you can use a service such as Amazon's Elastic Load Balancer to auto scale your front end web servers, you can also auto scale the web servers hosting your web services.

While you may be able to manage your proprietary file storage system by leveraging the open source data stores, a far better strategy would be to simply delegate the whole area of distributed file storage and delivery to inexpensive third-party services such as Amazon's Simple Storage Service (S3). You can use these types of storage services even if you aren't using AWS to host your web applications.

Regardless of whether you use a service such Amazons S3 or handle the file storage yourself, it's a good idea to employ a content delivery network (CDN) to send files to the users. The CDN can use Amazon S3 or public web servers to download and cache public content. If the files that users need to download are private files, you can store them in Amazon S3 again, but in private instead of public buckets, or use private file servers if you're handling the data storage yourself. The web application servers will then fetch the private files from S3 or your own private file servers, enabling users to download them.

Web servers can be front-end web servers or servers that host the web services in the backend.

Other Types of State

In addition to session data and file storage, other types of state such as local server caches, resource locks and application in-memory state negatively impact state. Caching in particular is tricky since it does enhance performance and reduce the workload of the database and web services, as you'll see later in this chapter.

Resource locks synchronize access to shared resources but it's very difficult to implement those locks correctly in practice, and you should instead try to move the state out of your application servers. You may want to consider a distributed locking service such as ZooKeeper or Consul.. As this chapter explains later on, ZooKeeper not only is used for distributed locking (for Java based applications), but also for leader election and managing runtime cluster membership information. For scripting languages such as Ruby, you can use a simpler lock implementation based on operations within a NoSQL database such as Memcached or Redis.

Regardless of whether you use ZooKeeper or Memcached or a different implementation of a locking service, the principle here is that you remove the state related functionality from the web server and move it to a separate, independent service. This makes it easy to scale out your web applications.

Scaling the DNS

1. As Chapter 8 (Cloud and Amazon AWS) shows, Amazon's Route 53 services is integrated extremely well with Amazon AWS and lets you easily configure DNS through your AWS web interface. Route 53 uses latency based routing to route clients to the data center nearest to them. In some ways this is similar to GeoDNS which I explained earlier in this chapter, and is probably even better in some ways than GeoDNS, since it bases its routing decisions on latency instead of location.

If you aren't hosting your infrastructure on Amazon, not to worry, as there are several DNS services out there such as easydns.com and dyn.com that provide services similar to those offered by Amazon's Route 53 service.

A CDN is commonly used for serving static files such as images, CSS and JavaScript files, but you can use a CDN to proxy all web requests arriving at the web servers, if you so wish.

Scaling the Load Balancers

Amazon's' Elastic Load balancing (ELB) is part of Amazon's auto scaling feature that automatically adds servers to meet rising workloads and also replaces the web servers automatically when the servers fail. Since ELB scales transparently and is highly available, you don't need to worry about managing it.

In addition to Amazon's AWS, other cloud providers such as Microsoft's Azure offers the Azure Load balancer and Rackspace offers the Cloud Load Balancers and OpenStack comes with LbaaS. All of these cloud providers offer similar load balancing services as ELB. You can scale the load balancing easily with any one of these services, regardless of whether you host your infrastructure on those providers or not.

Scaling the Web Servers

The programming language in which you implement your front end applications, as well as the type of web servers to use, are both crucial decisions you need to make when it comes to architecting the front–end web servers, which provide the presentation logic as well as serve as an aggregation layer for the web service results. Although using the same technology stack across all of a web application's multiple layers offers several benefits, in practice, it's common for different layers to use different architectures to solve their unique problems.

In terms of the actual web servers, you have a choice between traditional web services such as Apache and Tomcat (or JBoss) and the newer web servers such as NGINX, which I discuss later in this chapter. In terms of increasing horizontal scalability, the

real concern here isn't the language framework or the type of web server. It really is a question of whether you've ensured that the front-end web servers are truly stateless.

Using Caching to scale the front-end

Caching helps you scale by reducing the workload of web services and databases in the backend. There are multiple ways in which you can manage caching at the front end of your web applications:

- Using a Proxy server: As you'll learn shortly, you can use a Content Delivery Network (CDN) to cache web content. However, CDNs aren't ideal for caching entire web pages. You can use a reverse proxy server such as NGINX (or Varnish) to better control both the type and the duration of the cached web content.
- Caching in the web browser (HTTP caching): Today's web browsers can store large amounts of data, ranging into multiple megabytes in size. Modern web applications such as Single Page Applications (SPAs) can access the local browser storage from the application's (JavaScript) code, thus reducing the workload of the web servers. You normally implement HTTP caching in the front-end layer and in the backend web services layer in a similar fashion.
- Using an object cache: As I explained earlier in this chapter, an object cache allows you to store fragments of a web server's responses. Both Redis and Memcached are very powerful shared distributed object caches that can help you scale applications to millions of users through clusters of caching servers.

Later sections in this chapter will explain the concepts of a proxy server, browser caching, and object caching in more detail.

Scaling the Web Services

You can scale your web services using multiple strategies such as making the servers that are hosting the web services stateless, caching the service's HTTP responses and through functional partitioning. The following sections explain each of these strategies.

Making the Servers Stateless

As with the rest of your architecture, keeping state out of the servers hosting your web services plays a significant role in scaling your web services. You can keep state off the servers by letting it store just auto expiring caches and by storing all other persistent data representing the application state in external data stores, caches and message queues. Keeping the servers stateless will let you employ load balancers to distribute client work over all the servers hosting web services, and allows you to take servers out of the load balancing pool when they crash. You can add and remove servers, as well as perform upgrades transparently without affecting your users. Scal-

ing is as simple as just adding more web service hosts to the load balancer pool to handle more connections.

Using HTTP Protocol Caching

A REST web service's GET responses are cacheable, but in order to make the most of this, you must ensure that all the GET method handles are read-only, since it means that issuing GET requests would leave the state of the web service unaltered. If all GET handles are strictly read-only in nature, you can add caching proxies in front of the web services to handle most of the incoming web traffic.

In order to implement HTTP caching at the web service layer to promote scaling, you add a layer of reverse proxies between the web service clients such as the front–end servers and the web and your web services. All requests will now pass through a reverse proxy. This layer of reverse proxy HTTP cache will distribute requests among multiple web service servers.

Patterns such as using local object caches would make it difficult to cache the GET responses, since the web servers can end up with different versions of the data. You can make the resources public to ensure that you only have a single cached object per URL and thus enhance the efficiency of HTTP caching, and make it truly lower the workload of the web services. Making the GET handles public lets you reuse the objects easily.

Using Microservices to Enhance Scaling

You can break up a large web service into multiple independent microservices, with each component addressing a specific functionality. For example, you can create microservices such as a ProductCatalogService and a RecommendationService and other similar services, each addressing a separate subsystem, in order to break up the work that was being handled by a single web service.

Web services help web applications scale since the web services can use different technologies and you can scale each of the web services independently, by adding more web servers to host a specific web service, for example. APIs also help you scale because you can break up the web services layer into much smaller microservices.

Creating multiple, independent microservice based subsystems will distribute the total work among more databases and other services, which enhances scaling. Since different microservices can have different usage patterns, with some microservices needing to perform more read requests, and others with mostly write requests, for example, you can scale each of the services independently. For example, some services may require different types of databases or a different way of caching the content – you can employ the "best of breed" strategy for each of the microservices.

Making Effective use of Third-party services

Organizations with small to medium size operations can significantly enhance their competitive situation by using services offered by third-party entities instead of building everything from scratch, besides having to devote considerable expense and effort to maintain the services. Three of the most useful such services are:

- Content Delivery Networks (CDNs)
- Site analytics
- Client logging

Let me briefly describe how outsourcing each of these services helps you develop more effective web services and SPAs.

Site Analytics

As with traditional websites, web services and SPAs can benefit from using analytic services such as Google Analytics and New Relic to help them understand the usage patterns of their web sites and unearth performance bottlenecks. SPAs can be enhanced in two different ways to take advantage of a service such as Google Analytics:

- Use Google Events to track the hashtag changes: Google Events uses parameters passed along at the end of requests to process the information about that event. Developers can organize and track various events. For example, if the following call shows up in the reports for an SPA it tells you that a chat event has occurred and that the user sent a message on the game page.

```
_trackEvent( 'chat', 'message', 'game' );
```

Let's see how Google Analytics can help you optimize your website performance. You can use a Google Analytics report to correlate the traffic statistics for your top 10 web pages with the average time it takes to render the web pages. This helps you create value based rankings of the web pages such as the following:

```
Value Rank    Page                Average Seconds (Respnse Time)      Requests per Hour
1               /                                0.33                        1000000
2               /mypage1                         0.99                         200000
3               /mypage2                         0.60                          95000
…
```

Ranking the web pages as shown here lets you quickly estimate the potential bang for the back, so to speak. It lets you see, for example that improving their home page load times by just 5 ms offers vastly greater improvement in overall performance as compared to increasing the performance of the web pages mypage1 or mypage2 by 10 ms or 20 ms. Web optimization, as with other optimizations, requires the calculation of

the costs and benefits of the optimization efforts, and data such as the simple set of metrics shown here are the best way to prioritize your efforts.

Server-Side Google Analytics: You can also track the server side with something like Node.js, to see the types of data being requested from the server. This will help during the troubleshooting of slow server requests. However, tracking the server side isn't as useful as tracking the client-side actions.

CLIENT LOGGING

In single page applications (SPAs), which are increasingly becoming the norm for web applications, unlike in traditional websites, when a client hits an error on the server, the error isn't recorded in a log file. While you can write code to track all errors, it's better to use a third-party service to collect the client errors. You of course, can add your own enhancements or tracking features, even with a third-party service in place.

There are many useful third-party services that collect and aggregate errors and metrics generated by new applications. All client logging services work the same underneath: they catch errors with an event handler (window.onerror) and trap the errors by surrounding the code with a try/catch block.

The company NewRelic happens to be the most popular of these services, and is the current standard for monitoring web application performance. A service such as NewRelic provides considerable insight into the performance of the server as well the clients. NewRelic provides not only error logging, but also performance metrics for each step of the request and response cycle, so you can understand if the application response is slow because the database processing is slow, or if the browser isn't rendering the CSS styles fast enough.

Besides NewRelic, you can also check out the error logging services such as Errorception (for JavaScript error logging), Bugsense (good for mobile apps) and Airbrake (specializes in Ruby on Rail applications).

CONTENT DELIVERY NETWORKS (CDNS)

You've already learned about CDNs earlier in this chapter. CDNs provide scalability for static content. In a SPA context, where Node.js is commonly used as the web server, you're at a disadvantage when serving static content, especially large static content files that contain images, CSS and JavaScript since they can't take advantage of the asynchronous nature of Node.js. Apache Web server does a far better job with the delivery of such files due to its pre-fork capability.

In our next section, let's review application scalability, which is the heart of most if not all modern enhancements in web application technologies.

Working with Web Servers

Organizations of every size use web servers, some to serve simple web sites and others to support heavy duty work by playing the role of backend web servers. There are several web servers out there, all open source products, of which the most popular ones today are the Apache Web Server and NGINX, with the latter still with substantially smaller usage numbers than the former, but whose usage is growing at a much faster rate. Let's take a quick look at these two popular web servers and the uses for which they're best suited.

Working with the Apache Web Server

The Apache Web Server (Apache from here on) is the most well-known of all web servers. It's important to understand that although it's quite easy to get started with Apache out of the box, it's not optimized for using CPU and RAM efficiently. Apache is quite bulky, with several modules that provide various chunks of functionality.

While Apache does an admirable job in serving dynamic sites (Linux + Apache + MySQL + PHP constitute the well-known application framework known as LAMP), it's really not ideal for sites that just want to serve a lot of static files at high speed. You can remove most of the modules and streamline Apache for these types of sites, but why do so when you can use a web server that's been explicitly designed to do that type of work (serve static files at high speeds)? In this type of scenario, NGINX might be the better choice.

The NGINX Web Server

While the Apache web server is still the most common web server used in the world, there are questions as to its scalability and performance when dealing with modern website architectures. The NGINX (pronounced "Engine-X) web server, an open source web server introduced in 2004, is fast becoming the go to web server for providing high performance and high concurrency, while using a low amount of memory. Right now, NGINX is the number two web server on the internet, behind the Apache web server, and moving up very fast.

Note that NGINX isn't just a high performance HTTP server – it's also a reverse proxy, as well as an IMAP/POP3 proxy server.

Benefits Offered by Nginx

The NGINX web server or hardware reverse proxy both offer great functionality and performance. NGINX offers the following properties which are highly useful:

- Nginx can work as a load balancer and contains advanced features such as WebSockets and throttling.

- You can also configure NGINX to override HTTP headers, which lets you apply HTTP caching to applications that don't implement caching headers correctly (or even use it to override configured caching policies).
- Since Nginx is also FastCGI server, you can run web applications on the same web server stack as the reverse proxies.
- Nginx is highly efficient since it uses asynchronous processing always: this allows the server to proxy tens of thousands of simultaneous web connections with an extremely minimal connection overhead.

Nginx is getting more popular day by day when compared to other web servers, and its' share of the internet uses is growing at a fast clip. Netcraft reported that in September 2015, while Apache and Microsoft both lost market share, NIGIX grew its share of total web sites on the Internet to about 15%. This is a truly phenomenal number for a web server as new as NGINX.

Why High Concurrency is Important

Web sites aren't any longer the simple affairs they used to be a few years ago – today's web applications need to be up 24/7, and in addition to e-commerce, provide entertainment (movies, gaming etc) and various types of live information to users.

Concurrency has always been a challenge when architecting these complex web applications. Increasing social web and mobile usage means that there are a large number of simultaneously connected users using a web application. On top of this, modern web browsers enable you to open multiple simultaneous connections to the same website to enhance the page loading speed.

Clients often stay connected to web sites to reduce the hassles of having to open new connections. The web server must therefore account for enough memory to support a large number of live connections. While faster disks, more powerful CPUs, and more memory certainly do help in providing higher concurrency levels, the choking point for high demand web applications is going to be the web server, which should be able to scale nonlinearly as the number of simultaneous connections and requests per second creeps higher and higher.

The Apache web server, under its traditional architecture, doesn't allow a website to scale nonlinearly. Each web page request leads to Apache spawning a new process (or thread). Thus, to scale, Apache spawns a copy of itself for every new connection, making it a platform not easily amenable to scaling, due to the high memory and CPU demanded by its architecture.

In 2003, Daniel Kegel came up with the C10K manifesto. The C10K manifesto was a call to architect new types of web servers that can handle ten thousand simultaneous connections. The NGINX server is a direct outcome of the attempts to answer the call of the C10K manifesto, and certainly meets the challenge posed by Kegel.

The NGINX server was architected with scalability as its main focus. NGINX is event based, so it doesn't spawn new processes/threads for each new web page request. As the memory and CPU usage doesn't increase linearly with workloads, the web server can easily support tens of thousands of concurrent connections.

I might add here that Apache has also enhanced its capabilities with the release of the Apache 2.4x branch. New multiprocessing core modules and new proxy modules designed to enhance scaling, performance and resource utilization have been added to compete better with event-driven web servers such as NGINX.

Apache and NGINX have their relative strengths, so it's not really one or to the other in most cases. Since NGINX is great at serving static content, you can deploy it in the front end to serve pre-compressed static data such as HTML files, image files and CSS files, etc. You many even take advantage of its load balancing capabilities with its easy to set up reverse proxy module. The reverse proxy module lets NGINX redirect requests for specific MIME types of a web site to a different server. At the same time, you can deploy one or more Apache web servers for delivering dynamic content such as PHP pages. The reverse proxy module will redirect the request for dynamic data to the Apache server.

Caching Proxies and Reverse Proxying

Web proxy caching is the storing of copies of frequently used web objects such as documents and images, and serve this information to the user. The goal is to speed up service to the clients, while saving on the Internet bandwidth usage. There are two types of proxies – caching proxies and reverse proxies, as explained in the following sections.

Caching Proxies

A caching proxy is a server that caches HTTP content. Local caching proxy servers sit in the network, in between your web servers and the clients and all HTTP traffic is routed through the caching servers so they can cache as many web requests as possible. Caching proxies use what's called a read-through cache to store web responses which can be reused by sending them to multiple servers instead of generating new web traffic for each user. More than corporate networks, it's Internet Service Providers (ISPs) that install caching proxies to reduce the web traffic generated by users.

Local proxy servers are much less popular these days since network bandwidth has become way cheaper over time. Also, most websites use the Secure Sockets Layer (SSL) exclusively to serve content. Since SSL communications are encrypted, caching proxies can't intercept those requests, since they can't encrypt and decrypt messages without the required security certificates.

Reverse Proxies

A reverse proxy doesn't reverse anything – it does the something as a caching proxy! These types of proxy servers are called reverse proxy servers because of their location – you place them inside your data centers to cache the response of your web servers and thus reduce their workload. Reverse proxies help you scale in many ways:

- You can employ a bank of reverse proxy servers since each proxy is independent of the others, thus promoting your application's scalability
- They can reduce the requests reaching the web servers, if you are able to cache full pages
- A reverse proxy helps you control both the request types to be cached and the duration for which the requests are cached.
- By placing a bank of reverse proxy servers between the front end web servers and the web servers hosting the web services, you can make the web service layer much faster by caching the web service responses.

As explained elsewhere in this Chapter (in the section on the NGINX web server), NGINX has several features which make it an excellent proxy server. If you've chosen to use a hardware load balancer in your data center, you can make it perform double duty by letting it act as a reverse proxy server as well. If not, an open-source reverse proxy technology solution such as NGINX, Varnish or Apache Traffic Server will work very well. You can scale any of these by simply adding more instances.

High availability and Keepalived

High availability means that an application restarts or reroutes work to another system automatically when the first server encounters a failure. In order to setup a highly available system, the system must be able to redirect the workload, and there also must be a mechanism to monitor failures and automatically transition the system to a healthy server when service interruptions are sensed.

Keepalived is a Linux –based routing software that lets you achieve high availability. Keepalived assigns multiple nodes to a virtual IP and monitors those nodes, and fails over when a node goes down. Keepalived uses the VRRP (Virtual Router Redundancy Protocol) protocol, for supporting high availability.

VRRP is a network protocol that automatically assigns available IP routers to participating hosts. This lets you increase the availability of routing paths through automatic default gateway selections on an IP subnetwork. VRRP allows Keepalived to perform router failover and provide resilient infrastructures. The VRRP protocol ensures that one of the nodes in the web server infrastructure is the master, and assigns it a higher priority than the other nodes. The others are deemed backup nodes, and listen for multicast packets from the master.

If the backup nodes don't receive any VRRP advertisements for a specific interval, one of the backup nodes takes over as the master and assigns the configured IP to itself. If there are multiple backup nodes with the same priority, the node with the highest IP wins the election. You can set up a highly available web service with Keepalived. Basically what you're doing is configuring a floating IP address that's automatically moved between web servers. If the primary web server goes down, the floating IP is automatically assigned to the second server, allowing the service to resume. Keepalived is often used together with HAProxy to provide redundancy, since HAProxy doesn't come with its own redundancy capabilities.

Handling Data Storage with Databases

Databases are ubiquitous in any IT environment and are used for storing various types of information. There are several types of databases, based on the type of information they store as well as the types of data retrieval they allow, as explained in the following sections. Most organizations today use multiple types of databases, with a mix of relational and NoSQL databases.

Relational databases

A relational database management system (RDBMS) is the most well-known type of database, and still remains the predominant type of database in use, even with the advent of several newer types of databases. Most applications such as shopping, sales, and human resources are handled best by a relational database.

Relational databases store information in the form of relationships among various entities (such as employees and managers), and are extremely efficient for querying and storing transactional data. Oracle, IBM's DB2 and Microsoft SQL Server are the leading commercial relational databases and MySQL, PostGreSQL, MariaDB (very similar to the MySQL database) are the leading open source relational databases.

Commercial relational databases such as Oracle require full-fledged administrators owing to their complexity. On the other hand, it's not uncommon for organizations to look to the Linux system administrator to help out with the administration of open source databases such as MySQL and PostGres (and especially the newer NoSQL databases such as MongoDB, CouchDB and Cassandra), and perform tasks such as installing, backing up and recovering the database.

Other Types of Databases

In today's world, while relational databases are still important, there are several new types of databases that are increasingly becoming important, as explained in the following sections.

NoSQL Databases

NoSQL databases are useful for handling unstructured data and are especially good at handling large quantities of such data. In this chapter, I discuss MongoDB and Cassandra, two very popular NoSQL databases that you ought to be familiar with. Later chapters discuss both of these databases in more detail. In addition, there are several other popular NoSQL databases such as CouchDB.

Caching Databases

Memcached and Redis are two well-known caching databases. I explain the role of both of these data stores in this chapter, and discuss them in more detail in Chapter 9.

Cloud Databases

Cloud databases are hosted by cloud providers such as Amazon and Google for their cloud customers. Amazon offers Amazon DynamoDB, a powerful NoSQL database, and Google, the Google Cloud Storage and Google Cloud Datastore. Chapter 9 explains Amazon's DynamoDB and RDS Datastore. Cloud databases take the complexities of database management out of an organization's hands, but they've their own headaches, such as security issues, for example, since the data is hosted by the cloud provider.

In addition to the types of databases listed here, there are many other less well known database types, such as object databases (for example, db4o), and NewSQL databases such as VoltDB and Clustrix, as well.

It's important to understand that most organization today use several types of databases. This happens to be true not only for a large environment, where you expect many databases of many different types, but even for smaller organizations, especially those that deal with web applications.

MongoDB as a Backend Database

MongoDB is a highly scalable open source NoSQL database that provides high performance for document-oriented storage. The following sections explain the salient features of MongoDB.

A Document Database

Document-oriented storage means that instead of using traditional rows and columns like a relational database, MongoDB stores data in the JSON document format. Instead of tables, you have collections, and documents play the role of rows.

MongoDB uses collections to store all documents. Although collections are called the counterparts of relational database tables, they are defined entirely differently since

there's no concept of a schema in MongoDB. A collection is simply a group of documents that share common indexes.

Dynamic Schemas

Relational databases are all about schemas – the data you store in the database must conform to an existing schema, which describes the type of data you can store, such as characters and integers, and their length. MongoDB doesn't use schemas – you can store any type of JSON document within any collection. This means that a collection can have disparate types of documents in it, each with a different structure. You can also change the document structure by updating it – no problems there.

Automatic Scaling

MongoDB lets you scale horizontally with commodity servers, instead of having to go the more expensive route of vertical scaling, as is the case with a relational database. Although you can create clusters of relational databases, that's mostly for high availability, and not for enhancing performance. Scaling with multiple servers in a MongoDB database, on the other hand, is expressly designed for enhanced performance. There are two aspects to the automatic (or horizontal) scaling:

- Automatic sharding: helps distribute data across a cluster of servers
- Replica sets: provide support for low-latency high throughput deployments

High Performance

MongoDB supports high performance data storage, by supporting embedded data models that reduce I/O activity. It also supports indexing to support faster queries – you can include keys from embedded documents and even arrays.

High Availability

MongoDB uses a replication facility called replica sets, to provide both automatic failover and data redundancy. A replica set is defined as a set of MongoDB servers that store replicas of the same data set, enhancing both redundancy and data availability.

```
For Single Page Applications(SPAs), which I discuss elsewhere in this chapter, MongoDB's document-
```

MongoDB and the CAP Theorem

As with other document databases and other databases that fall under the broad umbrella of NoSQL Databases, MongoDB doesn't offer support for the well-known ACID properties provided by traditional relational databases. ACID properties are the hallmark of all modern transactional databases (such as MySQL and Oracle), and refer to the following set of principles expected of a database:

- Atomicity: This is the all-or-none principle which requires that if even one element of a transaction fails, the entire transaction fails. The transaction succeeds only if all of its tasks are performed successfully.
- Consistency: This property ensures that transactions are always fully completed, by requiring that the database must be in a consistent state both at the beginning and at the end of a transaction.
- Isolation: Each transaction is considered independent of the other transactions. No transaction has access to any other transaction that's in an unfinished state.
- Durability: Once a transaction completes, it's "permanent". That is, the transaction is recorded to persistent storage and will survive a system breakdown such as a power or disk failure.

While the ACID requirements served traditional relational databases just fine for many years, the advent and eventual popularity of non-relational data such as unstructured data, non-relational data, and the proliferation of distributed computing systems led to new views about the required transaction properties that modern databases must satisfy. The Consistency, Availability and Partition Tolerance (CAP) theorem sought to refine the requirements to be met for implementing applications in modern distributed computing systems. The CAP theorem actually stands for the following three principles:

- Consistency: this is the same as the ACID consistency property. Satisfying this requirement means that all the nodes in a cluster see all the data they're supposed to be able to view.
- Availability: the system must be available when requested.
- Partition Tolerance: failure of a single node in a distributed system mustn't lead to the failure of the entire system. The system must be available even if there's some data loss or a partial system failure.

The problem is, at any given time, a distributed system can usually support only two out of the three requirements listed here. This means that tradeoffs are almost always inevitable when using distributed data stores such as the popular NoSQL databases.

For database reliability purposes, meeting the Availability and Partition Tolerance requirements is absolutely essential, of course. That means that the consistency requirement is often at risk. However, leading NoSQL databases such as Cassandra and Amazon's DynamoDB deal with the loss of consistency just fine. How so? This is possible due the adoption by these databases of something called the BASE system, which is a modified set of ACID requirements to fit modern NoSQL and related non-relational databases. Here's what BASE stands for:

- Basically Available: the system guarantees the availability of data in the sense that it'll respond to any request. However, the response could be a "failure" to obtain the request data set, or the data set returned may be in an inconsistent or changing state.
- Soft: the state of the system is always "soft, in the sense that the "eventual consistency" (the final requirement) may be causing changes in the system state at any given time
- Eventually Consistent: The system will eventually become consistent once it stops receiving new data inputs. As long as the system is receiving inputs, it doesn't check for the consistency of each transaction before it moves to the next transaction.

Amazon's DynamoDB, for example, which lies behind Amazon's shopping cart technology, stresses high availability, meaning it can afford to go easy on the consistency angle. These types of databases eschew the complex queries necessary to support consistency in the traditional sense, settling instead for the eventual consistency goal. Eventual consistency in this context means that in a distributed system, not all nodes see the same version of the data – at any given time the state may diverge between nodes – that is, it's possible for some nodes to serve stale data. However, given sufficient time, the state will come to be the same across the system.

MongoDB, a popular NoSQL database, on the other hand, favors consistency and partition tolerance over high availability.

The main point to take away from this discussion of CAP and BASE is that while NoSQL databases have their advantages, particularly in the way they support horizontal scaling and the efficient processing of non-relational data, they do come with unique drawbacks and involve crucial sacrifices in terms of simultaneous support for traditional principles such as data consistency and availability.

Consistency, while it's a laudable objective that a database can satisfy, has a negative impact on cost effective horizontal scaling. If the database needs to check the consistency of every transaction continuously, a database with billions of transactions will incur a significant cost to perform all the checks.

It's the principle of eventual consistency that has allowed Google, Twitter and Amazon, among others, to continuously interact with millions of their global customers, keeping their systems available and supporting partition tolerance. Without the principle of eventual consistency, there wouldn't be all these systems today that deal successfully with the exponential rise of data volumes due to cloud computing, social networking and related modern trends.

Caching

A cache is simply a set of data that you store for future use. Caching is a key technology that web applications use to increase both performance and scalability. Caching can play a significant role in scaling your applications, since you need fewer computing resources to serve an ever growing customer base. Caching is a common technique that's used at almost every level of the application stack, including operating systems, databases, HTTP browser caches, HTTP proxies and reverse proxies, as well as application object caches.

Not everything can be cached – and should be cached! Good candidates for caching are objects that won't become invalidated with the passage of a short period of time. If users are frequently updating the data, it's not feasible to cache that data as the data becomes quickly invalidated.

Regardless of the specific caching strategy you employ, application code that seeks to reuse cached objects for multiple requests or several users is always going to make the responses faster, besides saving resources such as server CPU/RAM/, and network bandwidth. Even a design that only caches just page fragments instead of entire web pages will provide terrific performance benefits.

For web applications, you need to be concerned about two main types of caches – HTTP-based caches and custom object caches.

HTTP-Based Caching (Browser caching)

In a web application stack, the HTTP-based cache is the most common and predominant component. In order to understand HTTP based caching, it's important to first understand something about the all-important HTTP caching headers. HTTP caching is common, and often a web request makes use of several HTTP caches linked to each other.

HTTP caches are read-through caches. In a read-through caching architecture, clients connect to the cache first to request resources such as web pages or CSS files. The cache will return a resource to the client directly if it's in the cache, and if it's not there, contacts the originating server and fetches the data for the client. Read-through caches are transparent to the client, who's unaware as to where the resource is being sent from, and they can't distinguish between a cached object and an object they get by connecting directly to the service.

Letting the browser cache the content dramatically increases performance in many cases. You are storing the files on the client machine itself in this case. Sometimes the client server may even store previously rendered web pages, completely bypassing the network connection to the server (this is the reason why when you press the "back" arrow, the previous page appears so quickly on your screen!) and HTTP caches help

lower the load on you data center and move it to external servers that are located near the users. Not only that, but caching also results in faster responses to user requests.

HTTP headers not only help cache web pages and static resources, but also the responses of web services. You can insert an HTTP cache between the web services and client. In fact, REST based web services excel at caching web service responses.

How HTTP Caching is Used

At one extreme, you can cache responses indefinitely. Static content such as images and CSS usually is cached "forever". You can cache static content such as this forever and use new URLs for newer versions of the static files, thus ensuring that users are always using the correct (that is, compatible) HTML, CSS and JavaScript files. On the other extreme, you can ensure that a HTTP response isn't ever cached, by using a HTTP header such as Cache-Control: no-cache.

Types of HTTP Caching

You can configure two types of HTTP caches, as explained here:

- Browser cache: All web browsers use a browser cache based on disk storage as well as memory, to avid resending HTTP requests for resources that have already been cached. This caching helps load web pages faster and keeps resource usage low.
- Caching proxies: Caching proxies (also called web proxies) are servers that sit between the internet and the users (that is, the web browsers) and use a read-through cache to reduce the web traffic generated by the network's clientele.

A Web Cache is also called a proxy server, and is an entity that satisfies HTTP requests on behalf of an origin Web server. Rather than each browser connecting directly to the internet, they relay their requests through the proxy. The proxy server is the only one that initiates internet connections. The clients connect to the proxy server, which'll forward the requests to the target web server only if it doesn't find the content in its cache.

Both ISPs (Internet Service Providers) and corporate networks (local proxy servers) can use a caching proxy, with ISPs seeking to maximize the caching of web requests to lower the web traffic generated by users.

Local proxy servers have one big advantage over the ISP proxy servers - when you use SSL, the external caching servers can't intercept the requests since they lack the required security certificates to read the messages.

Good proxy servers can dramatically reduce the network bandwidth usage in an organization, as well as improve performance – so you get to run things faster, but at a lower cost! Web caches can significantly cut down on the Web traffic, keeping you from having to upgrade your bandwidth, and improving application performance. Usually, an ISP purchases and installs the Web Cache. A company for example, may install a Web cache and configure all of its user browsers to point to the Web cache, so as to redirect all the user HTTP requests from the origin servers to the Web Cache.

Content Delivery Networks (CDN) are functionally equivalent to the caching proxies, since they also rely on HTTP headers for caching – the big difference is that the CDN service provider manages all the caching.

- Reverse proxies: These are servers that work quite similar to caching proxies, with the difference being that the caching servers are located within the data center itself to reduce the load on the web servers. The reverse proxy server intercepts web traffic from clients connecting to a site and forwards the requests to the web server if they can't serve it directly from their own cache. Reverse proxies are great for overriding HTTP headers and manage request caching easily. They also help you scale REST based web services by sitting between the front-end and the web-service servers, lowering the amount of requests that your web service web servers need to serve.

Benefits of a Web Cache

Web caches help you in two ways; they reduce the response time for client requests and also reduce the amount of traffic e flowing through your company's access link to the Internet. Let's' use some hypothetical numbers to breakdown the total response time of a web request to understand how caching can help you big time. Total response time is the time it takes to satisfy a browser's request for an object, and is composed of the following three types of delays:

- LAN delay
- Access delay
- Internet delay

Let's see how a Webcache can make a difference in keep Web response times very low. Let's assume the following:

- Your company's network is a high speed LAN and a router in the Internet connects to a router in the Internet via a 15 Mbps link.

- The average size of an object is 1 Mbits.
- On average, about 15 requests are made by your company's user browses to the origin Web servers (no Webcache at this point)
- The HTTP messages are on the average quite small, and therefore, there's negligible traffic between your router and the Internet router.
- The Internet side router takes on average 2 seconds to forward an HTTP request and receive the response – this is called the Internet delay.

You can measure the LAN delay by looking at the traffic density on the LAN:

```
(15 requests/second) * (1 Mbits/request)/(100 Mbps) = 0.15
```

A low traffic intensity such as 0.15 means that you can ignore the LAN delay, which will be in tens of milliseconds duration at most.

The access delay however is significant, as the traffic density on the access link which is from your own router to the Internet router, is:

```
(15 requests/second) * (1 Mbits/request)/(15 Mbps)=1
```

An access delay of 1 is bad – it means that the access delays are large and grow without bound. You're going to end up with an average response time for HTTP requests that'll be several minutes long, which is something you just can't have.

One way to bring down the total response time is to upgrade your access link from 15 Mbps to 100 Mbps (quite expensive), which will lower the traffic intensity to 0.15. Thus means that the access delay will be insignificant, just as the LAN delay is, meaning that the response time will be comprised of just the Internet delay, which is 2 seconds.

The other alternative doesn't involve an expensive upgrade of your access links – you just install a Web cache in your network! You can use open source software and commodity software to do this.

Let's' say the hit ratio of the cache is 0.4. That means 40% of all web requests are satisfied directly from the cache and only 60% of the requests are sent to the origin servers. Since the access servers now handle only 60% of the traffic as compared to its earlier traffic volume, the traffic intensity on the access link goes down to 0.6 from its previous value of 1.0. On a 15 Mbps link, a traffic intensity that's less than 0.8 means the access delay is negligible (in tens of milliseconds).

The average response time now is:

```
0.4 * (0.01 seconds) + 0.6 * (2.00 seconds) = 1.2 seconds (approx.)
```

As you can see, a Web Cache leads to dramatically low response times, even when compared to a solution that requires a faster (and much more expensive) access link.

There really isn't much for you to do if you farm out HTTP caching to a CDN. However, since reverse proxy servers are run from your own data center, you need to manage them yourself. A hardware load balancer, as mentioned earlier, can also provide reverse proxying services. You can also use an open source solution such as NGINX or Varnish. NGINX for example is famous for handling tens of thousands of requests every second per each instance. The essential keys to effective management of proxying then are the type and size of the cached responses and the length of time for which you want to cache them.

A Content Distribution Network (CDN) uses distributed web caches throughout the Internet, helping localize the web traffic. You can use a shared CDN such as Akamai and Limelight to speed up the response times of web requests.

Caching Objects

Caching objects is another key way to boost web application performance. Object caching falls under the category of application caching, where the application developers assume the responsibility for the content to be cached, as well as the time for which the content must be cached.

Caching stores such as Memcached let you set expiry times for web documents and also let you update Memcached when new web pages are added to your site. For example, if you store a web page in Memcached and that content remains valid for one minute, all your uses will be sent data directly from Memcached and your web servers won't have to handle that load. The web servers simply generate a new page every minute and users get that new page when they refresh their browser. Since only the application developer knows the application well enough to determine the content and duration of the cached objects, object caching isn't something that's as simple as browser caching.

A cache is a server or a service geared towards reducing both the latency and the resources involved in generating responses by serving previously generated and stored content. Caching is a critical technique for scaling an infrastructure.

```
Caching application objects is done in a different way than HTTP content caching. Applications exp
```

As with HTTP caches, there are several types of object caches: client-side caches, local caches and distributed object caches. The first two are simple affairs, as described here:

- Client-side Caches: Today's web browsers, which all use JavaScript, can store application data such as user preferences, for example, on a user's laptop or on mobile devices. Caching data on the user's devices of course makes web applications run faster and also lowers the load on your servers. JavaScript running in the browser can retrieve the cached objects from the cache. Note that the users can always wipe the cache clean. Single-page applications (SPAs) that we learned about earlier in this chapter, benefit from the client-side cache since they run a lot of code within the browser, and also rely on asynchronous web requests (AJAX). This is especially true for SPAs explicitly designed for mobile devices.

Caching objects in local memory lets applications access resources extremely fast, since there's no network latency.

- Local Caches: A local cache is located on the web servers. There are a few different ways to cache objects locally, but all of them employ the same strategy: store the objects on the same server where the application code is running. One way is for the front end and back end applications to use local caches by caching application objects in a pool. The application incurs virtually no cost when accessing cached objects since they're stored right in the application's process memory. Some languages such as PHP (but not Java, which is multithreaded) can also use shared memory segments that allow multiple processes to access the stored objects in the cache. Alternately, you can deploy a local caching server with each web server.

Memcached and Redis are two very popular caching servers and you'll learn more about them in the next section. When you're just starting out, or if you have to manage a small web application, you can run both the webserver and the caching server on the same machine to cut costs and make things run faster since the network roundtrips are very short.

Local Caching

Local caching, as you guessed, doesn't involve an external caching database. Local caching, regardless of how you want to implement it in practice, is very easy to set up and doesn't involve too much complexity. They're also very low latency solutions and don't involve any issues with locking etc. however, there's no synchronization between the local application caches when you employ a bunch of them.

Lack of synchronization among the caches on different servers means that you run the risk of inconsistent data because of different cached objects representing different

values for the same information. The application servers don't coordinate the storing of the cached objects, leading to redundancy and duplication of the objects stored in the cache. Fortunately, there's a caching solution that addresses all of these issues with local object caches – a distributed object cache, which I discuss next.

Distributed Object Caching – Memcached and Redis

A distributed object cache works exactly the same as a local object cache – both use a simple key-value store where clients can store objects for a set duration of time. The big difference is that unlike a local cache, a distributed cache is remotely hosted. Redis and Memcached are popular open source solutions that serve a wide variety of uses. Memcached is a fast key/value store that's very simple and lacks too many bells and whistles, making it very easy to implement. It's used in several of the busiest web sites in the world today.

Distributed caching servers are easy to work with through any programming language. Here's an example showing a caching interface in PHP:

```
$m = new Memcached();
$m->addServer('10.0.0.1', 11211);
$m->set('UserCount', 123, 600);
```

This code does the following:

- Sets Memcached as the caching server
- Sets the IP for the cache server
- Sets the caching data: the name of the object to be cached, its key value and the TTL (duration for which it'll stay in the cache) for that object

A cache server such as Memcached or Redis is really a database, and thus offers most of the capabilities of a key-value store, such as replication, query optimization and efficient use of memory.

Caching servers aren't an alternative to the regular databases – all you're doing is when you generate new database content from a relational database for example, you store the content in Memcached or Redis for future use. For subsequent requests for that data, you first check the cache and if the data is there, you send the content back without having to call the database.

While caching is easy to setup and manage, there are situations where it isn't ideal. Transactions constantly modify, or add and delete the data in the database tables. If the user must absolutely, positively get the very latest version of the data, caching becomes problematic, as it tends to lag behind the current state of the database.

If you're using Amazon AWS to host your applications you can use Amazon Elastic Cache, which is Amazon's hosted cache cluster that uses Memcached or Redis (you get to pick), or set up you own caching service on your EC2 instances.

You've learned the basics of caching in this section, and how it helps you scale as well as make your applications run faster. While there are several types of caching, not all caches are equal, however! The earlier in the request process for a web page or data an application satisfies the request by retrieving it from a cache, the more beneficial is the cache.

Caching servers are typically deployed on dedicated machines, and you usually deploy them in clusters, with multiple caching servers. You can implement replication or data partitioning to scale, once you exhaust the limits of adding more memory to the caching servers. For example, if you're using Memcached as your object cache, you can partition the data among multiple Memcached servers in order to scale up.

When formulating a caching strategy it's important to remember that you can't (and don't want to) cache everything. You must choose which web pages or services you need to cache. To do that, use some type of metrics. Refer to the metrics example I provided earlier in this chapter (page 11), which showed how third-party services such as Google Analytics can help you maximize the potential gains from caching.

Asynchronous Processing, Messaging Applications and MOM

In traditional software execution, synchronous processing is the standard way to perform operations. A caller such as a function, a thread, or a process sending a request to another process, or an application making requests to a remote server won't proceed to the next step until it gets the response for its request. In other words, the caller waits for a response or responses before continuing further execution. Under an asynchronous processing model, however, a caller doesn't wait for the responses from the services it contacts. It sends requests and continues processing without ever being blocked.

Let me take a simple example here to demonstrate the difference between the synchronous and asynchronous approaches. Let's say an application's code sends out an e-mail to a user. Under the synchronous processing model, the code needs to wait for the e-mail service to do what it takes to send that email out: resolve the IP addresses, establish the network connection, and finally send the email to an SMTP server. The email service needs to also encode and transfer the message and its attachments, all of which takes some time (a few seconds). During this time, the execution of the appli-

cation code pauses, a pause that's often referred to as blocking, since the code is waiting on an external operation to complete.

Platforms such as Node.js come with built in asynchronous processing capabilities, making messaging and message brokers less useful than for other platforms such as Java and PHP, which aren't asynchronous.

Obviously, the synchronous processing model isn't conducive to building responsive applications: not only is this a slow method but it's also highly resource intensive, since blocked threads continue to consume resources even when they're just waiting. Total execution time is the sum of the time taken to perform each of the operations in a service, and because your app is doing things serially rather than parallelly, blocking operations slow down your application response times. The benefits that accrue from asynchronous processing are really due to the fact that your applications and services perform work parallelly instead of sequentially, thus bringing into play vastly higher resources to process the workloads. Execution times are rapid and user interest is unlikely to flag under such architectures.

Responsive applications are really what it's all about, as asynchronous processing helps build these types of applications since it doesn't involve blocking operations.

Messages and Message Queues

Essentially, a message is a piece of code written in XML or JSON that contains instructions for performing the asynchronous operation. A message queue (managed by a message broker) allows you to reap the benefits of synchronous processing. Asynchronous processing can be beneficial in a situation where you want to keep clients from waiting for time consuming tasks to complete – ideally, the client should be able to continue their execution with no blocking.

Here are some typical use cases for using asynchronous processing and message queues:

- Any operation that requires a heavy amount of resources such as the generation of heavy reports can be sent to a message queue.
- Anytime you need to perform operations on a remote server that takes some time to complete, an asynchronous model of processing can speed up things.
- Any critical operations such as placing orders or processing payments can't be expected to wait for less critical operations to complete. You can divert the less important parts of the operation to a message queue so they can be processed asynchronously by a separate message consumer.

Components of a Messaging Architecture

Messaging revolves around the following three crucial constituent operations:

- Producing: this simply means the process of sending messages. A producer is a program that creates and sends messages to a message queue. Message producers are often also called message publishers.
- Queuing: a message queue is a location to store messages, and is part of the message broker. Multiple producers can send messages to a queue and multiple consumers can retrieve messages from a queue.
- Consuming: means the same as receiving messages. A message consumer is a program that waits to receive messages sent by a producer. It's the message consumer that actually performs the asynchronous operation for the message producer. It's common for producers, consumers, and queues to reside on different servers, and even use different technologies.

Message queues are the heart of asynchronous processing, so let's learn more about them in the next section.

Message Queues

The message queue stores and distributes asynchronous requests. As the producers create new messages and send them to the queue, the queue arranges those messages in a sequence and sends them along to the consumers. Consumers then will act on the messages, by performing the asynchronous actions.

Message Queues and Asynchronous Processing

Message queues provide the essence of asynchronous processing – non-blocking operations - and therefore, non-blocking I/O. The message producers and message consumers work independently, neither of them blocking the other, nor with either being aware of the other. One of them dedicates itself to creating messages and the other to the processing of those message requests.

Message queues not only enable asynchronous processing, but are useful in evening out spurts in message traffic and in isolating failures. During times of heavy traffic, the message queues continue to accept the high traffic and keep queuing the messages. The front end application may produce a large number of messages which the back end component will consume, but the user isn't affected since the front end servers aren't waiting for the operations to complete.

A key advantage of message queues is that they promote scalability – you can employ banks of message producers and consumers on dedicated servers and just keep adding additional servers to handle a growing workload.

How Message Queues are Implemented

At its simplest, a message queue is just a thread running within the main application process. You can also implement it by storing the messages in the file system or in a relational database such as MySQL.

For heavy duty message processing as well as for tackling on additional features such as high availability on to the message queue, you need a full-fledged specialized messaging application called a message broker or message oriented middleware, which is the topic of our next section.

Message Brokers and Message Oriented Middleware (MOM)

A message broker is a dedicated application that provides not only message queuing, but also the routing and delivery of the messages.

Message brokers relieve you from having to custom write code for providing critical messaging functions – you simply configure the broker to get the functionality you desire.

Message broker software is also often called message-oriented middleware (MOM) or an enterprise service bus (ESB). However, a Message Broker is really a higher level concept that's built on top of MOM (or an ESB), with the MOM providing the underlying services such as message persistence and guaranteed delivery.

A message broker adds others things to a MOM, such as rules based business process integration, content based routing, data transformation engine, etc.

What Message Brokers do

A message broker accepts and forwards messages. Just as a post office accepts your mail and delivers it to whomever you address the mail to, a message broker such as RabbitMQ accepts, stores, and forwards data, with the data in this context being messages.

The message broker performs a critical function in asynchronous processing – it separates the consumers from the producers. One of the great things that message brokers do very well is the fast queuing and dequeing of large volumes of messages, since they're optimized for high throughput.

Messaging Protocols

You can specify how clients connect to the broker and how messages are transmitted, by selecting a correct messaging protocol. Messaging protocols control the transmis-

sion of messages from producers to consumers. I briefly describe the most commonly employed protocols here:

- Streaming Text–Oriented Messaging Protocol (STMP): This is a rudimentary messaging protocol that has low overhead, but has no advanced features such as those offered by other commonly used protocols.
- Advanced Message Queuing Protocol (AMQP): AMQP is a comprehensive messaging protocol that has been well accepted as an industry standard for messaging. The best thing about AMQP is that it's a standardized protocol, and hence is easy to integrate into your software. AMQP offers features such as guaranteed delivery of messages, transactions and many other advanced features.
- Java Messaging Service (JMS): JMS is a powerful messaging standard but it's confined to Java technologies such as Java or Scala.

Of the three messaging protocols listed here, AMQP is the most commonly used protocol among enterprises.

Pull versus Push Messaging

Developers can configure message consumers to work in two different ways: pull or push:

- The pull model, also known as a cron-like model, lets the message consumer pull messages off the message queue. For example, in an Email service, the consumer picks up the messages from the queue and uses SMTP to forward the emails to the mail servers. Applications like PHP and Ruby use a pull model whereby the consumers connect to the queue once in a while and consume the available messages (or a set number of messages). Once they consume the messages, the consumers detach themselves from the queue.
- In the push model, the message consumer is always connected to the message broker, and maintains an idle wait by blocking on the socket read operation. The message broker will push new messages through the permanent connection, as fast as the consumer can handle them. Applications such as Java and Node.js which use persistent application containers to maintain constant connections to the message brokers.

Subscription Methods

Message consumers can be configured to use various subscription methods to pull messages off the message queues maintained by the message broker. Here are the two common subscription methods:

- Publish/subscribe (pub/sub): In the pub/sub subscription method, the broker publishes messages not to a message queue, but to a topic. A topic in this context is something like a channel. Consumers connect to the broker and let it know the topic or topics they want to subscribe to. The broker transmits all messages published to those topics to the consumer that requested them, and the consumers receive the messages in their private queue. Thus, in this mode, there's a dedicated message queue for each consumer, containing just those messages that were published on a specific topic or topics.
- Direct worker queue: Producers send all messages to a single queue, with each message routed to a specific consumer. So, multiple consumers share the single worker queue. Tasks that take considerable time such as sending emails and uploading content to external services to multiple consumers benefit from this method.

Popular Message Brokers

There are several powerful, robust and scalable open-source message brokers and two of the most popular of these are RabbitMQ and ActiveMQ. If you are hosting your operations in the Amazon cloud, you can also use Amazon's Simple Queue Service (SQS), which I discuss in Chapter 9.

Your choice of a message broker solution should be guided by your use cases and on key criteria such as the volume of message, message size, rate of message consumption, concurrent producers/consumers. Two important factors are whether you want to ensure that messages are safeguarded from loss by storing them, and whether you need message acknowledgment.

RabbitMQ is an open source message broker written in the Erlang programming language, and is currently the leading open source messaging broker.

One of the best things that sophisticated message brokers such as RabbitMQ and ActiveMQ offer is their ability to create custom routes. Routes determine which messages are sent to a queue. Earlier, you learned about the publish/subscribe and direct worker queue subscription methods. RabbitMQ lets you create flexible routing rules by matching text patterns. For example, you can create queues that capture all the error messages from a system and a consumer that will then consume these messages and send notifications to appropriate teams. Alternately, a consumer could write messages from a queue to a file.

Both RabbitMQ and ActiveMQ contain roughly the same features, and offer similar performance benefits.

Powerful message brokers such as ActiveMQ and RabbitMQ come with out of the box capabilities for setting up custom routing schemes and other things. Instead of

rewriting or modifying the producer/customer code, all you have to do is just configure the message broker to take advantage of these features.

As chapter 9 explains, Amazon's SQS (Amazon Simple Queue Service), while not perfect, offers benefits not available with either RabbitMQ or ActiveMQ.

The Model-View-Controller Architecture and Single Paged Applications

Most web applications retrieve data from a data store of some type (relational database, NoSQL database or a flat file) and send it to the users. The application also modifies stored data or adds new data or removes data according to what the user does in the user interface.

Since on the face of it everything revolves around the interactions between the user interface and the data store, an obvious design strategy is to link these two together directly to minimize programming and improve performance. Not so fast! While this seems to be "logical" approach, the fact that the user interfaces change frequently and given the fact that applications often incorporate business logic that goes well beyond the mere transmission of raw data from the data stores, means that one ought to look at a much more sophisticated application design.

The Problem

By combining the presentation logic with the business logic, you lose one of the biggest advantages of web based applications, which let you change the UI anytime you want, without worrying about having to redistribute the applications. It's well known that in most web applications, user interface logic changes much more often than business logic. If you combine the presentation code and business logic together, each time you make a slight change in the UI, you'll need to test all your business logic! It's very easy to introduce errors into such a web application design.

Following are the drawbacks of directly tying together the presentation and the business logic portions of a web application.

- Testing user interfaces is tedious and takes a much longer time, as compared to the testing of the business logic. Therefore, the less non-UI code you link to the UI code, the better you can test the application.
- User interface activity may include simple presentation of data by retrieving it from the data store and displaying it to the user in an attractive fashion. When the user modifies data, however, the control is passed to the business logic portion of the application, which contacts the data source and modifies the data.

- Whereas business logic is completely independent of devices, user interface code is extremely device-dependent. Instead of requiring major changes in the UI to make the application portable across different devices and all the testing and related efforts that this involves, you can simply separate the UI and business logic, both to speed up the migration to different devices, as well as to reduce the errors inherent in such a process.
- It takes different skill sets to develop great HTML pages as compared to coming up with ingenious business logic, thus necessitating the separation of the development effort for the two areas.
- A single page request tends to combine the processing of the action associated with the link chosen by the user and the rendering of that page.

MVC to the Rescue

The Model-View-Controller architecture (formally introduced in 1988, although the idea existed in a simpler form since the 1970's) separates the modeling of the application domain, the presentation of the data, and the actions based on user input into three separate classes:

- Model: the model manages the behavior and data of the application and responds to requests for information and to instructions to change the state of the model (modifying the data)
- View: manages the display of information
- Controller: interprets the user inputs and informs the model and/or the view to modify data when necessary.

In a modern web application, the view is the browser and the controller is the set of server side components that handles the HTTP requests from the users. In this architecture, the view and the controller depend on the model, but the model is independent of the two. This separation lets you build and test the model independent of how you present the data.

All modern web applications such as those based on Node.js and the Ruby on Rails framework follow the MVC design pattern to separate usr interface logic from business logic. The MVC design pattern avoids all the problems with traditional application design listed in the previous section. It also lets the model be tested independently of the presentation, by separating the model from the presentation logic.

MVC is now the well accepted architecture for web applications. Most of the popular web application frameworks follow the MVC pattern. There's some difference in how different web application frameworks interpret the MVC pattern, with the difference

being how they apportion the MVC responsibilities between the client and server. In recent years, there have been tremendous improvements in the capabilities of clients and newer web app frameworks such as AngularJS, EmberJS and Backbone let MVC components to partially execute on the client itself.

Ruby on Rails

Ruby on Rails (Rails from here on) is an extremely popular web application framework, created by David Heinemeier Hansson around 2004 as part of a project for his web application software development company named 37signals (now known as basecamp). As its name indicates, Rails is a web development framework written in the Ruby programming language. Rails is currently the #1 tool for building dynamic web applications.

The enormous popularity of Rails owes quite a bit to the use by Rails of the Ruby language, which acts as a kind of domain–specific language for developing web applications. This is what makes it so easy to perform typically complex web application programming tasks such as generating HTML and routing URLs so effortlessly, and in a highly compact and efficient manner.

The MVC Pattern and Rails

The standard Rails application structure contains an application directory called app/ with three subdirectories: models, views, and controllers. This isn't a coincidence – Rails follows the Model-View-Controller (MVC) architectural pattern. As you can recall, MVC enforces separation between the business (domain) logic and the input and presentation logic associated with the user interface. In the case of web applications, the business logic is encapsulated by data models for entities such as users and products, and the user interface is nothing but the web pages in a web browser.

When a web browser interacts with a Rails application, it sends a request that's received by a web server and is passed along to a rails controller. The controller controls what happens next. Sometimes the controller may immediately render a view, which is a template that gets converted to HTML and sent to the browser as a response.

In dynamic web applications, which most web apps are today, the controller will interact with a model. A model is a Ruby object that represents a site entity such as a user. The model is responsible for contacting the backend database and retrieving the results requested through the browser. The controller invokes the model, gets the result through the model and renders the view and returns the complete web page to the browser as HTML.

Controllers, Actions, and Routes

The controllers include what are called actions inside them. The actions are actually functions that perform specific tasks such as retrieving results from a database or printing a message on the screen.

Rails uses a router that sits in front of the controller and determines where to send requests that are coming from web browsers. There's a root route that specifies the page served on the root URL. The root URL is of the format and is often referred to simply as / ("slash"). You specify various routes in the Rails routes file (config/routes.rb). Each route consists of the controller name and the associated action. Each route is an instruction to Rails as to which action to perform, and thus which web page to create and send back to the browser.

RAILS AND REST

Earlier in this chapter, you learned about Representational State Transfer (REST), which is an architecture for developing web applications (as well as distributed networked systems). Rails uses the REST architecture, which means that application compoents such as users and microposts are modeled as resources which can be read or modified, More precisely, these resources can be created, read, updated and deleted – these operations correspond to the well-known create, select, update and delete operations of relational databases – and to the POST, GET, PATCH (early Rails versions used PUT for the updates) and DELETE requests of the HTTP protocol.

Full stack JavaScript Development with MEAN

Traditionally web programming required felicity with several programming languages. For client side programming, one was expected to know HTML (markup), CSS (styling) and JavaScript (functionality). On the server side, developers needed to know a language such as Java, PHP, Perl, plus, of course, SQL, which is a full-fledged language in its own right. In addition, when dealing with web applications, one had to know data formats well too, such as XML and JSON. The range of programming languages and the complexity involved in each of those languages (and data formats) has led to specialization among developers, into front-end and back-end development teams.

JavaScript has simplified things considerably by using a single language throughout the development stack, leading to the birth of the moniker "full stack development".

The MEAN web application stack is a framework that uses MongoDB, Express, Angular JS and Node.js for building modern web applications (SPAs). In the following sections, I briefly explain the various components that make up the MEAN stack.

MONGODB

MongoDB is highly reliable, extremely scalable and performs very well, and although it's a NoSQL database, contains some key features that are normally found only in a relational database. The big thing about MongoDB in the context of building SPAs is that it lets you use JavaScript and JSON through entire application that your developers are building.

MongoDB's command line interface uses JavaScript for querying data, meaning you can use the same expression to manipulate data as in a browser environment. Furthermore, MongoDB uses JSON as its storage format, and all the data management tools are designed with JSON in mind.

Express

Express serves as the web framework in a MEAN application. Express, built with the Ruby language (using the Sinatra framework), describes itself as a minimalist framework for Node.js, and provides a thin layer of fundamental web application features.

AngularJS

AngularJS is an extremely popular frontend framework for creating SPAs, and uses the MVC approach to web applications. HTML wasn't really designed for declaring the dynamic views that web applications require, although it's still great for declaring static documents. AngularJS helps extend the HTML vocabulary for web applications.

Node.js

Node.js is a server side platform for building scalable web applications. Node.js leverages Google Chrome's V8 JavaScript engine and uses an event driven non-blocking I/O model that helps you build real-time web applications.

Node.js isn't something that's totally alien – it's simply a headless JavaScript runtime and uses the same JavaScript engine (V8) found inside Google Chrome, with the big difference that Node.js lets you run JavaScript from the command line instead of from within a web browser.

Node.js is a popular framework that helps create scalable web applications, and it's really Node.js that has helped make JavaScript a leading alternative for server-side programming. Node.js enables JavaScript API to move beyond the browser environment by letting you use JavaScript to perform server side operations such as accessing the file system and opening network sockets.

Node has been around only for about 6 years now (starting in 2009) and has been embraced with great fervor by developers who have been able to increase throughput by using the key features of Node.

Node is especially suitable when a web application keeps connections open with a large number of users when there's no active communication between users and the server for long stretches of time. Or, there may an exchange of just a tiny bit of data between the client and the server over a period of time. Node excels in these types of environments by letting a single host running a Node.js server support a far greater number of concurrent connections than alternative technologies.

Conciseness is a hallmark of Node.js, as can be seen from the following code, which implements a web server with very little work on your side:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

This code does the following:

- Starts a server that listens on port 1337
- When the server receives a connection, it sends the message "Hello World".
- The code prints a message to the console to tell you it's running on port 1337.

The bottom line with the MEAN (or any similar framework) is that a developer is able to create a production grade web application with just HTML5, CSS3 and JavaScript. You can consider the MEAN stack as a modern day alternative to the traditional well known LAMP stack for building web applications, which consists of Linux, Apache web server, MySQL and PHP.

Single Page Applications – the new Paradigm for Web Applications

Up until just a few years ago, all web applications were what are called multipage web applications. Recently, single-page-applications (SPAs) have become very popular. In addition, modern web applications sometimes come as a mix of both the traditional multi-page and the modern single-page application paradigm. In a world where the users are very sophisticated and expect easy communication and immediate responsiveness from the websites they use, SPAs are the new standard for web applications, replacing the old clunky websites that re-render entire pages after each user click.

Let me briefly explain how the traditional approach differs from the modern way of building web applications.

The Traditional Approach

Originally when the web was created and people started building web based applications, you used just HTML, a web server, and a language such as PHP to code the

application to build the application. While you can still build web applications just fine with the traditional approach, they aren't very scalable web sites. Single-page applications have basically supplanted the traditional web application model for all heavily used websites.

The Rise of the Single-Page Application

An SPA is an application delivered to the web browser that doesn't reload the page during its use. The applications function on just a single page in the browser and they give the appearance and feeling of a desktop application, since they're smoother than traditional applications.

Like any other web application, a SPA helps users complete a specific task, say, the reading of a document. Instead of reloading the web page, it lets you read a file from the same web page where you are right now. Think of the SPA as a fat client loaded from a web server. Several developments in the recent years have helped to make single-page applications popular, the most important being the adoption of Java Script, AJAX and HTML5.

Benefits of a SPA

SPAs offer several benefits to your users as compared to traditional websites. For one, they deliver the best parts of both a desktop application and a website, since an SPA can render like a desktop application. The SPA needs to draw only those parts of the interface that change, whereas a traditional website redraws the complete page on every single user action, which means there's a pause and a "flash" during the retrieval of the new page from the server by the browser, and the subsequent redrawing of the new page.

If the web page is large or if there's a slow connection or a busy server, the user wait might be long, whereas the SPA renders rapidly and provides immediate feedback to the user.

Like a desktop application, an SPA can keep the user posted of its state by dynamically rendering progress bars or busy indicators, whereas traditional websites have no such mechanism to tell the user when the next page will be arriving.

Users can access SPAs from any web connection, with just a browser, such as smart phones, tablets, etc. They also are cross-platform like websites, whereas desktop applications aren't always so.

SPAs are highly updateable, just as a website is, since all the users need to do to access the new version of an SPA is to just refresh their browser. Some SPAs are updated multiple times in a single day, whereas updating desktop applications is no trivial affair – besides taking too long to deploy the new versions, often there could be long

intervals, sometimes as long as several years(!), between the old and the new versions of a desktop application.

JavaScript

While older technologies such as Java applets and Flash were also SPA platforms, it's JavaScript SPAs that have made SPAs popular. Up until a few years ago, Flash and Java were the most widely adopted SPA client platforms because their speed and consistency was far better than that of JavaScript and browser rendering. Even in the early days (about 10 or so years ago), JavaScript offered the following advantages compared to Flash and Java:

- A no-plugin architecture that reduces development and maintenance efforts
- The no-plug in architecture also means JavaScript needs fewer resources to run
- A single client language (JavaScript) is used for everything, instead of a bunch of languages
- More fluid and interactive web pages

However, you couldn't rely on JavaScript for consistently providing crucial capabilities on most browsers. Today, most of the early weaknesses of JavaScript have been either removed or diminished in importance, thus bringing the advantages of JavaScript and browser rendering to the fore.

JavaScript has been around for a while, being the standard for client-side scripting in its previous incarnation. On the client side, JavaScript is the only language supported by all the popular browsers. In its early days, JavaScript provided simple page interactions by performing tasks such as changing an image's attributes on mouse overs and supporting collapsible menus, this providing functionality missing in HTML. Over time, JavaScript has become more general purpose, moving beyond the client-side usage patterns of its early days.

AJAX and SPAs

Asynchronous JavaScript and XML (AJAX) is the technology that underlies a SPA application. AJAX is a non-blocking way for clients to communicate with a web server without reloading the page. Static content is where the web pages don't change at all, and are the mainstay of simple web sites with a few web pages.

Web applications use dynamic content, where the web pages are generated on the fly in response to search requests or button clicks by the users. AJAX lets the web browser poll the server for new data. In an application that has a contact form that uses AJAX to submit the form, the page won't reload when I submit the form. Instead it merely shows me a confirmation text on the page indicating that my response was

sent. When you don't use AJAX during the form submission, the entire page will reload or may bring up a completely different page with the confirmation message.

As the use of high speed internet grew several years ago, Ajax applications became more popular. The applications made background requests instead of fully reloading the web page, in order to make the page more responsive to requests. As Ajax requests grew in importance, applications consequently started used fewer page loads, culminating in the birth of the SPA, which uses just a single page load and uses Ajax calls to request all further data.

Traditional applications use a full response (POST) to the web server each time you send a form such as what I described earlier. An SPA makes asynchronous calls to update bits and pieces of the content. It's AJAX that enables the process of making these partial requests/responses to the server. In this context, it's important to note that rather than polling the server for new data, newer technologies like WebSockets make the browser maintain an open connection with the server so that the server can send data on demand.

Almost all SPAs use AJAX to load data and for other purposes, but AJAX has other uses as well. In fact, the majority of AJAX uses are non-SPA related.

You can look at SPA as a paradigm for building web applications, while AJAX represents the actual set of techniques that allows JavaScript based applications to support SPAs.

Web Services

A web service is a software component stored on a server, which can be accessed by an application or software component from a different server over a network. Web services promote the reusability of software components in internet based applications. Web services communicate through technologies such as XML, JSON and HTTP.

Two key Java APIs facilitate web services:

- JAX-WS is based on the Simple Object Access protocol (SOAP) that allows web services and clients to communicate even if they are written in different languages.
- JAX-RS: uses the Representational State Transfer (REST) network architecture based on the traditional web request and response mechanisms such as GET and POST, which I explained earlier in this chapter.

Web services, since they are platform and language independent, allow organizations to work together whether their hardware, software and communication technologies are compatible or not.

Amazon, eBay, PayPal and Google and others make their server-side applications available to their partners through web services. Using Web services, businesses can spend less time developing everything from scratch, and focus on more innovative services to provide enhanced shopping experiences for their customers.

An online music site for example, may have links to the websites of companies that sell concert tickets. In this case, the online music store is said to be consuming the concert ticket web service on its site. By consuming the concert ticket service, the music store provides additional services to its customers, besides benefiting financially from the web services. When an application consumes a web service, it means that it invokes the web services running on servers running elsewhere on the internet.

In the Java programming language, a web service is a Java class that allows its methods to be called by applications running on other servers through common data formats and protocols, such as XML, JSON and HTTP.

Web Service Basics

The server on which a web service lives is called the web service host. In the Java programming language, a web service is implemented as a class that lives in the host.

Publishing a web service is making the web service available to receive client requests. Consuming a web service is the using of a web service from a client application. Client applications send requests to the web service host and receive responses from the server. Now you can see how an application can retrieve data through a web service, without having direct access to the data. Same is the case where an application without massive processing power can utilize another server's resources to perform computations.

Web services can use one of two protocols to do their work - SOAP or REST, as explained in the following sections.

Simple Object Access Protocol (SOAP)

Simple Object Protocol (SOAP), the original technology used to communicate with web services, is a platform independent protocol that uses XML to interact with web services. The SOAP protocol describes how to mark up requests and responses so they can be sent via protocols such as HTTP. The SOAP message is XML markup that tells the web service how to process the message. Since SOAP messages are written in

XML, they're platform independent. Since firewalls allow HTTP traffic, SOAP based services can easily send and receive SOAP messages over HTTP connections.

When an application invokes a SOAP based web service's method, the request and additional information is packaged in a SOAP message within a SOAP envelope and sent to the web service host. The web service host processes the message contents by calling the method the client wishes to execute with the arguments specified by the client, and sends back the results to the client as another SOAP Message, to be parsed by the client in order to receive the results.

Representational State Transfer (REST)

Representational State Transfer (REST) is an alternative to SOAP and provides a different architectural style for implementing web services, with the web services referred to as RESTful web services. RESTful web services adhere to web standards and use traditional request and response mechanisms such as the GET and POST request methods.

In a RESTful web service, each operation is identified by a distinct URL, which lets the server know which operation to perform when it receives a request. Typically, RESTful web services return the data in the XML or JSON format, but they can also return it via HTML or plain text.

RESTful web services can be used directly from a web server or embedded in programs. When used in browser based applications, the browser can locally cache the results of REST operations when the web server in invoked via a GET request. Later requests for the operations are faster because they can be loaded from the browser's cache. Amazon's web services (aws.amazon.com) are a good example of RESTful web services.

Understanding APIs

APIs help expose data from legacy systems and also build an application without starting from scratch. Using APIs, a business can easily share information with its customers and suppliers, and customers can perform tasks such as checking the availability of a product from anywhere, and hailing a cab ride from a smartphone.

APIs are behind much of today's online work. Around 2001 companies started sharing their Web-based APIs with external users. Today, APIs are the operating system of the Web as well as mobile activity. By helping mash together data and services, APIs connect systems previously isolated from each other, and help create new applications.

APIs reduce the need to build apps from scratch – you can simply acquire APIs from providers to provide features such as payment processing and authentication. A company can bring in data from other entities to add services such as geographical maps

and credit card processing to their applications without having to write code for all the functionality. Thus, APIs reduce the cost of starting up a business and hence also reduce the barriers to entering a market. Many web sites (as well as web services) offer APIs that allow one to explicitly request data in a structured formats, thus saving you the trouble of having to scrape those sites for that data.

In addition to powering applications that customers use, APIs enhance communications between servers, thus helping automate processes and predicting problems, thus making businesses run more efficiently.

API usage has increased steadily over the past several years, driven by the increasing importance of web applications and the deluge of mobile applications. Early on, all web applications were simple monolithic designs, and used basic HTML, JavaScript and communicated over HTTP. In the past 10 years. APIs have proliferated, as alternative ways to interact with web applications. APIs help interactions with web applications and the need for organizations to integrate their systems on the web has led to the widespread use of APIs. It's mobile applications, however, that opened the floodgates for APIs. APIs help mobile applications easily access the data and functions of web applications without developers having to rewrite everything new for the mobile platform.

Two Types of Web Services

There are two main architectural styles to building web services – function centric services and resource centric services, as I explain here.

Function Centric Web Services

Function centric web services have been around for quite a while, since the early 1980s. Function centric web services can be thought of as services that would come into play where the application's code calls any function. The web service will transparently send the code and the data required for executing the function to a remote server and retrieves the results from that server, without the application being aware of the fact that the function was executed on the remote server.

Function centric technologies include technologies such as Common Object Request Broker Architecture (CORBA), Distributed Component Object Model (DCOM) and the Simple Object Access Protocol (SOAP), with SOAP being the most commonly employed technology.

SOAP uses XML to encode messages and uses HTTP to transport the requests and responses to and from the clients and servers. A web service provider uses two types of XML resources – Web Service Definition language (WSDL) files and XML Schema definition (XSD) files to describe the available methods and the definition of the data structures to be interchanged between the provider and the web application. These

two XML resources constitute a web services contract and (assuming you're using Java in this case) service developers use this information from the provider to create a Java client library and end up with a set of Java classes to implement the web service.

The SOAP server manages the SOAP libraries that perform tasks such as authentication, and error handling. Client developers on the other hand use the native Java client library generated based on the web service contract and compile and deploy the client application with the web service client code incorporated into that client application.

SOAP is very complex to implement and isn't conducive for many beginning efforts in web application development, or even for scaling web services. While the XML documents that are used for SOAP requests contain the request parameters and method names, the URLs don't contain the information required for making remote procedural calls and hence you can't cache the responses based on the URLs. Furthermore, the various web service specifications of SOAP (called the ws-* specifications) such as those relating to transactions, for example, make web service protocols stateful, thus preventing you from making the web servers stateless, which ought to be the key goal in scaling web applications.

Developing SOAP web services with dynamic languages such as Ruby and Python ran into many integration issues. Web technologies required a viable integratable option to SOAP, and this led to the creation of JavaScript Object Notation (JSON) based REST web services. Before we delve into the resource based web services, it probably is a good idea to learn a bit about JSON documents.

Resource Based Web Services

REST based web services have supplanted SOAP based function-centric web services as the leading web service architecture a few years ago, and are the current architectural standard for web services. Instead of using functions as their basis, resource centric web services revolve around the concept of a resource, and only a limited set of operations are allowed to be performed on the resource objects.

REST services use URLs to identify resources, and hence can use the HTTP methods I described earlier in this chapter, such as GET, PUT, POST and DELETE. In a REST based web service, the GET method fetches information about the resource and the POST method updates a resource or adds an entry, while DELETE will remove an object.

Compared to a typical SOAP web service architecture, with its many standards and ws-* specifications, all of which make it hard to work with and also to integrate, REST based web services are easy to set up since you deal with just four basic HTTP request methods. The REST framework needs to support only this small amount of functionality, meaning that the web stack need not be complex at all. You also don't need to

manage API contract artifacts such as the WSDL and XSD files required by SOAP based web services.

In addition to their inherent simplicity, REST web services are stateless and their GET method operations can be cached by HTTP caches between the client and the web services. This allows you to offload heavy web traffic over to reverse proxy servers to lighten the load on your web services as well as the databases.

However, it's not all roses with REST web service architectures – they do have some important drawbacks. Since they are so simple to implement, REST services require clients to authenticate using a security mechanism such as OAuth2 (a popular authorization framework that lets applications obtain limited access to user accounts on an HYTTP service such as Facebook, for example), and use HTTP's transport layer security (TLS) to encrypt messages.

REST is definitely an easier architecture for a new firm to get started with compared to SOAP based services and is much easier to integrate with other web technologies, in any web stack you are likely to employ. If your needs don't require a sophisticated architecture with numerous features, using REST based web services is an easy decision to make, due to its enormous simplicity.

Service-Based Architectures and Microservices

Starting around the year 2005, service-oriented architecture (SOA) became the most popular architecture to promote the reuse of business functionality, and to enable business groups to communicate and collaborate in a better fashion.

SOA is an architecture that uses loosely coupled and highly autonomous services that each focuses on solving a specific business need. Loosely coupled in this context means that the different compoents are independent from each other and know only a minimal amount of other components. On the practical side, this means fewer dependencies between the components, which means changes in once component are less likely to adversely affect other components.

Decoupling also lets the people who work with the components specialize in those systems and also that you can scale each component independently. The high autonomy means individual web services act like an application themselves. For example the following would all be web services that work together:

- Product Catalog Service
- Recommendation Service
- Payment Processing Service

Web services use functional partitioning to divide a complex system into independent loosely coupe applications, with each service handling a small portion of the total functionality.

In an SOA architecture, the goal is to create generic services that are highly autonomous and decoupled from other services, and can be strung together to build complex applications. Service orchestration and service policies are used to build the complex applications.

In the microservices architecture, instead of building a single monolithic architecture based application, developers build a suite of components, which work together over the network. Each of the components in a microservice architecture can be written in the language that's best suited for a task, and the components can be deployed independently of the others. In addition, when you need to scale a component, you can do so independently for just that component, without worrying about the rest of the components.

Horizontally scalable applications benefit significantly from a microservices architecture. Take the case of a financial trading firm that deals with options and futures contracts. Its application may have components such as the following:

- A user interface for the traders
- Code for setting trades that interacts with the stock exchanges and the order management system
- A proprietary pricing system

When the company modifies its pricing algorithms, only the pricing system is touched and the user interface and the back end interfaces are left alone. This obviously leads to faster changes and a more agile business.

SOA architectures typically use the Simple Object Access Protocol (SOAP), which is a set of technologies that help define, discover, and use web services. Prior to this, humongous monolithic applications were the only way to architect applications, and SOA seemed like a great alternative to this unwieldy way of architecting applications. SOA implementation however turned out to be a nightmare for many folks, since it's a complex architecture and has resulted in numerous failed projects, thus tarnishing its image.

Similarities between traditional SOAs and the Microservice approach

Both microservices and SOA are service based, meaning that services are at the center of everything. Microservices architecture is quite similar to SOA in its basic premise of breaking up inefficient monolithic applications into modular components.

Microservices are connected with each other through a thin layer of simple APIs and the well-known standards of HTTP. Services are used to implement business functionality in both approaches. They both are also distributed architectures, where the services are accessed remotely through remote access protocols such as Representational State Transfer (REST), Simple Object Access Protocol (SOAP), Java message Service (MS), Advanced Message Queuing Protocol (AMQP), Remote Method Invocation (RMI) and other similar protocols.

Distributed applications, while complex to implement, offer key benefits when compared to monolithic applications, such as increased scalability and a more focused development of the applications. The use of self-contained applications is common within a distributed architecture, helping enhance the reliability and speed of these applications, while simultaneously making it easier to maintain them.

Modularity is at the center of distributed service based architectures. Modularity means that the application is broken up into small self-contained services that you design, develop, and deploy separately, with no dependence on other application components. Whereas traditional monolithic applications usually require massive unwieldy rewrites or refactoring to incorporate changes dictated by the business, modular architectures let you rewrite the small self-contained services from scratch, thus keeping the application "fresh" in terms of its design and functionality.

While some may think that microservices are just SOA in a new garb ("Microservices are SOA done right"), there are key differences between the two architectures. Since microservices is what everybody seems to be doing these days, I'll focus on those services in this chapter and won't go into the details of traditional SOA architectures.

There are two major types of services that microservice architectures focus on: functional services that support specific business functions such as sales and marketing, and infrastructure services that support nonfunctional tasks such as auditing, authorization and logging. You can view the functional services as external facing services and the infrastructure services as private, internal shared services. Microservices adopt a share-as-little-as-possible approach, meaning that the services are sealed units with little or no dependency on other compoents.

Differences between SOA and Microservices

While both SOA and microservices are based in the idea of modular services that break up a monolithic application, there are some significant differences between the two approaches, as explained in the following sections.

Service Types

Microservices generally have fewer service types, with functional and infrastructure services being the two major service types. The typical microservices architecture looks like the following:

```
client requests => API layer => functional service => infrastructure service
```

SOA architectures are different from the microservice service types. There are more service types in SOA, such as enterprise services and application services. The typical SOA architecture resembles the following:

```
client requests => messaging middleware => enterprise services =>
application services => infrastructure services
```

In this architecture:

- Business services are abstract high level services that capture the core enterprise level business functions. These functions are usually represented through XML, Web Service Definition Language (WSDL), or Business Process Execution language (BPEL). Typically these services don't involve implementation.
- The enterprise services are the actual steps that provide the implementation for the functionality offered by the business services.
- The messaging middleware helps the business services and the corresponding enterprise services to communicate.
- Application services are even more fine-grained than enterprise services and provide narrow business functionality. These services can be invoked directly or through enterprise services.

In general, SOA means that services tend to cover a large amount of business functionality, such as a claims processing service for example. Microservices typically are very fine-grained with services that are much more narrowly focused with typical service names such as UpdateCustomerAddress and GetCreditRating, for example.

Need for Coordination

Since microservices involve fewer service types, usually the same application development team owns both the functional and the infrastructure services. However, in SOA, the larger number of service types means that there are different service owners for each of the service types. For example, business services are owned by the business users and application services by app development teams, and infrastructure services by the infrastructure service teams. Therefore, there's a need for coordination among the multiple groups to satisfy the business requests. Microservices, on the other hand, typically don't require this coordination since only one group is involved.

Time to Market

The smaller number of service types and the fact that there's no need for coordination among multiple teams means that microservices can be developed, tested, deployed and maintained with less effort and cost, and are also much faster to get to market.

Sharing the Components

SOA in general shares components among the services, with its share-as-much-as-possible architecture, whereas microservices typically consist of standalone services that are independent of each other. Microservices change and evolve independent of other services in the enterprise. While sharing repeated service functionality among multiple services does reduce duplication of business functionality, changes in those shared services over time could lead to problems, since the change may not impact all the services that share the changed service in a uniform fashion.

CHAPTER 4

Server Virtualization and Linux Containers

Server virtualization, which involves running multiple virtual servers on a single physical machine, is ubiquitous and is one of the key foundational layers of modern cloud computing architectures. Virtualization lets a physical server's resources such as CPU, RAM and storage to be shared among several virtual servers. Virtualization helps substantially lower your costs of supporting a complex computing environment, besides speeding up deployments, as virtual machines can be spun up in a fraction of the time it takes to order, receive and configure physical servers.

In this chapter, I discuss the basic architecture of server virtualization first, and follow it up by explaining the concept of a hypervisor, which is the key piece of software that serves as a resource allocation and hardware abstraction layer between the physical server and the virtual servers you create on the physical server. I also explain the different types of virtualization such as full and paravirtualization.

This chapter isn't limited to traditional hardware virtualization. Container virtualization is relatively new and is quite different from hardware virtualization. Unlike hardware virtualization, container virtualization doesn't mimic a physical server with its own OS and resources. This type of virtualization is all about enabling applications to execute in a common OS kernel. There's no need for a separate OS for each application, and therefore the containers are lightweight, and thus impose a lower overhead compared to hardware virtualization.

This chapter introduces the Linux Containers technology. Linux containers keep applications together with their runtime components by combining application isolation and image-based deployment strategies. By packaging the applications with their libraries and dependencies such as the required binaries, containers make the applications autonomous. This frees up the applications from their dependence on various components of the underlying operating system.

The fact that containers don't include an OS kernel means that they're faster and much more agile than VMs (virtual machines). The big difference is that all containers on a host must use the same OS kernel. The chapter delves into the Linux technology that makes possible containers – namely, namespaces, Linux Control Groups (Cgroups) and SELinux. Chapter 5 continues the discussion of containers, and is dedicated to container virtualization including Docker, currently the most popular way to containerize applications.

Linux Server Virtualization

Linux server virtualization is the running of one or more virtual machines on a physical server that's running the Linux operating system. Normally a server runs a single operating system (OS) at a time. As a result, application vendors had to rewrite portions of their applications so they'd work on various types of operating systems. Obviously, this is a costly process in terms of time and effort.

Hardware virtualization, which lets a single server run multiple operating systems, became a great solution for this problem. Servers running virtualization software are able to host applications that run on different operating systems, using a single hardware platform as the foundation. The host operating system supports multiple virtual machines, each of which could belong to the same, or a different OS.

Virtualization didn't happen overnight. IBM mainframe systems from about 45 years ago started allowing applications to use a portion of a system's resources. Virtualization become mainstream technology in the early 2000's when technology made it possible to offer virtualization on x86 servers. The awareness that the server utilization rate was extremely low, as well as the rising cost of maintaining data centers with their high power costs, has made virtualization wide spread. A majority of the servers running across the world today are virtual – virtual servers way outnumber physical servers.

The Architecture of Virtual Machines

As you can guess, unlike a physical machine, a virtual machine (VM) doesn't really exist – it's a software artifact that imitates or mimics a physical server. That doesn't mean that a VM is something that's only in our minds – it actually consists of a set of files.

There's a main VM configuration file that specifies how much memory and storage is allocated to the VM. The configuration file also names the virtual NICs assigned to the VM, as well as the I/O it's allowed to access. One of these files is the VM configuration file, which specifies how many CPUs, how much RAM and which I/O devices the VM can access. The configuration files show the VM storage as a set of virtual disks, which are actually files in the underlying physical file system.

When an administrator needs to duplicate a physical server, a lot of work is required to acquire the new server, install the OS and application files on it, and copy the data over. Since a VM is just a set of files, you can get one ready in literally minutes after making just a handful of changes in the VM configuration file. Alternatively, you can provision new VMs through VM templates. A template contains default settings for hardware and software. Provisioning tools can simply use a VM template and customize it when they deploy new servers.

Virtualization in the early x86 was based purely software-based virtualization. Although Pope and Goldberg in their seminal paper "Formal Requirements for Virutalizable Third Generation Architectures" specified the three key properties for a virtual machine monitor (efficiency, resource control and equivalence), it wasn't until the mid-2000's that the x86 architecture started satisfying these three requirements. Hardware-assisted virtualization is the way that these ideal requirements started being realized.

Software-based virtualization has inherent limitations. The x86 architecture employs the concept of privilege levels (also called privilege reigns) for processing machine instructions

The Virtual Machine Monitor (Hypervisor)

The key software that makes virtualization possible is the virtual machine monitor (VMM), actually known by its other name, hypervisor. A hypervisor is the software that does the heavy lifting in virtualized systems – it coordinates the low-level interaction between virtual machines and the underling host physical server hardware. The hypervisor sits between the VMs and the physical server and allows the VMs to partake of the physical server's resources such as disk drives, RAM and CPU.

Virtualization lets a powerful physical server appear as several smaller servers, thus saving you space, power and other infrastructure expenses. A big advantage of virtualizing an environment is resource sharing among the servers, meaning that when one of the virtual servers is idle or almost so, other servers running on the same physical server can use the idle resources granted to the first server and speed up their own processing.

How VMs share Resources

Virtualization lets the resources of the host server such as CPU, RAM, physical storage and network bandwidth be shared among the virtual servers running on top of a physical server. Often, even on a non-virtualized server shortage of any one of these resources can slow applications down. How, then, can multiple servers share a single set of resources without bringing the system to a halt? Let's learn how virtualization

typically handles the sharing of these resources among the VMs, to avoid bottlenecks and other performance issues.

In the context of resource allocation to the virtual machines, it's important to understand the key concept of overcommitting. Overcommitting is the allocation of more virtualized CPUs or memory than there's available on the physical server, and refers to the fact that you assume that none of the virtual servers will use their resources to the full extent on a continuous basis. This allows you to allocate the physical server's resources in a way that the sum of the allocated resources often exceeds the physical resource limit of the server. Using virtual resources in this fashion allows you to increase guest density on a physical server.

Disk Storage

Storage is not as much virtualized as the other server resources are. You simply allocate a chunk of the host storage space to each of the VMs, and this space is exclusively reserved for those VMs. Multiple VMs writing to the same storage disk might cause bottlenecks, but you can avoid them through the use of high performance disks, RAID arrays configured for speed, and network storage systems, all of which increase the throughput of data.

When it comes to storage, virtualaizion often uses the concept of thin provisioning, which lets you allocate storage in a flexible manner so as to optimize the storage available to each guest VM. Thin provisioning makes it appear that there's more physical storage on the guest than what's really available. Thin provisioning is different from overprovisioning, and applies only to storage and not to CPU or RAM.

CPU

CPU sharing is done on the basis of time slicing, wherein all the processing requests are sliced up and shared among the virtual servers. In effect, this is the same as running multiple processes on a non-virtualized server. CPU is probably the hardest resource to share, since CPU requests need to be satisfied in a timely fashion. You may at times see a small waiting time for CPU, but this is to be expected, and is no big deal. However, excessive waiting times can create havoc with the performance of applications.

Virtualized CPUs (vCPUs) can be overcommitted. You need to be careful with overcommitting vCPUs, as loads at or close to 100% CPU usage may lead to requests being dropped, or slow response times. You're likely to see performance deterioration when running more vCPUs on a VM than are present on the physical server. Virtual CPUs are best overcommitted when each guest VM has a small number of vCPUs when compared to the total CPUs of the underlying host. A hypervisor such as KVM can easily handle switches between the VMs, when you assign vCPUs at a ratio of five CPUs (on 5 VMs) per on physical CPU on the host server.

Network Bandwidth

Network bandwidth can be overprovisioned since it's unlikely that all VMs will be fully utilizing their network bandwidth at all times.

Memory

It's possible to overcommit memory as well, since it's not common to see the RAM being fully used by all the VMs running on a server at any given time. Some hypervisors can perform a "memory reclamation", whereby they reclaim RAM from the VMs to balance the load among the VMs.

It's important to remember that applications that use 100% of the allocated memory or CPU on a VM can become unstable in an overcommitted virtual environment. In a production environment, it's crtical to test extensively before overcommitting memory or CPU resources, as the overcommit ratios depend on the nature of the workloads.

Benefits offered by Virtual Machines

Virtualization offers several benefits to an IT department, such as the following:

Lower Costs

Virtualization lowers the cost of hardware purchases and maintenance, power and cooling, data center space and involves far lesser administrative and management effort.

Server Consolidation

Server consolidation is probably the most common, and one of the biggest motivating factors behind the drive to virtualize systems. Consolidation means that you reduce the footprint of your physical servers, saving not only capital outlays but also operating costs in terms of lower energy consumption in the data center. For example, you need fewer floor switches and networking capacity with virtualization when compared to physical servers.

Isolation

Since the guest operating systems are fully isolated from the underlying host, even if the VM is corrupted, the host is still in an operating state.

Easy Migration

You can move a running virtual machine from one physical server to another, without disconnecting either the client or the applications. You can move a running VM to a different physical server without impacting the users, using tools such as

vMotion (VMware) and Live Migration (RedHat Linux), both of which enhance the uptime of the virtualized systems.

Dynamic Load Balancing

You can move VMs from one physical server to another for load balancing purposes, so you can load balance your applications across the infrastructure.

Higher Availability

You can quickly restart a failed VM on a different physical server. Since virtual guests aren't very dependent on the hardware, and the host provides snapshot features. You can easily restore a known running system in the case of a disaster.

To summarize, virtualization offers several compelling benefits, which led to tis widespread usage in today's IT environments. Reduced capital outlays for purchase and support since you need to purchase fewer physical servers, faster provisioning, the ease of supporting legacy applications side by side with current applications, and the fact that virtualization gets you attuned to the way things are done in modern cloud based environments , have all been factors for its widespread use.

Drawbacks of Virtualization

Virtualization isn't a costless solution – you do need to keep in mind the following drawbacks of virtualization:

- There's often a performance overhead for the abstraction layer of virtualization
- Overprovisioning is always a potential problem in a virtualized environment and this could lead to performance degradation, especially during peak usage.
- Rewriting existing applications for a virtual environment may impose a stiff upfront cost
- Losing a single hypervisor could means losing all the VMs based on that hypervisor
- Administrators need specialized training and expertise to successfully manage the virtualized environments.

Virtualization Types

In addition to sharing the CPU and RAM of the parent server, VM guests share the I/O as well. The classification of hypervisors into different types is based on two basic criteria: the amount of hardware that's virtualized and the extent of the modifications required of the guest system. Modern virtualization is hardware based and doesn't use traditional software I/O virtualization (emulation) techniques. Software virtualization

uses slow techniques such as binary translation to run unmodified operating systems. By virtualizing at the hardware level, virtualization seeks to deliver native performance levels. The following sections explain the two popular I/O virtualization techniques – paravirtualization and full virtualization.

Paravirtualization

Paravirtualization, as the name itself indicates, isn't really "complete virtualization" since the guest OS needs to be modified.

The paravirtualization method presents a software interface to the VM that's similar to that of the host hardware. That is, instead of emulating the hardware environment, it acts as a thin layer to enable the guest system to share the system resources.

Under paravirtualization, the kernel of the guest OS running on the host server is modified, so it can recognize the virtualization software layer (hypervisor). Privileged operations are replaced by calls to the hypervisor, in order to reduce the time the guest OS will spend performing operations that are more difficult to run in the virtual environment than in the non-virtualized environment. Costly operations are performed on the native host system instead of on the guest's virtualized system. The hypervisor performs tasks on behalf of the guest OS and provides interfaces for critical kernel operations such as interrupt handling and memory management. Both Xen and VMWare are popular examples of paravirtualization.

A big difference between fully virtualized and paravirtualized architectures is that you can run different operating systems between guest and host systems under full virtualization, but not under paravirtualization.

Since paravirtualization modifies the OS, it's also called OS-assisted virtualization, with the guest OS being aware that it's being virtualized. Paravirtualization offers the following benefits:

- Under paravirtualization, the hypervisor and the virtual guests communicate directly, with the lower overhead due to direct access to the underlying hardware translating to higher performance. VMs that are "aware" that they're virtualized offer higher performance.
- Since paravirtualization doesn't include any device driver at all, it uses the device drivers in one of the guest operating systems, called the privileged guest. You therefore aren't limited to the device drivers contained in the virtualization software.

Paravirtualization, however, requires you to modify either the guest OS or the use of paravirtualized drivers. It therefore imposes the following limitations:

- You're limited to open source operating systems and proprietary operating systems where the owners have consented to make the required code modifications to work with a specific hypervisor. Paravirtualization isn't very portable since it doesn't support unmodified operating systems such as Microsoft Windows.
- Support and maintainability issues in production environments due to the OS kernel modifications needed for paravirtualization.

Paravirtualization can cover the whole kernel or just the drivers that virtualize the I/O devices. Xen, an open source virtualization project, is a good example of a paravirtualized environment. Xen virtualizes the CPU and memory by modifying the Linux kernel and it virtualizes the I/O with custom guest OS device drivers.

In addition to full and paravirtualization, there's also something called software virtualization, which uses emulation techniques to run unmodified virtual operating systems. Linux distributions such as RedHat Linux don't support software virtualization.

Full Virtualization

Full virtualization is a technique where the guest operating system is presented a simulated hardware interface by a hardware emulator. In full virtualization, the virtualization software, usually referred to as a hypervisor (guest OS drivers) emulates all hardware devices on the virtual system. The hypervisor creates an emulated hardware device and presents it to the guest operating system. This emulated hardware environment is also called a Virtual Machine Monitor or VMM, as explained earlier.

Guests use the features of the underlying host physical system to create a new virtual system called a virtual machine. All components of that the virtual machine presents to the operating system are virtualized. The hypervisor simulates specific hardware. For example when QEMU simulates an x86 machine, it provides a virtual Realtek 8139C+PCI as the network adapter. This means that the guest OS is unaware that it's running on virtual, and not on real hardware.

he VM allows the guest OS to run without any modifications and the OS behaves as if it has exclusive access to the underlying host system. Since the physical devices on the host server may be different from the emulated drivers, the hypervisor needs to process the I/O before it goes to the physical device, thus forcing the I/O operations to move through two software layers. This means not only slower I/O performance but also higher CPU usage.

In paravirtualization, the virtualaizion software layer abstracts only a portion of the host system's resources, and in full virtualization, it abstracts all of the host system resources.

Since the guest OS is a full emulation of the host hardware, this virtualization technique is called full virtualization.

You can run multiple unmodified guest operating systems independently on the same box with full virtualization. It's the hypervisor that helps run the guest operating systems without any modification, by coordinating the CPU of the virtual machine and the host machine's system resources.

The hypervisor offers CPU emulation to modify privileged and protected CPU operations performed by the guest OS. The hypervisor intercepts the system calls made by the guest operating systems to the emulated host hardware and maps them to the actual underlying hardware. You can have guest systems belonging to various operating systems such as Linux and Windows running on the same host server. Once again, the guest operating systems are completely unaware of the fact that they're virtualized and thus don't require any modifications.

Full virtualization requires complete emulation, which means more resources for processing from the hypervisor.

QEMU (which underlies KVM, to be discussed later in this chapter), VMWare ESXi and VirtualBox are popular fully virtualized hypervisors. Full virtualization offers many benefits, as summarized here:

- The hypervisor offers a standardized environment for hardware for the guest OS. Since the guest OS and the hypervisor are a consistent package together, you can migrate this package across different types of physical servers.
- The guest OS doesn't require any modification.
- It simplifies migration and portability for the virtual machines
- Applications run in truly isolated guest operating systems
- The method supports multiple operating systems which may be different in terms of their patch level or even completely different from each other, such as the Windows and Linux operating systems

The biggest drawback of full virtualization is that since the hypervisor needs to process data, some of the processing power of the host server is commandeered by the hypervisor and this degrades performance somewhat.

Type of Hypervisors

The hypervisor, by presenting virtualized hardware interfaces to all the VM guests, controls the platform resources. There are two types of Hypervisors, based on where exactly the hypervisor sits relative to the operating system and the host, named Type 1 and Type 2 hypervisors.

Type 1 Hypervisors

A Type 1 hypervisor (also called a native or bare metal hypervisor) is software that runs directly on the bare metal of the physical server, just as the host OS does. Once you install and configure the hypervisor, you can start creating guest machines on the host server.

Architecturally, the Type 1 hypervisor sits directly on the host hardware and is responsible for allocating memory to the virtual machines, as well as providing an interface for administration and for monitoring tools. VMWare ESX Server, Microsoft Hyper-V and several variations of the open source KVM hypervisor are examples of a Type 1 hypervisor.

Due to its direct access to the host server, the Type 1 hypervisor doesn't require separate CPU cycles or memory for the VMs and thus delivers greater performance.

It's important to understand that most implementations of a bare metal hypervisor require virtualization support at the hardware level through hardware assisted virtualization techniques (explained later in this chapter), and VMWare and KVM are two such hypervisors.

Type 2 Hypervisors

A Type 2 hypervisor (also called a hosted hypervisor) is deployed by loading it on top of the underlying OS running on the physical server, such as Linux or Windows. The virtualization layer runs like a hosted application directly in top of the host OS. The hypervisor provides each of the virtual machines running on the host system with resources such as a virtual BIOS, virtual devices and virtual memory. The guest operating systems depend on the host OS for accessing the host's resources.

A Type 2 hypervisor is useful in situations where you don't want to dedicate an entire server for virtualization. For example, you may want to run a Linux OS on your Windows laptop – both VMWare Workstation and Oracle VM Virtual Box are examples of Type 2 hypervisors.

Traditionally, a Type-1 hypervisor is defined as a "small operating system". Since a Type 1 hypervisor directly controls the resources of the underlying host, its performance is generally better than that of a Type 2 hypervisor, which depends on the OS to handle all interactions with the hardware. Since Type 2 hypervisors need to perform extra processing ('instruction translation'), they can potentially adversely affect the host server and the applications as well.

You can pack more VMs with a Type 1 hypervisor because this type of hypervisor doesn't compete with the host OS for resources.

Kernel Level Virtualization (Hardware Assisted Virtualization)

Under kernel level virtualization, the host OS contains extensions within its kernel to manage virtual machines. The virtualization layer is embedded in the operating system kernel itself. Since the hypervisor is embedded in the Linux kernel, it has a very small footprint and disk and network performance is higher in this mode. The popular open source Kernel Virtual Machine (KVM) virtualization model uses kernel level virtualization (hardware-assisted virtualization method).

Bare Metal versus Hosted Hypervisors

Virtualization solutions that use a Type-2 hypervisor such as VirtualBox are great for enabling single users or small organizations to run multiple VMs on a single physical server. VirtualBox and similar solutions run as client applications and not directly on the host server hardware. Enterprise computing requires high performance virtualization strategies that are closer to the host's physical hardware. Bare metal virtualization involves much less overhead and also exploits the built in hardware support for virtualization better than Type-2 hypervisors.

Most Linux systems support two types of open-source bare-metal virtualization technologies: Xen and Kernel Virtual Machine (KVM). Both Xen and KVM support full virtualization, and Xen also supports the paravirtualization mode. Let's start with a review of the older Xen technology and then move on to KVM virtualization, which is the de facto standard for virtualization in most Linux distributions today.

Xen Virtualization

Xen was created in 2003 and acquired later on by Citrix, which announced in 2013 that the Xen Project would be a collaborative project between itself, Xen's main contributor, and the Linux foundation. Xen is very popular in the public cloud environment with companies such as Amazon Web Services and Rackspace Cloud using it for their customers.

Xen is capable of running multiple types of guest operating systems. When you boot the Xen hypervisor on the host physical hardware, it automatically starts a primary virtual machine called Domain 0 (or dom0), or the management domain. Domain 0

manages the systems and by performing tasks such as creating additional virtual machines, and managing the virtual devices for the virtual machines, as well as tasks such as suspending, resuming, and migrating virtual machines, the primary VM will provide the virtual management capabilities for all other VMs, called the Xen guests. You administer Xen through the xm command-line suite.

The Xen daemon, named xend, runs in the dom0 VM and is the central controller of virtual resources across all VMs running on the Xen hypervisor. You can manage the VMs using an open source virtual machine manager such as OpenXenManager, or a commercial manager such as Citrix XenCenter.

Xen Architecture

Xen is a Type 1 hypervisor and so it runs directly on the host hardware. Xen inserts a virtualization layer between the hardware and the virtual machines, by creating pools of system resources, and the VMs treat the virtualized resources as if they were physical resources.

Xen uses paravirtualization, means the guest OS must be modified to support the Xen environment. The modification of the guest OS lets Xen use the guest OS as the "most privileged software". Paravirtualization also enables Xen to use more efficient interfaces such as virtual block devices to emulate hardware devices.

Xen's Benefits and Drawbacks

Xen offers highly optimized performance due to its combination of paravirtualization and hardware assisted virtualization. However, it has a fairly large footprint and integrating it isn't easy and could overwhelm the Linux kernel over time. It also relies on third-party products for device drivers as well as for backup and recovery and for fault tolerance. High I/O usually slows down Xen based systems.

While Xen offers a higher performance than KVM, it's the ease of use of KVM virtualization which has led to it's becoming the leading virtualization solution in Linux environments.

KVM supports native virtualization on processors that contain extensions for hardware virtualization. KVM supports several types of processers and guest operating systems, such as Linux (many distributions), Windows, and Solaris. There's also a modified version of QEMU that uses KVM to run Mac OS X virtual machines.

Kernel-Based Virtual Machines (KVM)

Linux KVM (Kernel-based Virtual Machine) is the most popular open-source virtualization technology today. Over the past few years, KVM has overtaken Xen as the default open source technology for creating virtual machines on most Linux distributions.

Although KVM has been part of the Linux kernel since the 2.6.20 release (2007), until release 3.0, you had to apply several patches to integrate KVM support into the Linux kernel. Post 3.0 Linux kernels automatically enable KVM's integration into the kernel, allowing it to take advantage of improvements in the Linux kernel versions. Being a part of the Linux kernel is a big deal, since it means frequent updates and a lower Total Cost of Operation (TCO). In addition, KVM is highly secure since it's integrated with SELinux in both RedHat Linux and CentOS.

KVM differs from Xen in that it uses the Linux kernel as its hypervisor. Although a Type-1 hypervisor is supposed to be similar to a small OS, the fact that you can configure a custom lightweight Linux kernel and the availability of large amounts of RAM on today's powerful 64-bit servers means that the size of the Linux kernel isn't a hindrance.

Just as Xen has its xm toolset, KVM has an administrative infrastructure that it has inherited from QEMU (short for Quick Emulator) a Linux emulation and virtualization package which achieves superior performance by using dynamic translation. By executing the guest cod directly on the host CPU, QEMU achieves performance close to the native OS.

QEMU supports virtualization while executing under the Xen hypervisor or by using the Linux KVM kernel module .Red Hat has developed the libvert virtualization API to help simplify the administration of various virtualization technologies such as KVM, Xen, LXC containers, VirtualBox and Microsoft Hyper-V. As an administrator, it's great to learn libvert because you can manage multiple virtualization technologies by learning a single set of commands (command line and graphical) based on the libvert API.

In order to support KVM virtualization, you need to install various packages, with the required package list depending on your Linux distribution.

Using Storage Pools

When you create one or two KVM based VMs, you can use disk images that you can create on the local disk storage of the host. Each VM will in essence be a disk image stored in a local file. However, for creating enterprise wide virtualization environments, this manual process of creating VMs is quite tedious and hard to manage. The libvert package lets you create storage pools to serve as an abstraction for the actual VM images and file systems.

The libvert package provides standard, technology independent administrative commands to manage virtualization environments.

A storage pool is a specific amount of storage set aside by the administrator for use by the guest VMs.

Storage pools are divided into storage volumes which are then assigned to guest VMs as block devices.

A storage pool can be a local directory, physical disk, logical volume, or a network file system (NFS) or block–level networked storage managed by libvert. Using libvert, you manage the storage pool and create and store VM images in the pool. Note that in order to perform a live migration of a VM to a different server, you should locate the VM disk image in an NFS, block-level networked storage, or in HBA (SCSI Host Bus Adapter) storage that can be accessed from multiple hosts.

Creating the Virtual Machines

The libvirt package contains the virsh command suite that provides the commands to create and manage the virtualization objects that libvert uses, such as the domains (VMs), storage pool, networks, devices, etc. Following is an example that shows how to create an NFS based (netfs) storage pool:

```
virsh pool-create-as NFS-POOL netfs \
--source-host 192.168.6.248 \
--source-path /DATA/POOL \
--target /var/lib/libvirt/images/MY-NFS-POOL
```

In this command, MY-NFS-POOL is the name of the new storage pool and the local mount point that'll be used to access this NFS based storage pool is /var/lib/libvirt/images/MY-NFS-POOL. Once you create the storage pool as shown here, you can create VMs in that pool with the virt-install command, as shown here:

```
virt-install
--name RHEL-6.3-LAMP \
--os-type=linux \
--os-variant=rhel6 \
--cdrom /mnt/ISO/rhel63-server-x86_64.iso \
--disk pool=My-NFS-POOL,format=raw,size=100 \
--ram 4096 \
--vcpus=2 \
--network bridge=br0 \
--hvm \
--virt-type=kvm \
```

Here's a summary of the key options specified with the virt-install command:

- --os-type and –os-variant: indicate that this VM will be optimized for the Linux RedHat Enterprise Linux 6,3 release,
- --cdrom: specifies the ISP image (virtual CDROM device that will be used to perform this installation)

- --disk: specifies that the VM will be created with 100GB of storage from the storage pool named NFS-01.
- --ram and –vcpus: specify the RAM and virtual CPUs for the VM
- --hvm: indicates that this is a fully virtualized system (default)
- --virt-type: specifies kvm as the hypervisor (default)

RedHat Enterprise Virtualization (RHEV) is based on KVM.

Considerations in Selecting the Physical Servers for virtualization

The choice of the physical servers for a virtualized environment is critical, and you've several choices, as explained in the following sections.

Build Your Own versus Purchasing

You can create your own servers by purchasing and putting together all the individual components such as the disk drives, RAM and CPU. You should expect to spend less to build your own systems, so that's good, but the drawback is the time it takes to get your systems together. In addition, you're responsible for maintaining these systems with partial or no service contracts to support you, with all the attendant headaches.

If you're considering putting together your own systems, it may be a good idea to check out the Open Compute project (*http://opencompute.org*), which creates low cost server hardware specifications and mechanical drawings (designs). The goal of Open Compute is to design servers that efficient, inexpensive and easy to service. Consequently these specs contain far fewer parts than traditional servers. You can try and purchase hardware that meets these specifications to ensure you're getting good hardware when you're trying to keep expenses low.

Purchasing complete systems from a well-known vendor is the easiest and most reliable way to go, since you don't need to worry about the quality of the hardware and software, in addition to getting first class support. It also gets maintenance off your hands. However, as you know, you're going to pay for all the bells and whistles.

Rack and Blade Servers

Blade servers are commonly used for virtualization since they allow for a larger number of virtual machines per chassis. Rack servers don't let you create as many VMs per chassis.

The choice of the type of server depends on factors such as their ease of maintainability, power consumption, remote console access, server form factor, and so on.

Migrating Virtual Machines

Migrating virtual machines means the moving of a guest virtual machine from one server to another. You can migrate a VM in two ways: live and offline, as explained here:

- Live Migration: this process moves an active VM from one physical server to another. In Red Hat Enterprise Linux, this process moves the VM's memory and its disk volumes as well, using what's called live block migration.
- Offline Migration: During an offline migration, you shut down the guest VM and move the image of the VM's memory to the new host. You can then resume the VM on the destination host and the memory previously used by the VM on the original host is released back to the host.

You can migrate VMs for the following purposes:

- Load Balancing: You can migrate one or more VMs from a host to relieve it's load
- Upgrades to the Host: When you're upgrading the OS on a host, you can avoid downtime for your applications by migrating the VMs on that host to other hosts.
- Geographical Reasons: Sometimes you may want to migrate a VM to a different host in a different geographical location, to lower the latency of the applications hosted by the VM.

Application Deployment and Management with Linux Containers

A Linux container (LXc) is a set of processes that are isolated from other processes running on a server. While virtualization and its hypervisors logically abstract the hardware, containers provide isolation, letting multiple applications share the same OS instance.

You can use a container to encapsulate different types of application dependencies. For example, if your application requires a particular version of a database or scripting language, the containers can encapsulate those versions. This means that multiple versions of the database or scripting language can run in the same environment, without requiring a completely different software stack for each application, each with its own OS. You don't pay for all of this with a performance hit, as containerized

applications deliver roughly the same performance as applications that you deploy on bare metal.

Linux Containers have increasingly become an alternative to using traditional virtualization, so much so that containerization is often referred to as the "new virtualization".

On the face of it both virtualization and containerization seem to perform the same function by letting you run multiple virtual operating systems on top of a single OS kernel. However, unlike in traditional virtualization, a container doesn't run multiple operating systems. Rather, it "contains" the multiple guest operating systems in their own userspace, while running a single OS kernel.

At a simple level, containers involve less overhead since there's no need to emulate the hardware. The big drawback is that you can't run multiple types of operating systems in the same hardware. You can run 10 Linux instances in a server with container based virtualization, but you can't run both Linux and Microsoft Server guests side by side.

This chapter introduces you to Linux container technology and the principles that underlie that technology, and also compares traditional virtualization with containerization. Chapter 5 is dedicated to Docker containers and container orchestration technologies such as Kubernates.

Linux Containers (LXc) allow the running of multiple isolated server installs called containers on a single host. LXc doesn't use offer a virtual machine – instead it offers a virtual environment with its own process and network space.

Linux containers have analogies in other well known 'Nix operating systems:

- FreeBSD: Jails
- SmartOS: Zones
- Oracle Solaris: Zones

Linux containers (through Docker) are radically changing the way applications are built, deployed and instantiated. By making it easy to package the applications along with all of their dependencies, containers accelerate application delivery. You can run the same containerized applications in all your environments – dev, test, and production. Furthermore, your platform can be anything: a physical server, a virtual server, or the public cloud.

Containers are designed to provide fast and efficient virtualization. Containerization provides different views of the system to different processes, by compartmentalizing

the system. This compartmentalization ensures guaranteed access to resources such as CPU and IO, while maintaining security.

Since each container shares the same hardware as well as the Linux kernel with the host system, containerization isn't the same as full virtualization. Although the containers running on a host share the same host hardware and kernel, they can run different Linux distributions. For example, a container can run CentOS while the host runs on Ubuntu.

NOTE Linux Containers (LXC) constitute a container management system that became part of the Linux Kernel 2.6.24 in August 2008. As with Docker (see Chapter 5), Linux Containers make use of several Linux kernel modules such as cgroups, SELinux, and AppArmor.

Linux Containers combine an application and all of its dependencies into a package which you can make a versioned artifact. Containers provide application isolation while offering the flexibility of image-based deployment methods. Containers help isolate applications to avoid conflicts between their runtime dependencies and their configurations, and allow you to run different versions of the same application on the same host. This type of deployment provides a way to roll back to an older version of an application if a newer version doesn't quite pan out.

Linux containers have their roots in the release of the chroot tool in 1982, which is a filesystem specific container type virtualization tool. Let's quickly review chroot briefly to see how it compares to modern containerization.

Chroot and Containers

Linux containerization is often seen as an advancement of the chroot technique, with dimensions other than just the file system. Whereas chroot offers isolation just at the file system level, LXc offer full isolation between the host and a container and between a container and other containers.

The Linux chroot() command (pronounced "cha-root") lets a process (and its child processes) redefine the root directory from their perspective. For example, If you chroot the directory /www, and when you issue the command cd, instead of taking you to the normal root directory ("/"), it leaves you at /www. Although /www isn't really the root directory, the program believes that it's so. In essence, chroot restricts the environment and that's the reason the environment is also referred to as a jail or chroot jail.

Since a process has a restricted view of the system, it can't access files outside of its directory, as well as libraries and files from other directories. An application must therefore have all the files that it needs right in the chroot environment. The key prin-

ciple here is that the environment should be self-contained within a single directory, with a faux root directory structure.

Linux containers are similar to chroot, but offer more isolation. Linux containers use additional concepts beyond chroot, such as control groups. Whereas chroot is limited to the file subsystem, control groups enable you to define groups encompassing several processes (such as "sshd" for example) and control resource usage for those groups for various subsystems such as the file system, memory, CPU and network resources and block devices.

Applications and their Isolation

Isolating applications is a key reason for using container technologies. In this context, an application is a unit of software that provides a specific set of services. While users are concerned just with the functionality of applications, administrators need to worry about the external dependencies that all applications must satisfy. These external dependencies include system libraries, third-party packages and databases.

Each of the "dependencies" has its own configuration requirements and running multiple versions of an application on a host is difficult due to potential conflicts among these requirements. For example, a version of an application may require a different set of system libraries than another version of the same application. While you can somehow manage to run multiple versions simultaneously through elaborate workarounds, the easiest solution to managing the dependencies is to isolate the applications.

Virtualization and Containerization

Both containerization and virtualization help address the issues involved in efficient application delivery, where applications in general are much more complex yet must be developed with lower expense and delivered faster, so they can quickly respond to changing business requirements.

At one level, you can view both containers and traditional virtualization as allowing you to do the same thing: let you run multiple applications on the same physical servers. How then, are containers a potentially better approach? Virtualization is great for abstracting from the underlying hardware, which helps lower your costs through consolidating servers, and make it easy to automate the provision of a complete stack that includes the OS, the application code and all of its dependencies.

However, great as the benefits of virtualization are, virtual machines have several limitations:

- By replacing physical servers with virtual servers, you do reduce the physical server units – yet server sprawl doesn't go away – you're simply replacing one type of sprawl with another!
- Virtual technology isn't suitable for microservices that can uses hundreds of thousands of processes, since each OS process requires a separate VM.
- Virtual machines can't be instantiated very quickly – they take several minutes to spin up, with means inferior user experience. Containers on the other hand can be spun up blazingly fast- within a few short seconds!
- Lifecycle management of VMs isn't a trivial affair – every VM has a minimum of two operating systems that need patching and upgrading – the hypervisor and the guest OS inside the VM. If you have a virtualized application with 20 VMs, you need to worry about patching 21 operating systems (20 guests systems + 1 hypervisor).

While traditional virtualization does offer complete isolation, once you secure containers with Linux namespaces, CGroups and SELinux, you can get virtually (no pun) the same amount of isolation. Linux containers offer a far more efficient way to build, deploy, and execute applications in today's modern application architectures that uses microservices and other new application paradigms. Linux containers offer an application isolation mechanism for lightweight multitenancy and offer simplified application delivery.

Containers, as I've mentioned earlier, have been around for over a decade, and things like Solaris Zones and FreeBDS Jails have been with us for even longer. The new thing about current containerization is that it's being used to encapsulate an application's components, including the application dependencies and required services. This encapsulation makes the applications portable. Docker has contributed substantially to the growth of containerization, by offering easy to use management tools as well as a great repository of container images. RedHat and others have also offered smaller footprint operating systems and frameworks for management, as well as containerization orchestration tools such as Kubernates (please see Chapter 5).

Benefits offered by Linux Containers

Linux containers enhance the efficiency of application building, shipping, deploying, and execution. Here's a summary of the benefits offered by containers.

Easier and Faster Provisioning

While VMs take several minutes to boot up, you can boot up a containerized application in mere seconds, due to the lack of the overhead imposed by a hypervisor and a guest OS. If you need to scale up the environment using a public cloud service, the ability to boot up fast is highly beneficial.

Lower Costs

Due to their minimal footprint, many more containers fit on a physical server than virtual machines.

Better Resource Utilization

You can monitor containers easily since they all run on a single OS instance. When idle, the containers don't use any server resources such as memory and CPU unlike a virtual machine, which grabs those resources when you start it up. You can also easily remove unused container instances and prevent a virtual machine like sprawl.

Easier Lifecycle Management

Even if you have a large number of applications running on a containerized server, you need to patch and upgrade just a single operating system, regardless of how many containers run on it, unlike in the case of virtual machines. Since there are fewer operating systems to take care of, you're more likely to upgrade than to apply incremental patches.

Greater Application Mobility

Containers make it easy to move application workload between private and public clouds. Virtual machines are usually much larger than containers, with sizes ranging often in the Gigabytes. Containers are invariably small (a few MBs) and so it's easier to transport and instantiate them.

Quicker Response to Changing Workloads

Containers speed up application development due the testing cycles being shorter, owing to the containers including all the application dependencies. You can build an app once and deploy it anywhere.

Easier Administration and Better Visibility

You have far fewer operating systems to manage since multiple containers share the same OS kernel. You also have better visibility into the workload of a container from the host environment, unlike with VMs, where you can't peek inside the VM.

Containers aren't a mere incremental enhancement of traditional virtualization. They offer many ways to speed up application development and deployment, especially in the areas of microservices, which I discussed in Chapter 3. Since microservices can startup and shutdown far quicker than traditional applications, containers are ideal for them. You can also scale resources such as CPU and memory independently for microservices with a container based approach.

Isolation of Services

Let's say your organization has a sensitive application that makes uses of SSL to encrypt data flowing through the public internet. If you're using a virtualized setup, the application image includes SSL and therefore, you'll need to modify the application image whenever there are SSL security flaws. Obviously, your application is down during this time period, which may be long, since you'll need to perform regression testing after making changes to SSL.

If you were using a container based architecture on the other hand, you can separate the SSL portion of the application and place it in its own container. The application code isn't intertwined with SSL in this architecture. Since you don't need to modify the application code, there's no need for any regression testing of the application following changes in SSL. A huge difference!

Two Types of Uses for Linux Containers

There are two different ways in which you can employ Linux containers. You can use containers for sandboxing applications, or you can utilize image-based containers to take advantage of the whole range of features offered by containerization. I explain the two approaches in the following sections.

Host Containers

Under the host containers use case, you use containers as lightweight application sandboxes. All you applications running in various containers are based on the same OS as the host, since all containers run in the same user space as the host system. For example, you can carve a RHEL 7 host into multiple secure and identical containers, with each container running a RHEL 7 userspace. Maintenance is easy since updates need to be applied just to the host system. The disadvantage to this type of containerization is that it's limited to just one type of OS, in this example a RHEL runtime.

Image-Based Containers

Image based containers include not just the application, but also application's runtime stack. Thus, the container runs an application that has nothing to do with the host OS. Both the container and application run times are packaged together and deployed as an image. The containers can be non-identical under image based containerization. This means you can run multiple instances of the same application on a server, each running on a different OS platform. This is especially useful when you need to run together application versions based on different OS versions such as RHEL 6.6 and RHEL 7.1, for example. Docker, which we discuss extensively in Chapter 5, is based on image based containers. Docker builds on LXc and it includes the userspace runtime of applications.

You can deploy containers both on bare metal, and on virtualized servers.

The Building Blocks of Linux Containers

Linux containers have become increasingly important as application packaging and delivery technology. Containers provide application isolation along with the flexibility of image-based deployment. Linux containers depend on several key components offered by the Linux kernel, such as the following:

- Cgroups (Control groups) – allow you to group processes for optimizing system resource usage by allocating resources among user-defined groups of tasks
- Namespaces – isolates processes by abstracting specific global system resources and making them appear as a distinct instance to all the processes within a namespace
- SELinux – securely separates containers by applying SELinux policy and labels.

Namespaces, cgroups and SELinux are all part of the Linux kernel, and they provide the support for containers, which run the applications. While there's a bunch of other technologies used by containers, namespaces, cgroups and SELinux account for most of the benefits you see with containers.

In the following sections, let's briefly review the key building blocks of Linux containers:

- Process isolation – namespaces provide this
- Resource management – Cgroups provide resource management capabilities
- Security – SELinux takes care of security

Namespaces and Process Isolation

The ability to create multiple namespaces enables process isolation. It's namespaces that make it possible for Linux containers to provide isolation between applications, with each namespace providing a boundary around applications. Each of these applications is a self-contained entity with its own file system, hostname and even a network stack. It's when it's running within a namespace that an application is considered to be running within a container.

A namespace makes a global system resource appear as a dedicated resource to processes running within that namespace. This helps different processes see different

views of the system, something which is similar to the concept of zones in the Solaris operating system. This separation of the resource instances lets multiple containers simultaneously use the same resource without conflicts. Namespaces offer lightweight process virtualization, although without a hypervisor layer as in the case of OS virtualization architectures such as KVM.

Mount namespaces together with chroots help you create isolated Linux installations for running non-conflicting applications.

In order to isolate multiple processes from each other, you need to isolate them at every place they may bump into each other. For example, the file system and network are two obvious points of conflict between two applications. Application containers use several types of namespaces as described here, each of which helps isolate a specific type of resource, such as the file system or the network.

Namespaces isolate processes and let you create a new environment with a subset of the resources. Once you set up a namespace, it's transparent to the processes. In most Linux distributions, the following namespaces are supported, with RHEL 7 and other distributions adding the user namespace as well.

- mnt
- net
- uts
- pid
- ipc

Let's briefly discuss the namespaces listed here in the following sections.

Mount Namespaces

Normally, file system mount points are global, meaning that all processes see the same set of mount points. Mount namespaces isolate the set of filesystem mount points viewed by various processes. Processes running within different mount namespaces, however, can each have different views of a file system hierarchy. Thus, a container can have a different /tmp directory from that of another container. The fact that each application sees a different file system means that dependent objects can be installed without conflicts among the applications.

UTS Namespaces

UTS namespaces let multiple containers have separate hostnames and domain names, thus providing isolation of these two system identifiers. UTS namespaces are useful when you combine them with network namespaces.

PID Namespaces

PID namespaces allow processes running in various containers to use the same PID, so each container can have its own init process, which is PID1. While you can view all processes running inside the containers from the host operating system, you can only see a container's own set of processes from that container. All processes, however, are visible within the "root" PID namespace.

Network Namespaces

Network namespaces allow containers to isolate the network stack, which includes things such as the network controllers, firewall, iptable rules, routing tables, etc. Each container can use separate virtual or real devices and have its own IP address. Network namespaces remove port conflicts among applications, since each application uses its own network stack, with a dedicated network address and TCP port.

IPC Namespaces

IPC (inter process communication) namespaces allow interprocess communication (IPC) resource isolation, which allows containers to create shared memory segments and semaphores with identical names, although they can't influence those resources that belong to other containers. Inter process communication environment includes things such as message queues, semaphores and shared memory.

Control Groups (cgroups)

Control groups (cgroups for short) let you allocate resources such as CPU time, block IO, RAM and network bandwidth among groups of tasks that you can define, thus providing you fine-grained control over system resources. Using cgroups, the administrator can hierarchically group and label processes and assign specific amounts of resources to these processes, thus making for an efficient allocation of resources.

The Linux nice command lets you set the "niceness" of a process, which influences the scheduling of that process. The nice values can range from -20 (most favorable scheduling) to a value of 19 (least favorable to the process). A process with a high niceness value is accorded lower priority and less CPU time, thus freeing up resources in favor of processes with a lower niceness value. Note that niceness doesn't really translate to priority – the scheduler is free to ignore the nice level you set. In traditional systems, all processes receive the same amount of system resources, and so an application with a larger number of processes can grab more system resources com-

pared to applications with fewer running processes. The relative importance of the application should ideally be the criterion on which resources ought to be allocated, but it isn't so, since resources are allocated at the process level.

Control groups let you move resource allocation from the process level to the application level. Control groups do this by first grouping and labeling processes into hierarchies, and setting resource limits for them. These cgroup hierarchies are then bound with the systemd unit tree, letting you manage system resources with the systemctl commands (or by editing the system unit files).

A Cgroup is a kernel provided filesystem, and is usually mounted at /cgroup, and contains directories similar to /proc and /sys that represent the running environment and kernel configuration options. In the following sections, I explain how cgroups are implemented in RedHat Enterprise Linux (and Fedora).

Cgroup Hierarchies

You organize cgroups in a tree-based hierarchy. Each process or task that runs on a server is in one and only one of the cgroups in a hierarchy. In a cgroup a number of tasks (same as processes) are associated with a set of subsystems. The subsystems act as parameters than can be assigned and define the "resource controllers" for memory, disk I/O, etc.

In RHEL 7 (and CentOS), the systemd process, which is the parent of all processes, provides three unit types for controlling resource usage – services, scopes and slices. Systemd automatically creates a hierarchy of slice, scope and service units that provide the structure for the cgroup tree. All three of these unit types can be created by the system administrator or by programs. Systemd also automatically mounts the hierarchies for important kernel resource controllers such as devices (allows or denies access to devices for tasks in a group), or memory (sets limits on memory usage by a cgroup's tasks). You can also create custom slices of your own with the systemctl command.

Here's a brief description of the three unit types provided by systemd:

- Service: services let systemd start and stop a process or a set of processes as a single unit. Services are named as name.service.
- Scope: processes such as user sessions, containers and VMs are called scopes and represent groups of externally created processes. Scopes are named as name.scope. For example, Apache processes and MySQL processes can belong to the same service but to different scopes – the first to the apache scope and the second to the Mysql scope.
- Slice: a slice is group of hierarchically organized scopes and services. Slices don't contain any processes – it's the scopes and services that do. Since a slice is hierarchical in nature, the name of a slice unit corresponds to its path in the hierar-

chy. If the slice name is parent-name.slice, the slice named parent-name.slice is a subslice of the parent slice.

The kernel creates the following four slices by default to run the system:

- -.slice – root slice
- system.slice – default location for system services (systemd automatically assigns services to this slice)
- user.slice – default location for user sessions
- machine.slice - default location for VMs and Linux containers

Slices are assigned to scopes. Users are assigned implicit subslices and you can define new slices and assign services and scopes to those slices. You can create permanent services and slice units with unit files. You can also create transient service and slice units at runtime through issuing API calls to PID 1. Transient services and slice units don't survive a reboot, and are released after they finish.

You can create two types of cgroups: transient and persistent. You can create transient cgroups for a service with the systemd-run command, and set limits on resources that the service can use. You can also assign a persistent cgroup to a service by editing its unit configuration file. The following example shows the syntax for creating a transient cgroup with systemd-run:

```
systemd-run --unit=name --scope --slice=slice_name command
```

Once you create the cgroup, you can start a new service with the systemd-run command, as shown here:

```
# systemd-run --unit=toptest --slice=test top -b
```

This command runs the top utility in a service unit, within a new slice named test.

You can override resources by configuring the unit file or at the command line as shown here:

```
#  systemctl set-property httpd.service CPUShares=524 MemoryLimit=500M
```

Unlike in virtualized systems, containers don't have a hypervisor to manage resource allocation, and each container appears as a regular Linux process from the point of view of the OS. Using cgroups helps allocate resources efficiently since you're using groups instead of processes. A CPU scheduler, for example, finds it easy to allocate resources among groups rather than among a large number of processes.

Systemd stores the configuration for each persistent unit in the /usr/lib/systemd/system directory. To change the configuration of a service unit you must modify the configuration file either manually by editing the file, or with the systemctl set-property command.

Viewing the Hierarchy of Control Groups

You can view the hierarchy of the control groups with the systemd-cgls command in RHEL 7, as shown in the following output from the command, which also shows you the actual processes running in the cgroups.

```
├─1 /usr/lib/systemd/systemd --switched-root --system --deserialize 20
├─user.slice
│ └─user-1000.slice
│   └─session-1.scope
│     ├─11459 gdm-session-worker [pam/gdm-password]
│     ├─11471 gnome-session --session gnome-classic
│     ...
│
└─system.slice
  ├─systemd-journald.service
  │ └─422 /usr/lib/systemd/systemd-journald
  ├─bluetooth.service
  │ │ └─5328 /usr/lib/systemd/systemd-localed
    ├─sshd.service
  │ └─1191 /usr/sbin/sshd -D
  │
  ...
```

As the output reveals, slices don't contain processes – it's the scopes and services that contain the processes. The -.slice is implicit and identifies with the hierarchy's root.

Cgroup Subsystems (Resource Controllers)

In older versions of Linux, administrators used the libcgroup package and the cgconfig command to build custom cgroup hierarchies. In this section, I show how RHEL 7 moves resource management from the process to the application level, by binding the cgroup hierarchies with the systemd unit tree. This lets you manage the resources either through systemctl commands or by editing the systemd unit files. You can still use the libcgroup package in release 7, but it's there only to assure backward compatibility.

A cgroup subsystem is also called a resource controller and stands for specific resources such as CPU or memory. The kernel (systemd) automatically mounts a set of resource controllers, and you can get the list from the /proc/cgroups file (or the lssubsys command). Here are the key controllers in a RHEL 7 system:

- cpu: provides cgroup tasks access to the CPU
- cpuset: assigns individual CPUs to tasks in a cgroup
- freezer: suspends or resumes all tasks in a cgroup when they reach a defined checkpoint
- memory: limits memory usage by a cgroup's tasks

Systemd provides a set of parameters with which you can tune the resource controllers.

Optimizing Resource Usage with CGroups

Now that you have a basic idea about cgroups and resource controllers, let's see how you can use cgroups to optimize resource usage.

Let's say you have two MySQL database servers, each running within its own KVM guest. Let's also assume that one of these is a high priority database and the other, a low priority database. When you run the two database servers together, by default the I/O throughput is the same for both.

Since one of the database services is a high priority database, you can prioritize the I/O throughput by assigning the high priority database service to a cgroup with large number of reserved I/O operations. At the same time, assign the low priority database server to a cgroup with a lower number of reserved I/O operations.

In order to prioritize the I/O throughput, you must first turn on resource accounting for both database servers:

```
# systemctl set-property db1.service BlockIOAccounting=true
# systemctl set-property db2.service BlockIOAccounting=true
```

Next, you can set the priority by setting a ratio of 5:1 between the high and low priority database services, as shown here:

```
# systemctl set-property db1.service BlockIOWeight=500
# systemctl set-property db2.service BlockIOWeight=100
```

I employed the resource controller BlockIOWeight in this example to priorotize the I/O throughput between the two database services, but you could also have configured block device I/O throttling by setting the blkio controller to achieve the same result.

SELinux and Container Security

The two major components of the container architecture that you've seen thus far – namespaces and cgroups aren't designed for providing security. Namespaces are good at making sure that the /dev directory in each container is isolated from changes in the host. However, a bad process from a container can still potentially hurt the host system. Similarly, cgroups help with the avoiding of denial of service attacks since they limit the resources any single container can use. However, SELinux, the third major component of modern container architectures is designed expressly to provide security, not only for containers, but for also normal Linux environments.

RedHat has been a significant contributor to SELinux over the years, along with the Secure Computing Corporation. The National Security Agency (NSA) developed

SELinux to provide the Mandatory Access Control (MAC) framework often required by the military and similar agencies. Under SELinux, processes and files are assigned a type and access to them is controlled through fine-grained access control polices. This limits potential damage from well-known security vulnerabilities such as buffer overflow attacks. SELinux significantly enhances the security of virtualized guests, in addition to the hosts themselves.

How SELinux Works

SELinux implements the following mechanisms in the Linux kernel:

- Mandatory Access Control (MAC)
- Multi-level Security (MLS)
- Multi-category security (MCS)

The sVirt package enhances SELinux and uses Libvirt to provide a MAC system for containers (and also for virtual machines). SELinux can then securely separate containers by keeping a container's root processes from affecting processes running outside the container.

Enabling SELinux

SELinux can be disabled or made to run in a permissive mode. In both of these modes, SELinux won't provide sufficient secure separation among containers. Following is a brief review of SELinux modes and how you can enable it.

SELinux operates in two modes (three if you want to add the default "disabled" mode) – enforcing and permissive. While SELinux is enabled in both the Enforcing and the Permissive modes, SELinux security policies are enforced only in the Enforcing mode. In the Permissive mode, the security policies are read but not applied. You can check the current SELinux mode with the getenforce command:

```
# getenforce
Enabled
#
```

There are several ways to set the SELinux mode, but the easiest way is to use the setenforce command. Here are ways you can execute this command:

```
# setenforce 1
# setenforce Enforcing
# setenforce 0
# setenforce Permissive
```

The first two options set the mode to Enforcing and the last two to Permissive.

sVert

RHEL 7 provides Secure Virtualization (sVirt), which integrates SELinux and virtualization, by applying MAC (Mandatory Access Controls) when using hypervisors and VMs. sVirt works very well with KVM, since both are part of the Linux kernel.

By implementing the MAC architecture in the Linux kernel, SELinux limits access to all resources on the host. In order to determine which users can accept a resource, each resource is configured with a SELinux context such as the following:

```
system_u:object_r:httpd_sys_content_t:s0
```

In this example, here's what the various entities in the SELinux context stand for:

- system_u: is user
- object_r: role
- httpd_sys_content_t: type
- s0: level

The goal of sVirt is to protect the host server from a malicious instance, as well to protect the virtual instances themselves from a bad instance. The way sVirt does this by configuring each VM created under KVM virtualization to run with a different SELinux label. By doing this, it creates a virtual fence around each of the VMs, through the use of unique category sets for each VM.

It's common for Linux distributions to ship with Booleans that let you enable or disable an entire set of allowances with a single command, such as the following:

- virt_use_nfs: controls the ability of the instances to use NFS mounted file systems.
- virt_use_usb: controls the ability of the instances to use USB devices.
- virt_use_xserver: controls the ability of the instances to interact with the X Windows system

Linux Containers versus Virtualization (KVM)

Virtualization (such as KVM virtualization) and containers may seem similar, but there are significant differences between the two technologies. The most basic of the differences is that virtualization requires dedicated Linux kernels to operate, whereas Linux containers share the same host system kernel.

Your choice between virtualization and containerization depends on your specific needs, based on the features and benefits offered by the two approaches. Following is a quick summary of the benefits/drawbacks for the two technologies.

Benefits and Drawbacks of Virtualization

Assuming you're using KVM virtualization, you can run operating systems of different types, including both Linux and Windows, should you need it. Since you run separate kernel instances, you can assure separation among applications, which ensures that issues with one kernel don't affect the other kernels running on the same host. In addition security is enhanced due to the separation of the kernels. On top of this, you can run multiple versions of an application on the host and the VM, besides being able to perform virtual migrations as I explained earlier.

On the minus side, you must remember that VMs need more resources and you can run fewer VM on a host compared to the number of containers you can run.

Benefits and Drawbacks of Containers

Containers help you isolate applications, but maintaining the applications is a lot easier than maintaining them on virtual machines. For example, when you upgrade an application on the host, all containers that run instances of that application will benefit from that change. You can run a very large number of containers on a host machine, due to their light footprint. Theoretically speaking, you can run up to 6000 containers on a host, whereas you can only run a few VMs on a host.

Containers offer the following additional benefits:

- Flexibility: since an application's runtime requirements are included with the application in the container, you can run containers in multiple environments.
- Security: since containers typically have their own network interfaces and file system that are isolated from other containers, you can isolate and secure applications running in a container from the rest of the processes running on the host server
- Performance: typically containers run much faster than applications that carry the heavy overhead of a dedicated VM
- Sizing: since they don't contain an entire operating system unlike VMs, containers of course, are very compact, which makes it quite easy to share them.
- Resource allocation: LXC helps you easily manage resource allocations in real time.
- Versatility: You can run different Linux distributions on the same host kernel using different containers for each Linux distribution.

Linux Containers and KVM Virtualization – the Differences

The key difference between KVM virtual machines and Linux containers is that KVM VMs require a separate kernel of their own while containers share the same kernel from the OS.

You can host more containers than VMs for a given hardware, since contains have a light footprint and VMs are resource hungry.

KVM Virtualization lets you:

- Boot different operating systems, including non-Linux systems.
- Separate kernels mean that terminating a kernel doesn't disable the whole system.
- Run multiple versions of an application on the same host since the guest VM is isolated from changes in the host.
- Perform live migrations of the VMs

Linux Containers: **Are designed to support the isolation of applications.** Since system wide changes are visible inside all containers, any change such as an application upgrade will automatically apply to all containers that run instances of the application. ** The lightweight nature of containers means that you can a very large number of them on a host, with the maximum number running into 6000 containers or more on some systems.

Limitations of LXC

Unlike a fully virtualized system, LXC won't let you run other operating systems. However, it's possible for you to install both a virtualized (full or para) system on the same kernel as the LXC host system and run both the virtualized guests and LXC guests simultaneously. Virtualized management APIs such as libvert and ganeti are helpful if you wish to implement such as hybrid system.

Container benefits

As organizations move beyond monolithic applications to microservices, new application workloads involve a connected matric put together to server specific business needs, but easily rearrangeable into a different format. Containers are a key part of this new application architecture. For developers who create applications, containers offer these benefits:

- Better quality releases
- Easier and faster scalability of the applications
- Isolation for applications

- Shorter development and test cycles and fewer deployment errors

From the point of the IT operations teams, containers provide:

- Better quality releases
- Efficient replacement of full virtualization
- Easier management of applications When a container is instantiated, the processes execute within a new userspace created when you mount the container image. The kernel ensures that the processes in the container are limited to executing system calls only from their own namespaces such as the mount namespace and the PID namespace. The namespaces are containerized in this case.

You can take the same containerized image and run it on a laptop or servers in your datacenter, or on virtual machines in the cloud.

A virtual machine packages virtual hardware, a kernel and a user space. A container packages just the userspace – there's no kernel or virtual hardware in a container.

Linux Container Adoption Issues

Linux containers are starting to be used widely, with some large cloud service produces already using them at scale. Following are some concerns that have led to a slower than expected adoption of Linux containers.

- Security: security issues are a concern with enterprise adoption, due to the fact that kernel exploits at the host OS level will mean that all contains living on that host are at risk. Vendors are therefore fine tuning security techniques such as mandatory access control (MAC) to tighten things up security wise. SELinux already offers Mandatory access controls, and a different project named libseccomp lets you eliminate syscalls, which prevent hacked containers from compromising the kernel supporting them.
- Management and orchestration: vendors are working on creating frameworks for managing container images and orchestrating their lifecycle. New tools are being created for supporting containers. Docker is a great framework that makes it very easy to create, manage and delete containers. Docker can also limit the resource containers can consume, as well as provide metrics on how containers are using the resources. New tools for building and testing containers are in the wings as well. Linux container adoption will accelerate by agreeing on a standard for inter-container communications, using solutions such as virtual switching ng, hardware enhanced switching and routing and so on.

Managing Linux Containers

While SELinux, Cgroups and namespaces make containerization possible, a key missing piece of Linux containers is the ability to manage them – Docker solves this problem very nicely, as explained in the next chapter, which is all about Docker containers.

LXc is a userspace interface for providing containment features for the Linux kernel. It enables you to easily create and manage both system and application containers. It also helps you easily automate container deployment. LXC seeks to create a Linux environment that's close to the standard Linux installations, but without using a separate kernel. LXC containers are usually regarded as a midway solution between a chroot and a full-fledged virtual machine.

LXC is free software, with most of the code releases under the terms of a GNU license. While LXc, is quite useful, it does have some requirements such as the following:

- Editing of configuration files for controlling resources
- It (LXc) maybe implemented differently between distributions, or even among different releases of the same distribution

Docker offers a much more efficient and powerful way to create and manage Linux containers. Docker is an application that enables almost effortless management of Linux containers through a standard format. Docker isn't a totally new technology – it builds on the concepts you've seen earlier in this chapter, such as namespaces, cgroups and LXc to go beyond what's possible with userspace tools such as LXc. Besides helping you efficiently manage containers, Docker, since it is a type of image based containerization, makes containers portable, thus making it easy to share them across hosts.

With this useful background of Linux containers, let's turn to a discussion of Docker containers in the next chapter.

CHAPTER 5

Working with Docker Containers

Docker is the most well-known container platform that's increasingly being deployed for developing and running portable distributed applications. A Linux system administrator (or a developer) can use Docker to build, ship and run applications on various platforms, including the cloud, both on physical as well as virtual machines.

Chapter 4 explained the foundations of generic containers, which are Cgroups, SELinux and namespaces. Where Docker comes in is that it lets you easily package, run and maintain containers with handy tools, both from the command line and through HTTP APIs. This is what makes it possible for you to easily package applications and their runtime environments into self-contained images. A simple Dockerfile is what you use to package the application and its runtime environment together.

Few technologies are affecting and disrupting established ways of running applications as Docker, which provides a sophisticated way to package and run your applications. As a Linux administrator, you can get Docker up and running as a service in a few short minutes. Not only that, it's really easy to manage and monitor the containers that you run in a development or production environment.

At the center of modern containerization is the Open Container Initiative launched in 2016. The initiative is managed by the Linux Foundation, to create standards around container formats and their runtime environment.

A key reason for the wide spread success of containerization and Docker, especially, is that containerization lets you efficiently deploy applications in a cloud based environment. Docker has made it extremely easy to containerize applications. However, it's also important to understand that a lot of things are still evolving when it comes to

containerization – only about 40% of Docker's customers are actually running the containers in production.

Containers make it possible to "overwrite' a container that's part of a running application, which means less downtime when introducing changes.

Single containers are pretty easy to run, but managing a set of related containers is a complex affair. Fortunately, there are several tools such as Kubernates, Swarm and others that allow you to do precisely this, and I show how you can take advantage of these container orchestration technologies.

Containers isolate applications from other apps running on a host, but don't offer the same extent of isolation as VMs – since VMs are ahead of the curve in times of security, a lot of people run containers inside VMs, thereby getting the best of both the worlds, so to speak.

You can run Docker in a number of Linux systems, such as Ubuntu, Fedora, RHEL, and CentOS. Containerization means doing things in a small way and hence you really don't need a traditional bloated Linux system to run Docker. In addition to stripped down versions of the standard Linux distributions, you can run Docker on specially designed lightweight Linux systems optimized for container deployments, such as CoreOS and Atomic Host.

Containers include both the application code and its configuration and associated dependencies. Since the Linux operating system that supports these containers doesn't have to support all the app's dependencies any longer, stripped down container-oriented operating systems such as CoreOs Red Hat's Atomic Host are increasingly becoming popular as vehicles for running containers in production. I explain the CoreOS and Atomic Host operating systems later in this chapter.

Docker Basics

Docker containers package software in a complete filesystem that includes everything an application needs to run. This means the app will always run the same way no matter the environment. Docker containers offer the following key benefits:

- Lightweight – the images are built from layered file systems, so multiple images share common files, thus reducing the disk usage. They also share the same OS kernel, thus using RAM more efficiently.
- Open Standards: The open standard model means containers run on all Linux distributions.

- Security: Containers isolate applications from each other as well as from the infrastructure itself.

The main use of Docker is to create and run applications designed and configured to run within a container. Here, it's good to pause and understand what a containerized application is. Traditional applications run directly on a host operating system. They use the Linux server's file system, process table, network interfaces and ports and so on. The problem with this set up is that you can't run multiple versions of a software package on the same server, as this would cause conflicts. It's also hard to move an application since you'd have to move the code as well as its dependencies, which isn't easy to do.

When you create an application container, you want to make it easy to share the containers with others and distribute it. Docker is a container runtime engine that makes it easy to package applications and push them to a remote registry, from where other users can pull those applications.

Often people are a bit confused as to how Docker containers compare to traditional virtual machines. It's simple: a VM emulates a foreign environment while a container is a lightweight OS designed to make self-contained applications portable. Due to their extreme lightweight nature, applications running in containers incur very little overhead.

A key issue in developing and deploying applications is the packaging together of all the required dependencies for a software application, such as its code, the runtime libraries, etc. With Docker, you simply package all the required code and dependencies into a single Docker image. As explained in Chapter 4, unlike in traditional virtualization where the VMIs (virtual machine images) run on separate guest operating systems, all Docker images in a server run within a single OS kernel.

```
Docker is playing a significant role in the recent ascendency and popularity of DevOps. Administra
```

When Docker Isn't Right for You

Sometimes Docker isn't right for you! Although you can run virtually anything you want within a container, you really don't want to run databases and any kind of stateful applications within Docker containers. Docker containers are ideal for applications such as microservices which don't maintain state. Since resizing the CPU and memory of a container involves restarting the container, applications that need dynamic resizing for CPU and RAM aren't ideal for Dockerization.

Docker is more suitable for applications such as microservices that don't maintain state.

Docker uses iptables to provide NAT between the host IP and the container's IPs. As a result, if your applications need high network throughput, they aren't ideal for Docker.

What Docker Consists of

In order to get going with Docker containers, you need to learn about the various components that play a role in Docker containerization. The following sections explain:

- The Docker Project
- The Docker Hub Registry
- Docker Images and Docker Containers
- The docker Command

The Docker Project

The Docker Project, of course is what developed the container format that we know today as Docker. The project has the twin goals of simplifying application development and distribution. The Docker Project provides a format for software containers, and also provides various tools to help you manage, provision and orchestrate containers. The Docker Project also manages the central repository called the Docker Hub Registry.

The Docker Hub Registry

The Docker Hub Registry (*https://registry.hub.docker.com*) acts as a store for you to save and develop your Docker container images. When you request a Docker container image that's not on your system, Docker by default checks the Docker Hub Registry for that image.

You store containers in registries and you can make the images available to download to systems running Docker.

The Docker Hub Registry hosts the official repositories for all Linux distributions and several application projects. You can create a Docker user account and maintain your own Docker repository where you can push your Docker images to. In order to store container images privately, you can create your own Docker Registry.

The Docker Service

The Docker service is the same as the Docker engine. It's the Docker service that grabs the images you pull from the Docker Hub Registry.

Docker Images and Docker Containers

Before we dive deep into Docker containerization, it's highly useful to clarify certain commonly used terms such as Docker images, containers, and so on.

Docker Images

A Docker image consists of all the components of an application, such as libraries, executables and configuration files – together, these components enable an application to run.

You can build a Docker image through a Dockerfile, or by downloading pre-built Docker images from the Docker Hub (*https://hub.docker.com/*). The Dockerfile helps you automate the creation of a Docker container image. A Dockerfile consists of instructions regarding the software to be downloaded, the commands to be run, networks, environment variables, and the required files and directories that must be added to the container's filesystem.

Docker Containers

A Docker container is the running instance of a Docker image, and thus the instance is what performs the actual work. A Docker container is an isolated environment where the Docker image will run. Each Docker container will include its filesystem and you can copy files from the host server to the Docker containers.

Applications can require other software, and this means that Docker containers can be linked together, with software from one container made available to another Docker container. A good way to manage a container with other containers is to use a product called Kubernates to orchestrate the containers into pods.

A Docker image is a stored version of a container. A container is an instance of an image.

You start a container with the docker run command. You can run a container in the background (detached) mode, so the instance can keep running after the docker run command exits.

The Docker Command

The docker command is what you use to manage containers. You can run the command from the command line or run it as a service daemon that handles requests to manage containers.

There are separate commands for managing images and for managing instances. A command such as docker images will show you all the Docker images on your system. To view the containers that are actually running, you use commands such as docker ps. The docker start and docker stop commands let you start and stop a Docker container instance.

The docker command is quite versatile, and lets you do all the following:

- Create and remove containers and images
- Manage running containers
- Manage images and work with Docker registries
- Watch Docker events and log messages, and the CPU and memory usage statistics for containers

What Linux Administrators should know in order to support Docker

Docker offers a different paradigm from what most of us are used to, in terms of how various features of the Linux operating system are made available to Docker containers. In order to work with Docker, you should know the following:

- Host privileges
- Networking
- Storage

Host Privileges

Containers use host privileges to directly access a limited set of OS features such as the process table, the IPC namespace, specific CPUs, and devices. Super privileged containers are allowed to not only access, but also change the system.

You use regular containers for running applications, and super privileged containers to add tools to help you access the host system.

Docker Networking

You have to follow specific rules for managing the host network interfaces from inside the Docker containers. You can use ambassador containers (proxies) and service discovery solutions (explained in detail later in this chapter) to connect services running on different hosts. Since these solutions require you to expose ports through the hosts and also don't scale well. You may need to provide full cross-host networking solutions by providing IP connectivity between containers.

Following are the three commonly used Docker networking modes:

- Bridge: The Docker Bridge, called docer0 is used to connect containers. Docker instantiates a veth pair connecting eth0 in the container to the Docker bridge (docker0). It uses IP forwarding and iptables rules used for IP masquerading to provide the external connectivity. The bridge networking mode is good for development, but it's not very efficient, making it unsuitable for production purposes.
- Host: Under the Host networking mode, the container shares the host's networking namespace, thus exposing it to the public network. This is much more efficient than the bridge mode.
- Container: In this mode, a container uses the networking namespace from a different container. This mode allows for reuse of efficient network stacks but suffers from the drawback that all containers sharing a network stack need to use an identical IP stack. Kubernates uses this mode.

Earlier on, Docker used links to network containers, but the current way is for networks to be created and managed separately form containers. Two key objects – network and services play a key role in networking. When you launch a container, it can be assigned to a specific network. Containers can publish services that let them be contacted through their name, without needing the links.

For cross-host networking of clusters of containers, there are several networking solutions such as the following:

- Overlay: This is the "batteries included" Docker solution for cross-host networking, and is probably the best solution during development and for small cloud-based environments.. Overlay uses VXLAN tunnels to connect hosts with their IP namespace and for eternal connectivity, relies on NAT.
- Weave: This is a more complete solution for networking, and includes WeaveDNS for service discovery and load balancing. Weave is easy to use and hence good for development purposes.
- Flannel: Flannel is a networking solution meant mainly for CoreOS Docker containers. Flannel assigns subnets to the each of the hosts and the subnets are used to assign IPs to individual containers. In Kubernates, Flannel is used to assign

unique IPs to each pod. Flannel offers a good simple solution in many production scenarios.

Docker Storage

You can use bind mounts to connect the host storage to a Docker container. Docker storage volumes are normal Linux directories on the host server that are bind mounted to the container. The following example shows one way to initialize a volume. You bind mount a volume at runtime with the –v flag as shown here:

```
$ docker run -it -name  test-container -h CONTAINER -v /data/ Debian /bin/bash
```

This docker command will make the directory /data (inside the container named test-container) into a volume.

Alternatively, you can create a volume inside a container by specifying the VOLUME instruction in the container's Dockerfile:

```
FROM debian:wheezy
VOLUME /data
```

This does the same thing as the specifying of the –v option in the docker command I showed earlier.

Setting up the Docker Container Run-Time Environment

Now that you've learned the basic terminology used in Docker environments, let's get started with Docker containers by learning how to get them to run on a Fedora Linux distribution that's sponsored by Red Hat. In order to use Docker, you need to set up the Docker engine on a real or virtual Linux server. In order to get started with using Docker, all you need is a software package named docker, which contains the all-important docker command.

The best way to get started is by installing a standard Linux distribution with a desktop interface so you can develop and debug the containers you create. Once you're ready to go to production, you can deploy them through a container oriented OS such as CoreOS or Project Atomic.

Let's learn how to set up the Docker environment on a Linux CentOS server. Before you can install Docker, of course, you must download and install CentOS on either a real host or on a virtual machine (I'm using a virtual server that I created with Vagrant and VirtualBox). Once you do this you're ready to perform the Docker installation. On a RHEL based system, the Docker container's package is named docker. Here's how you perform the installation:

1. Update all the installed packages to their latest versions.

```
# yum update
```

1. Restart the server.

```
# reboot
```

1. Install the Docker package (this contains the docker command that you'll use to manage Docker containers).

```
# yum install docker
```

1. Start the Docker service. Before starting the service, you must enable the service.

```
# systemctl enable docker.service
# systemctl start docker.service
```

You're done at this point and you can check the status of Docker in the following way:

```
$ [vagrant@localhost ~]$ systemctl status docker.service
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; vendor preset: disabled)
   Active: active (running) since Sun 2016-03-13 13:44:25 EDT; 19h ago
     Docs: http://docs.docker.com
 Main PID: 881 (docker)
   Memory: 2.9M
   CGroup: /system.slice/docker.service
           └─881 /usr/bin/docker daemon --selinux-enabled
[vagrant@localhost ~]$
```

Note the following:

- The docker service is enabled and is active
- The service name is the docker command (/usr/bin/docker)
- The –d option means that the Docker service runs as a daemon process
- The Docker service is SELinux enabled.

Now that you've installed Docker, you're ready to play with it, using the docker command.

Getting Information about the Containers

You can execute the docker info command to verify that Docker is installed, and get a bunch of useful information, as shown here:

```
[root@localhost ~]# docker info
Containers: 19
Images: 73
Storage Driver: devicemapper
 Pool Name: docker-253:0-525784-pool
```

```
 Pool Blocksize: 65.54 kB
 Backing Filesystem: extfs
 Data file: /dev/loop0
 Metadata file: /dev/loop1
 Data Space Used: 5.878 GB
 Data Space Total: 107.4 GB
 Data Space Available: 31.66 GB
 /metadata
 Library Version: 1.02.107-RHEL7 (2015-12-01)
Kernel Version: 3.10.0-327.10.1.el7.x86_64
Operating System: CentOS Linux 7 (Core)
CPUs: 1
Total Memory: 993.3 MiB
Name: localhost.localdomain
…
[root@localhost ~]#
```

As with any Linux service, you can configure Docker to start on boot. On an Ubuntu server, for example, you configure the docker daemon to start on boot, by running the following command:

```
$ sudo systemctl enable docker
```

Running Container Images

Once you get going with Docker by starting the Docker service as shown earlier, you can run a container without having any images on your system. This seems like sheer magic when you actually do this on your system! When you execute the docker run command, it goes and gets the image you are looking for from the Docker Hub (by default). The following example shows how to get and run the fedora image on your server.

```
# docker run fedora cat /etc/os-release
Unable to find image 'fedora:latest' locally
00a0c78eeb6d: Pull complete
Status: Downloaded newer image...

NAME=Fedora
VERSION="22 (Twenty Two)"
ID=fedora
...
```

When you run the docker command, this is what happens:

- The command locates the container image (fedora in our example), and downloads it to your laptop or server. The command looks in your local server first for the fedora: latest image, and if it fails to find it there, will download the image from the docker.io registry (same as the Docker Hub Registry).
- The command starts the container using the image.

- The docker command will finally run the cat command, which you passed as an attribute to the docker command.

Once you run this command, Docker stores the image on your local system. A container, as I explain shortly, is ephemeral in nature and disappears off the face of the earth when you stop it. When you restart the same container, Docker uses the stored image on the local server. Thus, the very first run will take much more time since Docker needs to download the image from the Docker Hub – subsequent runs are quite fast, since all Docker needs to do is to spin up the container using the image that you already have.

Managing Containers

In a previous section where I showed the first example of the docker run command, the command started the Fedora container and executed the Linux cat command – and then it exited. Often, a container such as one that runs a web server service, for example, needs to run continuously. So, it's good to learn how to run a container in the background.

Running Interactive Containers

Use the –d (for detached) option with the docker run command, to run a container in the background. In order to run it in the foreground in an interactive fashion, specify the –i option and to open a terminal session, add the –t option.

A common use case for an interactive container is when you want open a shell to do things inside the container, just as you'd log into a Linux server to view, modify and execute various things. The following example shows how to open a shell in the Fedora image that Docker has already downloaded on to my server.

```
# docker run -it fedora /bin/bash
bash-4.3#
You can view the processes running inside this Fedora container by issuing the ps command:
bash-4.3# ps -e
  PID TTY          TIME CMD
    1 ?        00:00:00 bash
    7 ?        00:00:00 ps
#
```

Wow! Instead of the usual hundreds of processes that run on a full-fledged Linux server, this Fedora container has but two processes! The process with the ID 1 isn't the usual init (or systemd) process on your Linux server – instead it's the bash command. The only other process is the ps command you just used – that's it! This is the amazing thing with containerized systems – they're lean to start with and you can make them as lean or as fat as you want it, to suit your needs. This is the big difference between a container and a whole virtual machine.

If you're curious to see the list of installed packages on this new server, issue the familiar rpm –qa command:

Making your base Image Heftier

In the previous section, I showed how to create a very light Fedora container using a fedora base image. There are two ways to create a heftier container with more capabilities (more software) – you can simply download an alternative fedora base image with more stuff already baked in. Or, you can add software to a simple base image such as what I've got here. Docker lets you add software to a running container, as shown here:

```
bash-4.3# yum install iproute net-tools bsd-games words \
          vsftpd httpd httpd-manual -y
Resolving Dependencies
--> Running transaction check
...
Complete!
bash-4.3# exit
```

This example shows how you can add any software you want to a container. Use the yum or dnf (RHEL and Fedora) or the apt-get (Debian and Ubuntu) commands from within a container, in order to add software.

Committing a Container

If I want to use my fedora container again, with all the software I've just added, of course the container must have these software already added to it. I just exited the running container but didn't stop that container, so it's still running, as you can see here:

```
# docker ps -a
CONTAINER ID   IMAGE          COMMAND      CREATED
   STATUS                     PORTS   NAMES
88f6c09523b5   fedora:latest  "/bin/bash"  3 hours ago
   Exited (0) 7 seconds ago           trusting_heisenberg
#
```

You know that this running container has been fortified with additional software. Why not save the container as a new image that you can use whenever you wish? You can do this with the docker commit command, as shown here:

```
# docker commit -a "Sam Alapati" 88f6c09523b5 Myfedora
```

The new image Myfedora is now stored as a separate image on your server, and is part of the Docker image set:

```
# docker images
REPOSITORY  TAG     IMAGE ID      CREATED         VIRTUAL SIZE
```

```
fedora      latest  834629358fe2  1 month ago     422.2 MB
testrun     latest  226f7543f12a  3 minutes ago   431.6 MB
```

Running Commands within a Container

Now that I've added the ip-route and the net-tools packages to the bare bones base fedora image, I know I can run the ip and the route commands inside the container, as shown here:

```
# ip addr show docker0
5: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
       valid_lft forever preferred_lft forever
#
```

The ip command reveals the IP address of the host's docker0 interface – 172.17.42.1/16. The host automatically assigns IP addresses to each container as it's spawned, using DHCP to do so.

Containers use a router through the host server's docker0 interface to access networks outside the local server.

Linking Containers

You use Docker links to enable containers on the same host to communicate with each other. By default, communications among the containers in a Docker network aren't exposed to the host network, since they use an internal Docker network.

Running Services inside a Container

Earlier, you learned how to issue various commands from inside a container, depending on the software that's available in the container. However, simply being able to run those types of commands (ps, route, ip, etc) isn't going to do you a whole lot of good, if that's all you can do! The great power of containers comes through the ability to run a service such as a web server from the container.

Running containerized services lets you preconfigure everything you need to run that service and thus make the service truly portable. In addition, containerization of a service helps you to easily scale the service – if you need more web servers to manage the workload, simply start up more containers, that's it - all you need to do is to make sure that each of the container you expose as a service uses a different IP address and/or port.

Creating a Containerized Web Server

Let's learn how to run an Apache web server inside a Docker container. An Apache web server serves web content from the /var/www directory by default. Also by default, it listens on the TCP ports 80 and 443 (secure). In the following example, I have my web server serve the basic content (index.html) from the /var/www/html directory. Here are the steps to get this done:

1. Create a directory for holding the web content.

```
# mkdir -p /var/www/html
```

1. Set the appropriate SELinux context (since I'm using Fedora, which is a RHEL compatible system).

```
# chcon -R -t httpd_sys_content_t /var/www/
```

3. 4. Add some content to the index.html page.

```
# echo " Apache is running, yeah!!" > var/www/html/index.html
```

Now that you've prepared the ground for the Apache container, run the httpd service using a Docker image. I use the mycont image I had created in the previous section.

```
# docker run -d -p 80:80 -p 443:443 -name=MyApahceServer \
   -v /var/www/:/var/www mycont \
  /usr/sbin/httpd -DFOREGROUND
```

Note the following:

- In this example, the –d option tells Docker to keep the newly created container running in the background until I tell it to stop running it – remember that by default the container runs a command and quits! The -DFOREGROUND option runs the httpd daemon in the foreground.
- The –p option publishes the container ports to a host port. The port to the left of the colon belongs to the host and the one after the colon is the container port. So, in this example, I'm exposing both the TCP HTTP port (80) and the HTTPS (443) port to the same port numbers on the host server.
- The important option you should know about here is the –v option. The –v option shows the bind mounted volumes. Using this option, you mount directories on the host system to directories on the container. In this case, I mount the default Apache directory for web content (/var/www) on the host to an identically named directory on the container.
- The image name is mycont – the name of the container created from this image is specified by the –name parameter. The name parameter is optional, but quite

useful in practice. So, if you want to open a shell in the new MyApacheServer container, you can do so by issuing the normal docker command:

```
# docker exec -it MyApacheServer /bin/bash
```

Now, the web server is running in the Docker container. You can test that the ports are configured correctly by issuing a simple curl command such as this:

```
# curl http://localhost/index.html
```

I see the contents of the index.html file stored in the /var/www/html directory in the host server, since port 80 from the container is exposed to port 80 on the host, and by using bind volumes, the container also uses the same Apache directories as the host.

Controlling the Container Resource Usage

When you run multiple containers on a host, resource allocation becomes a significant issue. You can control the usage of the host resources such as RAM and CPU with options such as –memory, --memory-swap, --cpu-shares and –cpuset-cpus.

Running Privileged Containers

By default, a container has limited access to a host's capabilities. For example, it can't access namespaces such as the process table and the IPC namespace on the host. Although, in general, you want to limit a container's access to the host and to other containers, the concept of a privileged container (also called a super privileged container) lets you grant a container greater access than what's allowed by default.

Building Docker Images

You can build Docker images in two ways:

- A developer can build an image and push it to a repository
- You can automatically build images with a CI/CD system, following each code push

In a production system, the best strategy would be to use a CI/CD system such as Jenkins to automate the building of images following code pushes. As each container is built, it's uploaded to a repository from where the CI/CD system can download and run the image

Building Images with a Dockerfile

You can automate the creation of images through a Dockerfile, which is analogous to the Vagrant configuration file named Vagrantfile, which lets you configure VMs easily. The Dockerfile is a text file that consists of instructions, and the Docker

builder reads this file and executes the instructions one by one to create the image. 2.1.1 Let me build a simple image to show how to create and use a Dockerfile. Here are the steps:

First, create an empty directory

```
$ mkdir sample_image
```

Next, create a file named Dockerfile and add the following to the file.

```
# Pick up the base image
FROM fedora
# Add author name
MAINTAINER Sam R. Alapati
# Add the command to run at the start of container
CMD date
```

Finally, Build the image with the docker build command.

```
$ cd sample_image
$ docker build –t /fedora/test .
```

The –t option specifies a tag name.

As you can see, the docker build command saves the intermediate images so it can use them in later builds to speed up the build process. After running each of the instructions in the Dockerfile, Docker commits the intermediate state image and runs a container with that image for executing the next instruction. It then removes the intermediate containers from the previous step. Once the last instruction is executed, it creates the final image.

Understanding the Dockerfile

A Dockerfile has the following format:

- INSTRUCTION arguments: It's customary to specify the instructions in upper case. A line with the # at the beginning is a comment. A Dockerfile usually has the following types of instructions:
- FROM: This must be your first instruction in the file, and it sets the base image. The default behavior is the latest tag:

```
FROM <image>
```

You can have multiple FROM instructions if you want to create multiple images. If you just specify the image name, such as FROM fedora, the build program will download the image from the Docker Hub. If you want to use your own private images or a third-party image, you need to specify it as shown in this example:

```
FROM registry-host:5000/
```

- MAINTAINER: sets the name of the author for the image:

```
MAINTAINER <name>
```

- RUN: you can execute the RUN instructions in the shell, or directly as an executable:

```
RUN <command> <param1>… <paramN>
Or
RUN [“executable”, “param1”, “param2”, …. “paramN”]
```

- LABEL: Use the LABEL instruction to tag a distribution, as in:

```
LABEL distro=fedora21
```

- CMD: when starting a container, the CMD instruction offers a default executable:

```
CMD  ["executable", "param1",...,"paramN" ]
```

- ENTRYPOINT: This instruction lets you configure the container as an executable:

```
ENTRYPOINT  ["executable", "param1",...,"paramN" ]
ENTRYPOINT <command> <param1> ... <pamamN>
```

- EXPOSE: exposes the network ports on the container on which to listen at runtime:

```
EXPOSE  <port> [<port> ... ]
```

Alternatively, you can expose ports at runtime when starting the containers.

In addition, there are additional instructions such as those that let you set the environment variables (ENV), ADD (copy files into the image), and COPY (lets you copy files from source to destination). The docker build –help command shows all the possible instructions you can use.

Building an Apache Image using a Dockerfile

Let me use a simple example to show how to build a Docker image through a Dockerfile. You can get the Dockerfile by installing the fedora-dockerfiles package in a Fedora system (in the /usr/share/fedora-dockerfiles directory). Or, you can get the Dockerfile for this from the Fedora-Dockerfiles GitHub repo, as shown here:

```
$ git clone https://github.com/nkhare/Fedora-Dockerfiles.git
```

Once you clone the repo, go to the apache subdirectory and view the Dockerfile:

```
$ cd Fedora-Dockerfiles/apache/
$ cat Dockerfile
FROM fedora:20
MAINTAINER "Scott Collier" <scollier@redhat.com>

RUN yum -y update && yum clean all
RUN yum -y install httpd && yum clean all
RUN echo "Apache" >> /var/www/html/index.html

EXPOSE 80

# Simple startup script to avoid some issues observed with container restart
ADD run-apache.sh /run-apache.sh
RUN chmod -v +x /run-apache.sh

CMD ["/run-apache.sh"]
```

The run-apache.sh you see in the last three instructions in the Dockerfile refer to the script that runs HTTPD in the foreground.

Now that you've the Dockerfile, it's easy to build a new image:

```
$ docker build -t fedora/apache .
Sending build context to Docker daemon 23.55 kB
Sending build context to Docker daemon
Step 1 : MAINTAINER "Scott Collier" <scollier@redhat.com>
Removing intermediate container 2048200e6338
Step 2 : RUN yum -y update && yum clean all
.... Installing/Update packages ...
Cleaning up everything
Step 3 : RUN yum -y install httpd && yum clean all
.... Installing HTTPD ...
Step 4 : RUN echo "Apache" >> /var/www/html/index.html

Step 5 : EXPOSE 80

Step 6 : ADD run-apache.sh /run-apache.sh
Step 7 : RUN chmod -v +x /run-apache.sh

mode of '/run-apache.sh' changed from 0644 (rw-r--r--) to 0755 (rwxr-xr-x)
Step 8 : CMD /run-apache.sh

Successfully built 5f8041b6002c
```

The following is what our little docker build command has done for us:

- It installed the HTTPD package in our base image
- It created a HTML page
- It exposed port 80 to server the web age

- It set the instructions to start up Apache when you start a container based on this image

You can test your new image by running a container from it:

```
$ ID = docker  run -d -p 80 fedora/apache
```

Get the IP address of the container with the inspect command:

```
$ docker inspect -format='{{.NetworkSettings.IPAddress}}' $ID
172.17.0.22
```

Use the curl command to access the web page:

```
$ curl 172.17.0.22
Apache
```

Image Layers

Every instruction in a Dockerfile results in a new image layer. Each of these image layers can be used to start a container. The way a new layer is created is unique – it's created by starting the containers based on the image of the previous layer, executing the Dockerfile instruction, and saving the new image.

Multiple pre-built containers are layered together to build an application image.

Docker Image Repositories

You can store Docker images in a Docker image repository, or maintain your own image repository in your data center. Docker's hosted image repo hub isn't known for its reliability and hence, you're better off running your own image repository. It's much more reliable and faster as well to go this route, since there's very little network latency involved in the do it yourself approach.

Using a Private Docker Registry

By default, if a container image isn't already available on your own system, Docker fetches it from the Docker Hub Registry (*https:///hub.docker.com*).

You can also set up your own private Docker registry. Maintaining a private registry isn't by any means mandatory – however, it offers the option of storing your images privately without having to push them to the public Docker Hub Registry. Once you set it up, you can push and pull images from the private registry without having to use the public Docker Hub Registry.

In this section, I show how to setup a private Docker registry and how to use it to manage your Docker images.

Setting up a Private Registry

Some Linux distributions such as Fedora have a docker-registry package that lets you start up the service that runs the private registry. You can install this package and start the docker-registry service with the systemctl command. On other Linux distributions, once you've a Docker service running, as explained earlier in this chapter, you must run the Docker provided container image named registry to set up the service. Note that Atomic Project Fedora doesn't include the docker registry package – therefore, you must use the registry container for this distribution as well.

The Docker Image Namespace

In a "pure" Docker system (something that only uses the Docker Project code, with no modifications) a command such as docker run <image_name> will always pull the same image from the Docker Hub to your local system. Similarly, a Docker system that you haven't modified with any patches will search only the Docker Hub when you issue a docker search command, assuming the image isn't already in a private registry on the server.

At the present time, you can't change the default registry to something other than the Docker Hub Registry. However, changes are afoot to modify this default behavior, as well as several other aspects relating to the image namespace. You can certainly change the default behavior by taking advantage of features that are already present in various Linux distributions.

```
Note the following key facts about setting up a private registry:
```

- You can use the same registry for multiple Docker client systems
- By default, the Docker registry stores images in the /var/lib/docker-registry directory. Since a larger number of images can consume a lot of storage space, ensure that you've plenty of storage available in the host where you set up the registry.

Managing the Private Registry

Once you pull an image from the Docker Hub, you can push that image to the local Docker registry on a server. Before pushing it, name the image using the docker tag command:

```
# docker tag hello-world localhost:5000/hello-me:latest
```

Once you tag the image, push the image to the local Docker registry:

```
# docker push localhost:5000/hello-me:latest
The push refers to a repository [localhost:5000/hello-me] (1 tags)
...
Pushing tag for rev [91c95931e552] on
     {http://localhost:5000/v1/repositories/hello-me/tags/latest}
```

You can remove the image from the local private registry with the docker rmi command. You can retrieve an image from the private registry with the docker pull command. The following example shows how to do this:

```
# docker rm myhello
# docker rmi hello-world localhost:5000/hello-me:latest
# docker pull localhost:5000/hello-me:latest
Pulling repository localhost:5000/hello-me
91c95931e552: Download complete
a8219747be10: Download complete
```

The docker images command shows all the images stored in the local private registry on a server.

```
[root@localhost ~]# docker images
REPOSITORY                          TAG       IMAGE ID       CREATED             VIRTUAL SIZE
docker.io/fedora                    latest    7427c9af1454   9 days ago          204.7 MB
docker.io/ubuntu                    latest    56063ad57855   10 days ago         187.9 MB
docker.io/hello-world               latest    975b84d108f1   5 months ago        960 B
docker.io/sequenceiq/hadoop-docker  2.7.0     ea842b97d1b8   10 months ago       1.76 GB
[root@localhost ~]#
```

In an Ubuntu system, you don't install the Docker registry from a package – instead, you download the registry container from the Docker Hub Registry. You can pull the registry image from the Docker Hub with the docker pull command as shown here:

```
# sudo docker pull registry:latest
```

The rest of the commands to test the creation of the registry, to pull and push images are identical to those I described above for the Fedora system.

New Operating Systems Optimized for Docker

Earlier on, I showed how you run Docker in a traditional Linux distribution such as Fedora or Ubuntu. The current trend however, is to use a new generation of operating systems that have been expressly designed for Docker. Two things separate these operating systems from traditional systems:

- Both of these operating systems are preconfigured to run Docker
- Only containers are expected to run on the server (pull an image and run a container)
- They use an atomic upgrade mechanism to manage the servers

The three features mentioned here mean that these operating systems are lightweight in nature – they contain just the absolute minimum capabilities to run a container. The systems don't have tools such as yum or apt-get to download packages separately from the containers. In this section I explain the following lightweight container oriented operating systems:

- CoreOS – You can install CoreOS, available on most cloud providers, on bare-metal and test it locally through Vagrant (to be explained in Chapter 6). Atomic Hosts: These include servers such as CentOS Atomic, RHEL Atomic and Fedora Atomic.
- Atomic Host: Red Hat Enterprise Linux Atomic host is a variation of Red Hat Enterprise Linux 7 that's optimized for running Linux containers in the Docker format.

CoreOS and Atomic Host aren't everything that's available to you – you can also use other operating systems and other techniques such as RancherOS, and VMWare Photon.

Using CoreOS

CoreOS is a popular new Linux distribution that's geared to running applications within containers. CoreOS is highly scalable and is an easy to manage operating system, which provides a distinct marking off of operational and application related functions.

Both of these lightweight operating systems let you deploy containers on a bare metal system, or in a cloud environment such as Amazon EC2 or the Google Cloud Platform.

CoreOS offers an ISO that lets you copy its image to a partition and boot it up with no sweat. If you're using CoreOS to deploy to a cloud environment, you can use a specialized tool such as cloud-config.

You can create a cluster of CoreOS servers and launch containers in that cluster using schedulers. Flannel is a network overlay technique that's a part of CoreOs, and serves as the way containers can communicate through a provider IP space across multiple servers.

Working with Atomic Host

Atomic Host is the product of the Atomic Project and is an RPM-based Linux distribution builder expressly designed to work with Docker containers. Fedora, RHEL

and CentOS versions can run as Atomic hosts. Depending on your environment, you can set up an Atomic Host in any one of the following ways:

- Download an Atomic qcow2 image and add configuration information to it with the cloud-init tool.
- Use a Vagrant file to spin up a CentOS Atomic VM (for CentOS Atomic)
- RHEL Atomic offers the Atomic Installation ISO and Fedora offers the Fedora 22 installer. Once you download and install the Fedora Atomic ISO installation image (assuming you want to setup a Fedora Atomic Host), you can start up the ISO image and start installing it. Once you complete the installation, run the following atomic command to ensure you have the latest Docker version.

```
# atomic host upgrade
```

Once you do this you are ready to set up a Docker registry, or start running docker commands.

The Docker Stack in Production

As with a virtualized environment, setting up Docker in a production system involves several architectural components, each of which performs a specialized task, such as building images and running the images as containers. Following are the typical tasks involved in managing a Docker system in production:

- Building the image
- Sending the image to a repository
- Downloading the image to a new host
- Running the image as a Docker container
- Connecting the container to other containers (clusters of containers)
- Sending traffic to the containers
- Monitoring the running containers in production

In order to manage these tasks, you must set up architectural components such as the following:

- A build system
- An image repository
- A configuration management system
- A deployment mechanism
- An orchestration system to tie multiple containers together

- A monitoring system

The following principles help you move containers to a production use from development:

- Keep production and development environments as similar as possible
- Make sure to employ a continuous integration/continuous delivery (CI/CD) system
- Keep the Docker setup simple
- Automate as much as possible

It's normal to use the term deployment when you want to get something into production. Deployment involves the building and testing of Docker images and deploying the official tested images to the production server(s). Once the image is on the server, of course, you must run the containers on the server. However, in order to carry this off on multiple servers, you must use a repeatable process that can handle the container configuration during each deployment.

Provisioning Resources with Docker Machine

The best way to providing new resources to run containers is to use Docker Machine. Docker Machine can do all the following:

- Create the servers
- Install Docker on the new servers
- Configure local Docker clients to access the new containers

In order to deploy at scale, the simple Docker client won't cut it – you need to have an orchestration tool to manage the complexities of talking to different servers and coordinating the container configuration in different environments. While you can create scripts to talk to the Docker daemon's Docker Remote API, it's somewhat like reinventing the wheel. A lot of work has already been done for you as regards orchestration, and you can take advantage of various approaches to handle your deployment needs. Two basic approaches here would be to use orchestration tools or to use distributed schedulers.

Docker Orchestration

Deploying large number of containers involves a lot of decisions and a lot of work. You need to do some or most of the following as part of deployment:

- Organizing containers into clusters

- Scheduling the server resources
- Running the containers
- Determining the hosts where the containers should run
- Routing traffic to the containers
- Allowing the containers to expose and discover services

Container orchestration tools such Kubernates provide most or all of the services listed here. Besides Kubernates, you can also use Docker Swarm, Mesos and Flannel for orchestration. Once of the big questions you must answer is which of the orchestration tools you must use – I shouldn't really use the word "must" here, so, let me modify the preceding sentence this way: which of the tools you may want to use – since orchestration tools add to your work as well, and small teams may not really need them.

Docker Orchestration and Clustering Tools

It's ridiculously simple to create Docker images and run Dockerized containers for various services. However, the real benefit and the real fun, so far as administrators are concerned, is when you run a Docker cluster, consisting of a large number of containers together. An easy way to jump into production with Docker containers is to use an orchestration tool. An orchestration tool coordinates the configuration and deployment of applications to multiple Docker containers simultaneously. Orchestration tools require minimal modifications to your system and you can get to your goal of zero-downtime deployments quite easily with these tools.

The most popular orchestration tools for Docker are Centurion, (New Relic), Helios (Spotify) and a Docker tooling set from Ansible, the popular configuration management system.

Distributed Schedulers for Docker Containers

Distributed schedulers also allow for zero downtime deployments, by letting you run the old and new application versions side by side until you're ready to migrate fully to the new versions. You simply specify policies regarding how you want to run your applications, and the scheduler figures out where to run it, and restart failed services on default containers.

There are several distributed schedulers you can consider, including the native Docker clustering tool named Swarm, which lets you create and manage coordinators. Google's Kubernates is quite popular in this area, and there are other alternatives such as Apache Mesos and Fleet from CoreOS. Let's take a quick look at the various distributed schedulers in the following sections.

Using Docker's Swarm as a clustering tool

Docker Swarm is much simpler than a powerful scheduling tool such as Apache Mesos or Kubernates, but allows you to create, deploy and manage containers across fairly large Docker clusters. Swarm doesn't really focus on configuration of the application or deployments, since its main purpose is to cluster the computing resources so Docker can use them.

When you use Swarm, the Docker client will still see a single interface, but that interface will be representing a cluster of Docker containers rather than just a lone Docker daemon as you've seen earlier in this chapter. You deploy Swarm as a single Docker container. Through this container, Swarm manages the Docker cluster. You deploy the Swarm container as an agent to each Docker host. This will allow you to merge all your hosts that run Docker containers into a single cluster.

You can install Docker Swarm manually or through the Docker Machine tool. When first starting out, the recommended approach is not to install Swarm via the manual method. It's far easier to install Docker Swarm using Docker Machine. Docker Machine also automatically generates he required TLS certificates to secure Swarm.

Setting up Swarm for Production Use

For production usage, administrators can manually create Swarm on a network by first pulling the Docker Swarm image. You can then configure the Swarm manager and all nodes to run Docker Swarm, all through using Docker. In order to do this you must install Docker on all nodes. In a production setting, you'd need to create a Swarm cluster by running multiple (at least two) Swarm managers in a high-availability configuration.

You can build a Swarm cluster within your data center or on the cloud. Let's say you want to deploy the Swarm cluster on the Amazon Web Services (AWS) platform. Here's an outline of how you can build a high-availability Swarm cluster for production.

1. Add network security rules to the AWS security group so required network traffic is allowed on the VPC network.
2. Create multiple Linux hosts that are part of the default security group. Let's say you create five hosts – of the five hosts, the Swarm primary and secondary managers would be nodes 1 and 2. The Swarm node would be on nodes 3 and 4. The discover backend would be on node 5, and you'll run a consul container on this host.
3. Install the Docker Engine on all the nodes, which enables the Swarm manager to address the nodes.

4. Set up a discovery backend, which helps the Swarm manager and nodes to authenticate themselves as cluster members. The discovery backend also helps the Swarm manager learn which nodes are available to run containers.
5. Create the Swarm cluster – this means you create the two Swarm managers in a high availability configuration. The first manager you start will be the primary manager and the second manager, a replica. When the primary manager fails, the replica becomes the primary manager.

Interacting with the Docker Cluster

Now that you've set up Swarm to manage the Docker container, you can interact with the entire Swarm–managed Docker cluster instead of a single host. Just set the DOCKER_HOST environment variable to point to the Swarm manager so you can run Docker commands against the Docker cluster.

Swarm has the advantage that it uses the standard Docker interface and this is easy to use and integrate into your current workflows. It's not designed however, to support complex scheduling. Fleet uses etcd to enable communications between machines and to store the cluster status.

How Fleet can help you

Fleet comes from CoreOS, the lightweight container operating system, and is designed to form a foundation or base layer or more advanced clustering solutions such as Kubernates.

Fleet is built upon systemd and extends the system and service initialization capabilities of systemd to a cluster of machines.

Apache Mesos and Marathon as a Cluster Manager

Apache Mesos is a cluster manager that can scale to cluster of hundreds or thousands of nodes. Mesos is unique in that it can support diverse workloads – both Docker containers and Hadoop jobs can coexist for example. High availability and resilience are the main benefits offered by Mesos. Mesos is a low level scheduler and supports various frameworks for orchestrating container, such as Marathon, Kubernates and Swarm. Mesos can support very large systems with thousands of nodes, but may be an overkill for small clusters of just a handful of nodes.

The Big Heavy – Kubernates

Kubernates is a Google-built container orchestration tool based on Google's production experience with containers. Kubernates uses the concept of pods, which are groups of containers that you schedule and deploy a unit. Each pod, consisting of up to five containers, provides a service. Kubernates uses other containers besides the

service providing pods, to provide logging and monitoring services. Kubernates isn't ideal for all applications, but for microservices with little state to maintain, it offers a scalable service with little configuration work involved.

Docker Containers, Service Discovery and Service Registration

System architectures are changing rapidly over time, with computing environments becoming ever more dynamic in nature to support newer architectures such as service–oriented architectures and microservices, which we discussed earlier in Chapter 3. If you think a really amazing infrastructure such as Amazons' EC2 (Chapter 9) is dynamic, wait until you learn about Mesos (Chapter 16), which is an even more dynamic computing framework,

Docker containerization has contributed to a much more dynamic computing environment. Modern computing environments involve a large number of hosts and there are many services running on these hosts. Services are taken off and brought back online continuously, making service discovery a critical part of architecting applications.

Ideally, regardless of the size of your environment, you should be able to automate the service discovery process and minimize the effort involved in configuring and connecting this multitude of services. Configuration management tools are designed to solve a totally different problem altogether and they don't scale very well when you tack on the responsibility of service discovery to their main functions.

The fundamental idea that underlies service discovery is that new instances of applications (clients of services) should programmatically be able to easily and automatically identify the current environment details, so they can connect of a service without manual intervention. All service discovery tools are implemented as a global registry that stores information about currently operating services. The registry is usually distributed among multiple hosts to make the configuration fault tolerant and scalable.

Service discovery lets client discover service instances and networking takes care of making the connections work – that is, it connects containers together.

Service discovery platforms are designed to service connection details, but they are based on key/value stores, which means they can store other types of data as well. Many deployments piggyback on the service discovery tools to write their configuration data to the service discovery tool.

In the fast proliferating container and microservice world, services are distributed in a PaaS architecture and an infrastructure that's supported by containers (or VM images) is immutable. Predefined metrics determine whether services will be scaled up or down based on the workloads and metrics. In this environment, the address of a service is dynamically assigned of course, and isn't available until the service is deployed and ready for use. It's this dynamic setting of the service endpoint address that is at the heart of modern service registration and discovery.

Service discovery is the automatic provisioning of the connection information of a service to the clients of that service.

Service registration and discovery works essentially in the following manner: each of the dynamic services registers with a broker, to whom it provides various details such as its endpoint address. Services that want to consume this service will then query the broker to find out the service location and invoke the service. The broker's function is to allow registration by the services and the querying of those services. You can use various brokers such as ZooKeeper, etcd, consul, and Kubernates (Netflix Eureka as well).

Connecting Containers through Ambassadors

In a production network, you can connect containers across hosts through the use ambassadors, which are proxy containers. Ambassadors just forward traffic to the actual services. Ambassadors are simple containers that setup connections between an application and a service (such as a database service). Ambassadors offer the benefit that you don't need to change your development code to make it work in a production environment. System administrators configure the applcition to use different clustered services without required code modifications.

In a development setup, developers use Docker links to link the application to a data base container. In a production environment, operation teams can use an ambassador to link the app to the production database service. Operations configure the ambassador so the traffic flows through it and use Docker links to connect the app to the ambassador.

Ambassadors do require extra work in configuring them and are potential points of failure. They also tend to be complex and when you require multiple connections, can overwhelm operations teams. In order to scale, it's better to use networking and service discovery solutions to discover and connect to remote services.

Let's trace the evolution of service discovery and service registration in the following sections, starting with the concept of zero-configuration networking, which was the initial step in the effort to automate the discovery of services.

How Service Discovery Works

Service discovery tools (to be discussed in the following sections) provide APIs that application components use to set or retrieve data. The discovery service is implemented as a reliable, distributed key-value store accessible through HTTP methods. Thus, the service discovery address for each component must either be hardcoded in the application, or supplied as a runtime option.

When a service comes online, it registers itself with a service discovery tool such as ZooKeeper, etcd or consul (all three to be introduced later). The service also provides the information that other services and application components would require in order to consume the services provided by the service that is registering itself. For example, an Oracle database will register the IP address and port for the Oracle Listener service, and optionally the user credentials to log into the database.

When another service that consumes the services provided by the first service comes online, it queries the service discovery registry for information and goes on to establish connections to those services. For example, a load balancer can query the service discovery framework and modify its configuration so it knows exactly across which set of backend servers it should distribute the workload.

Most service discovery tools provide automatic failure detection. When a component fails, the discovery service is automatically updated to indicate that the service is unavailable. Failure detection is usually done through regular health checks and periodic heartbeats from the components. The service discovery tools also use configuration timeouts to determine if components should be yanked out of the data store.

Zero-Configuration Networking

Zero-configuration networking uses several network technologies to create networks consisting of connected devices based on TCP/IP without requiring administrator intervention, or even specialized configuration servers. DNS allows resolution of host names and the Dynamic Host Configuration Protocol (DHCP) automatically assigns hostnames. Zeroconf takes things further: its automatically assigns network addresses, distributes and resolves the hostnames, and locates various network devices, all without DHCP and DNS.

Zeroconf has eventually led to the idea of service discovery, which is our next topic.

Service Discovery

Service discovery tools enable processes and services in a cluster to locate and talk to one another. Service discovery has the following components:

- A consistent and highly available service directory that lists all services
- A mechanism to register services in the directory and also monitor the service health
- A mechanism to locate and connect to services in a directory

Service discovery depends on identifying when processes are listening on a TCP (or UDP) port and then identifying and connecting to that port by name.

But, Doesn't' DNS Already Do This?

One's first thought might be, "but DNS already looks up hostnames, no?" That's most certainly true, DNS does perform name resolutions but DNS was never meant for systems with real-time name resolution changes. DNS was primarily designed for an environment where services are assigned standard ports such as the well-known port 80 for HTTP, port 22 for SSH, and so on.

DNS can give you the IP for a service host, and that would suffice in these environments. In today's modern environments, services often use non-standard ports, with multiple services running together on a server. How do you discover these services automatically)? This is the question that service discovery tools address.

DNS partially addresses the modern service discovery concerns through its service records (SRV), which provide both the port and the IP address in response to queries. Unfortunately, APIs and libraries can't do SRV record lookups, so DNS becomes the old simple DNS that can only resolve hostnames to IP addresses.

Distributed Lock Services

Around 2006, Google engineers created a distributed lock service named Chubby and started using it for internal name resolution instead of DNS. Chubby implemented a distributed consensus based on Paxos, an algorithm for ascertaining the consensus opinion among the members of a cluster), and is a key-value store that was used for locking resources, and coordinating leader elections. Paxos, however, isn't easy to implement, and eventually a more sophisticated tool named ZooKeeper (Out of the Apache Hadoop project) took over as the standard distributed lock service.

Zookeeper

ZooKeeper is a highly available and reliable distributed lock service that coordinates distributed systems. ZooKeeper has become the industry standard for coordinating

distributed systems and many projects such as Apache Hadoop, Apache Storm, Apache Mesos and Apache Kafka depend on ZooKeeper for distributed coordination, which is critical to their functioning. The fact that ZooKeeper is a highly available key/value data store as well made it a candidate for storing the cluster configuration and for serving as a directory of services.

ZooKeeper is a centralized service that can maintain configuration information, naming, provide group services and synchronization of distributed applications. Services can register with ZooKeeper using a logical name and the configuration information can include the URI endpoint and other information such as QoS (quality of service, discussed in Chapter 2).

ZooKeeper is among the most popular service discovery mechanisms employed by microservice architectures, so let's get a quick overview of this service. Here are essential concepts of ZooKeeper:

- Znodes: ZooKeeper stores data in a hierarchical namespace that consists of data registers called znodes, which are similar to files and directories.
- Ensemble: Multiple ZooKeeper instances running on separate servers are together known as an ensemble. The instances are aware of each other and must be odd numbered, meaning a set such as 3, 5, or 7 servers. The odd number requirement for the instances is to ensure there's a quorum when decisions are made for selecting a leader.
- Node Name: each node in a ZooKeeper namespace is identified by a path and the node name is a sequence of path elements.
- Configuration Data; Each node in a ZooKeeper namespace stores coordination data.
- Client and Server: Distributed clients connect to a single ZooKeeper Server instance, of which there are several in every distributed environment managed by ZooKeeper. The client maintains a TCP connection through which it sends heart beats and requests and receives responses from the server instance. If the connection to a ZooKeeper instance breaks, the client automatically connects to a different instance.

When you use a service such as ZooKeeper for service discovery and registration, each service in an application (for example the Catalog and Order services in a sales related service) registers and unregisters the service as part of its lifecycle initialization.

ZooKeeper is the most mature of the service discovery tools, but lacks several sophisticated features offered by newer tools such as etcd and consul.

Etcd and Consul

While ZooKeeper is pretty easy to install, configure, and manage from an administrator's perspective (it's the only coordinator used by Apache Hadoop clusters, which use it to support high availability), it's quite heavyweight, and requires highly sophisticated developmental efforts to implement it so it can manage service discovery.

Recently, a much simpler consensus algorithm, Raft, has come into being as an alternative to the Paxos algorithm. CoreOS engineers have used the Raft algorithm to come up with etcd, a distributed key-value store for distributed systems similar to ZooKeeper, written in the Go language. Etcd is a highly reliable key-value store for storing the most critical data of a distributed system.

Consul is another new distributed service discovery and configuration tool that makes it simple for services to register themselves and discover other services. In addition to service discovery and registration, Consul also focuses on failure detection through regular heath checking that prevents routing requests being made to unhealthy hosts. Consul also provides several advanced features such as ACL functionality and HAProxy configuration. Although Consul contains a DNS server, you can use the SkyDNS utility to provide DNS-based service discovery when you use etcd.

Besides ZooKeeper, etcd and Consul, there are also a large number of projects that build on basic service discovery, such as Crypt, Confd, Eureka, Marathon (mainly a scheduler), Synapse and Nerve. One can also "roll their own", if they require more features than what's offered by the available service discovery tools out there.

Service Registration

When you use SkyDNS and Consul, you need to perform registration, which is the final step of service discovery, by explicitly writing code to register a services (such as a Redis database service) with SkyDNS and Consul, for example. Or, the Redis containers can have logic to automatically register upon start. You can however, use a service that automatically registers containers upon their start by monitoring Docker events. The Registrator product from GliderLabs works with Consul, etcd an, or SkyDNS to automatically register containers. It performs registration by monitoring Docker event streams for container creation.

CHAPTER 6

Automating Server Deployment and Managing Development Environments

This chapter focus on automating server deployment and managing development environments.

I start off by discussing Linux package management tools and then move on to Fully Automatic Installation (FAI), which is the way to go when installing large number of Linux servers.

Setting up consistent virtual environments is a big concern in many organizations. Vagrant is an amazing tool that's quite easy to use yet very powerful, and one that helps you effortlessly spin up consistent development environments.

Managing a few servers at a time through shell scripts is fine, but when handling large and complex environments with various services running on them, you need different strategies. I discuss various tools that'll help you perform parallel command execution such as PDSH, as well as more sophisticated parallel execution frameworks such as Fabric and Mcollective.

Automated server provisioning tools are very helpful in installing large number of servers and managing them with configuration tools such as Chef and Puppet (I discuss these two well-known configuration management tools along with other popular CM tools such as Ansible/Salt) in Chapter 7.

The chapter concludes with a brief discussion of two popular server deployment automation tools: Razor and Cobbler.

Linux Package Management

Linux systems can contain thousands of software packages and adding, updating and removing those packages is a common task for system administrators. Linux software packet managers are essentially commands that you can run to install software binaries on a Linux server by fetching the binaries from a binary repository. Red Hat compatible systems use a package format called RPM (Red Hat Package Manager), with package installers that end in .rpm, and Debian based systems use the DEB (.db) format.

Using the rpm and dpkg commands

You can install software packages with the rpm command in a Red Hat based system, as shown here:

```
# rpm -ihv nmap-6.40.4.e17.x86_64.rpm
You can remove installed RPMs with the -e option:
# rpm -e nmap
```

On a Debian system you can add packages with the dpkg –i command and remove them with the dpkg –r command.

Although the rpm and dpkg commands work fine and do what they're designed for, they're quite problematic when installing packages that have dependencies. In such cases, when you try to install package abc, you're prompted to first install a required package such as xyz, which in turn may ask you to install another required package and so on. Package managers are utilities that handle all the dependencies for you and make installing and removing software a breeze.

Red Hat systems use the YUM (Yellow Dog Linux Updated, Modified) utility, invoked with the yum command. Debian systems on the other hand use the utility named APT (Advanced Package Tool), which you invoke with the apt command. Let's learn a bit about these tools next.

Why use a packet management system? The big benefit is that the package manager takes care of all software dependencies, installing any dependent packages automatically – this makes installing and upgrading packages a breeze.

In the absence of more sophisticated deployment tools, you can log into a server and use yum or aptget commands to install or upgrade software. The yum upgrade command for example will fetch all the latest packages from the binary repository and install them for you.

Package Management with YUM and APT

The YUM utility is a package manager for RedHat Linux systems. It helps you check for and automatically download and install the latest RPM packages. The YUM pack-

age manager also obtains and downloads the dependencies, automatically prompting you as needed.

Here are some examples that show how to use YUM by invoking the yum and the apt commands.

```
# yum install puppet
# yum remove puppet
Yum update      # upgrades all software on a server
# yum list available | grep puppet        # query list of packages available through YUM
# apt-get install puppet                  # installs Puppet
# apt-get remove puppet                   # remove the Puppet software
# apt-get update                          # sync the server's softare with APT repositories
# apt-get upgrade                         # upgrades all installed software that's different # apt
                                          # /var/cache/apt/archives directory
```

Fully Automatic Installation (FAI)

Installing Linux on dozens or hundreds of servers at a time is a task that you must automate. Manual installs don't scale, and are error prone to boot (no pun intended). Fully Automatic Installation (FAI), which is probably one of the oldest such systems, lets you install a Debian Linux OS unattended on one or more servers. FAI is non-interactive and lets you install and customize Linux systems on physical machines as well as virtual servers. Internally FAI uses a set of shell and Perl scripts to get the job done.

The key to using FAI, which is a way to perform a network installation, is the PXE – Pre-Execution Environment. The PXE server acts as a network boot server. Network installation with an installation server lets you install Linux on a set of systems using the network boot server. By doing this, all the systems you configured to boot using an image provided by this serer will do so, and will automatically start the installation program.

You don't need physical boot media to plug into the client in order to start the Linux installation. You can install Linux on multiple systems over the network using the network boot server. Let's therefore learn how to set up a PXE server first.

How FAI Works

FAI can install Linux OS not one or a very large number of machines. Here are the key components of FAI:

- Install server (also called the faiserver): runs the necessary DHCP, TFTP and NFS servers and provides configuration data for the clients on which you want to install a Linux OS.

- Configuration space: a subdirectory where you store the installation configuration files. The files include information about the hard disk layout (similar to the fstab file), software packages, time zones, user accounts, printers, etc.
- NFS-Root: a file system on the faiserver that serves as the complete file system for the target servers.

What you need for a Network Installation

You'll need a server that runs the following:

- A DHCP server to assign IP addresses
- A TFTP server to server the boot files
- An HTTP, FTP or NFS server to host the installation image

How it Works

As mentioned earlier, under a network installation, you don't need any physical boot media to be plugged into the client servers on which you are installing Linux. When you begin the installation, this is what the client servers do:

- Query the DHCP server
- Get the boot files from the TFTP server
- Download the Linux installation image from a HTTP, FTP or NFS server, depending on which server you're using.

In order to configure a network installation, you must configure the network server that holds the package repositories that are needed for the installation. Next, you need to configure the PXE server.

Setting up the Network Server

As mentioned earlier, you can choose NFS, HTTP (or HTTPS), or FTP to export the installation ISO image or the installation tree from the network server to the clients.

Once you set up the network server, you'll have made the ISO image accessible over NFS for clients to use as a network based installation source. The next step would be to configure the PXE server.

Setting up the PXE Server for a Network Installation

The PXE server contains the necessary files to boot the RHEL and start the network installation. In addition to the PXE server, you must also configure a DHCP server

and enable and start all necessary services. You must configure the PXE server by performing the following tasks (on a RHEL system):

1. Install the tftp package
2. Modify the firewall so it accepts incoming connections to the tftp service:
3. Configure the DHCP server to use the boot images packaged with SYSLINUX.
4. Extract the prelinux.o file from the SYSLINUX package in the ISO image file of the installation DVD.
5. Under the tftpboot directory, create a pxelinux/ directory and copy the prexelinux.o file into that directory. Under the prxelinux/ directory, create the pxelinux.cfg/ directory and add the configration file named default to that directory.
6. Copy the boot images under the tftp/ root directory. # cp /path/to/x86_64/os/images/pxeboot/(vmlinuz,initrd.img) /var/lib/tftpboot/pxelinux/
7. Start (or reload if they're running) the xinetd and dhcp services. # systemctl start xinetd.service dhcpd.service

The PXE server is now ready to start the network install. Start the client on which you want to install Linux and select PXE Boot as the boot source and start the network installation. This will start the installation program from the PXE server.

Using Kickstart

For RPM-based distributions such as Red Hat Linux and Fedora, using Kickstart rather than FAI is probably the better approach to perform automatic installations for RHEL and CentOS7 systems without out the need for user intervention. You still use the PXE-boot to get the servers started, but the configuration information is provided by Kickstart files read from a local FTP server rather than FAI.

In order to perform a Kickstart installation, you configure the PXE server in the same way as shown in the previous section – all you need to do is to add the Kickstart file to the mix. You thus use the PXE server together with Kickstart in this case.

.Creating a Custom Kickstart File

You can deploy RHEL and CentOs simultaneously on a large number of servers by using a Kickstart installation. The Kickstart files contain all the answers to the interactive actions that are posed by the installation program, such as the disk partitioning schemes and the packages the installer must install. Kickstart files thus help you automate the installations when dealing with a large number of server deployments by letting you use a single Kickstart file to install RHEL on all the machines.

In the following sections, I briefly explain the steps involved in performing an automatic RHEL Kickstart installation.

Creating a Kickstart File

It's best to manually install the RHEL Linux software (in in this case RHEL 7) on one system first. The Kickstart file is a simple text file and you can name it anything you want. The choices you make during this manual selection are stored in the news server in the /root/anaconda-ks.cfg file – this will serve as your Kickstart file for automatic installations.

You can modify the Kickstart file as you please. ON RHEL, there's also a Kickstart Configuration Tool available, that lets you walk through server configuration and create and download a Kickstart file – the only drawback is this file won't support advanced disk partitioning.

Verifying the Kickstart File

Before you can use the Kickstart file to automatically install the Linux binaries on a whole bunch of machines, it's a good idea to verity it's validity using the ksvalidator command line utility, which is part of the pyKickstart package.

Here's how you do it:

```
# yum install pyKickstart
# ksvalidator /pth/tp/Kickstart.ks
```

Making the Kickstart File Available

In order to make the Kickstart file available to the client machines, you must place it on a removable media (DVD or USB drive), or on a hard derive connected to the client server, or even better yet, a network share that the client machine can access. Since normally the new systems boot using a PXE server, it's probably a good idea to let the clients also download the Kickstart file from a network share.

Making the Installation Source Available

The Kickstart file contains a list of software packages required for the install. The installation process must then access an installation source such as a RHEL installation DVD ISO image for an installation tree to install those packages. Assuming that you're performing a network based (NFS) installation, you must make the installation tree available over the network, using the steps I described in the previous section.

An installation tree is a copy of the binary RHEL DVD with an identical directory structure.

Automating the Kickstart Installation

You can start a Kickstart installation manually with some user interaction at the system prompts. However, that's no fun, since I want you to see how to automate the whole darn thing! During the installation, you must specify a special boot option (inst.ks=) when booting the system.

Following is a typical Kickstart file.

```
#version=RHEL7
# System authorization information
auth --enableshadow --passalgo=sha512

# Use network installation
url --url="ftp://192.168.1.25/pub/"
# Run the Setup Agent on first boot
firstboot --enable
ignoredisk --only-use=sda
# Keyboard layouts
keyboard --vckeymap=us --xlayouts='us'
# System language
lang en_US.UTF-8

# Network information
network  --bootproto=dhcp --device=eno16777736 --ipv6=auto --activate
network  --hostname=localhost.localdomain
# Root password
rootpw --iscrypted $6$RMPTNRo5P7zulbAR$ueRnuz70DX2Z8Pb2oCgfXv4qXOjkdZlaMnC.CoLheFrUF4BEjRIX8rF.
```

Configuring the Clients to Automatically Install the Linux Software through Kickstart

You must instruct the clients to boot from the network from the BIOS, by selecting the Kickstart option from the PXE menu. Once the kernel and ram disk load, the client detects the Kickstart file and automatically installs the Linux software without any user intervention. If you want, you can connect to the installation process with a VNC client from another host in order to monitor the installation.

Automatically Spinning up Virtual Environments with Vagrant

Vagrant is a powerful open source tool that simplifies the task of spinning up virtual machines (VMs) and running them, with the help of simple command-line utilities. Vagrant makes it easy to distribute and share a virtual environment and it supports all major virtual platforms such as VirtualBox, VMWare and Hyper-V. It also supports all the well-known software configuration tools such as Chef, Puppet, Ansible and Salt.

Development environments can be notoriously hard to configure. Vagrant helps you, the Linux sysadmin, to take over the responsibility for setting up the development environment from the developers. New developers can be easily on boarded as a result, and it also makes it easy to update the environment used by the developers.

Vagrant is a configuration tool for VMs and helps developers quickly spin up new VMs easily. However, you can use it for other purposes. Generally, Vagrant is used to create VMs that help develop and deploy software. When you use Vagrant to set up development environments, it's a piece of cake to mimic a production environment, and also employ the same provisioning tools and strategies that you use in your production environment.

Developing consistent environments is the key to effective deployment of software. If your deployment pipeline is already in place and is working well for you, you can use Vagrant to recreate the same processes in the development environment. If your development pipelines need to be improved, Vagrant is ideal as an environment within which to develop the processes that make development environments consistent.

Vagrant is a big help in managing the configuration of VMs that you run out of Oracle Virtual Box, and is ideal for testing various things. Vagrant wasn't designed for heavy duty configuration across the data center. You can use a serious enterprise configuration system such as Ansible along with Vagrant, with Vagrant helping you test the configuration code that you're deploying with Ansible.

A few years ago, web applications were mostly all PHP and MySQL. Today, you have several web application frameworks such as Ruby on Rails, various databases, web servers, application servers and backend services. Installing all these components and configuring them correctly is a nightmare at times when you do it locally by hand.

Vagrant works with Oracle Virtual Box, and commercial alternatives such as VMWare, and also remote environments such as Amazon's Elastic Compute Cloud (EC2).

Beyond the nightmare of installing locally, problems like misconfiguration, differences between dev and prod environments, the difficulties of managing multiple projects and syncing development environments among all team members, are all problems that Vagrant elegantly solves. Working with multiple projects? Just create a separate VM for each machine. New team member to be on boarded quickly? Hand him/her a laptop and ask them to run just one command – vagrant up, to be up and running within minutes.

If you're an operations engineer, you can work on system automation scripts and can test them on a full-fledged sandbox that mimics production, so you can test real-

world scenarios. If something is gets messed up, simply destroy the VM and recreate a fully functioning environment at the snap of your fingers.

Although Vagrant is mostly used in a web application environment, you can use it for anything you want, as long you want to work with VMs. Vagrant is simply amazing – no two ways about it! Regardless of the complexity of your virtual environment and regardless of the type of virtualization you're using, you can easily bring up the complete virtual environment with the following simple vagrant command.

```
$ vagrant up
```

With just this one command, you get all the following done:

- Create a virtual machine based on any Linux distribution you like
- Change the properties of this VM such as RAM, storage, etc
- Set up the network interfaces for this VM so you can access it from a different server, local or remote
- Set up the shared folders
- Set the hostname for the server
- Boot up the VM – you can see in VirtualBox that there's a new VM in the running STATE NOW
- Provision software on the new VM via shell scripts, or CM tools such as Chef and Puppet

All this in a New York minute – wow!

The Vagrant commands let you ssh into the new VM, start, stop and resume it, and when you want to get rid of it, "destroy" the machine by deleting it from your hard drive – all virtual hard drives (file folders) relating to the VM are removed. Any data that you don't save in a shared folder is lost for good, so be careful when you "destroy" a Vagrant VM. You can also package the machine state and distribute it to other developers.

he creator of Vagrant, Mitchell Hashimoto, has described Vagrant as the "Swiss army knife for development environments" since it does everything you need to create and manage development environments and also automates things and helps set up dev environments that parallel production environments.

Vagrant helps you quickly spin up full-fledged but disposable working environments.

Some Background

In Chapter 4 you learned about the different types of virtualization. As you know by now, a virtual machine runs within a software process (the hypervisor) which runs on a host computer, and mimics a physical computer. It's the hypervisor that provides

the computing infrastructure such as CPUs and RAM to the virtual machines. You also learned about the two types of hypervisors – Type 1 and Type 2.

A Type 1 hypervisor runs on the bare metal host machine hardware and controls access to the computing resources, and their allocation to the VMs. VMWare ESX/ESXi and Oracle VM Server and examples of Type 1 hypervisors. Type 2 hypervisors on the other hand, sit on top of the OS and use the OD to control the computational resources. Vagrant environments typically use Type 2 hypervisors as the hosts for VMs that you create with it. The two most popular Type 2 hypervisors are Oracle VirtualBox and VMWare Workstation/Fusion family.

I use the freely available Oracle Virtual Box to show how to use Vagrant to spin up guest machines that run within the hypervisor. You can create guest machines that run an entirely different OS from that being run by the host computer. Since Vagrant employs the same API to run VMs on different hypervisors, sharing virtual environments is very easy between teams that are working with different OS platforms.

If you've ever set up a virtual development environment, you know you'd have to do most or all of the following to set up a new VM:

- Download the VM image
- Boot up the VM
- Configure shared folders, network connections, storage and memory
- Use a configuration tool such as Puppet/Chef, Salt or Ansible to install any required software.

The simple vagrant up command does all these tasks for you! The command sets up the entire development environment and a developer can also, with equal ease, destroy and recreate the virtual environment, all within a few short minutes as well.

Vagrant and its Alternatives

You can use alternatives such as plan desktop virtualization with VMWare and VirtualBox to do some of the things you can do with Vagrant – however these virtualization solutions don't have the unique workflow of Vagrant. While you can do everything you can do with Vagrant with a regular virtualization solution, it isn't an automated process as with Vagrant.

Containers are somewhat of an alternative but they really don't provide full virtualization – they are instead isolated environments running the same kernel. As you saw in Chapter 5, containers do offer numerous benefits, but a big downside is that they can run only the same OS that runs on the host. If you use containers, all members of a team need to use the same host OS.

Containers are great for production settings since they securely isolate resources without a performance overhead. In a development environment the limitations posed by containers outweigh their benefits.

The cloud is another alternative but you normally incur a higher financial cost going that route, compared to using a local Vagrant environments to manage you development needs.

Getting Started with Vagrant

In order to run Vagrant on a server (or your laptop), you need to first install Oracle VirtualBox, which is open source, and thus free. You can download Virtual Box from *http://virtualbox.org*.

A virtualization system such as VirtualBox is called a provider and although I use VirtualBox since it's free and easy to get going, there are alternative providers such as a VMWare provider.

Virtual Box has minimal system requirements, but since a single workstation will be running multiple VMs at any given time, you need to ensure that the laptop or server where you're playing with Vagrant has sufficient RAM. There's no hard and fast rule here as to how many VMs you can run on a server – it depends on the software running in each of the VMs hosted by the server.

You can run VirtualBox in a Windows or a Linux environment.

Once you've VirtualBox installed, you are ready to install Vagrant, which you can download from *http://vagrantup.com*.

Vagrant doesn't create and host VMs – it's the hypervisor (VirtualBox in my case) that does those things. Vagrant simply manages the VMs.

In a production environment, you can use hypervisors other than Oracle VirtualBox – you can use VMWare Desktop Applications (Fusion and Workstation) if you need to deal with a large number of VMs. If you want to simplify things, you can go with an external hypervisor such as Amazon EC2 or DigitalOcean.

Although Vagrant supports other virtualization providers, Oracle VirtualBox is the most popular provider for developers.

On a RedHat system such as Fedora, install Vagrant binaries as shown here:

```
$ yum install 'vagrant*'
```

You can check the installation by typing the following at the command line:

```
$ vagrant -verison
Vagrnt version 1.1.4
```

You're off to the races at this point.

Spinning up a New VM

You spin up new VMs with the vagrant up command, but first you must initialize the image. The vagrant init command creates a Vagrant configration file from a template. In the following example I use an Ubuntu base image, also referred to as a Vagrant box.

The vagrant init command initializes a new Vagrant environment by creating a Vagrantfile.

```
$ vagrant init hashicorp/precise64
A `Vagrantfile` has been placed in this directory. You are now
ready to `vagrant up` your first virtual environment! Please read
the comments in the Vagrantfile as well as documentation on
`vagrantup.com` for more information on using Vagrant.
$
```

Start the new virtual machine with the vagrant up command:

```
$ vagrant up

Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'hashicorp/precise64'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'hashicorp/precise64' is up to date...
==> default: Setting the name of the VM: SG0221771_default_1461505264672_17333
==> default: Forwarding ports...
    default: 22 (guest) => 2222 (host) (adapter 1)
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    ==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
    default: Guest Additions Version: 4.2.0
    default: VirtualBox Version: 5.0
==> default: Mounting shared folders...
$
```

Here's what happens when you run the vagrant up command:

- Vagrant creates a new VirtualBox machine based in the base image within the box specified in the Vagrantfile. Vagrant does this by copying the virtual hard disk files.
- VirtualBox randomly generates a MAC address when it creates a new machine. Vagrant matches the MAC address for NAT networking.

- Vagrant sets the name of the virtual machine
- Vagrant forwards port definitions (more on this later in this chapter)
- Vagrant boots the new VM
- Vagrant mounts the shared folders that you can use to share data between the VM and your laptop or server As you know, I'm using the Oracle VirtualBox in this example, but the procedure for spinning up new VMs is exactly the same when you bring up VMs using VMWare or AWS as a provider.

The vagrant up command starts and provisions the Vagrant environment. You now have a full featured 64-bit 12.04 LTS virtual machine running in the background. The Vagrant machine instance is up and running, although you won't see it, since it's headless (that is, it has no GUI). You can connect to this machine by using the following command:

```
$ vagrant ssh
Welcome to Ubuntu 12.04 LTS (GNU/Linux 3.2.0-23-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
New release '14.04.4 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Welcome to your Vagrant-built virtual machine.
Last login: Fri Sep 14 06:23:18 2012 from 10.0.2.2
vagrant@precise64:~$
```

When you log into the new VM, you do so as the default user vagrant:

```
$ vagrant@precise64:~$ whoami
vagrant
$ vagrant@precise64:~$
```

You really don't need the root password but if you want to, you can do so by issuing the command sudo passwd root as the user vagrant. You can then set the root password to anything you like and login as the user.

The vagrant ssh command connects to the machine via ssh. It drops you unto the SSH console within the new VM. You can do everything on this machine as you'd in a normal server, such as installing software, creating Docker containers, etc. You log out of the VM by typing exit:

```
vagrant@precise64:~$ exit
logout
Connection to 127.0.0.1 closed.
$
```

The exit command brings you out of the VM and puts you back into your terminal on the host server. Run the vagrant destroy command to delete your new VM – you can create it again, no problems, with the vagrant up command if you need it again.

```
$ vagrant destroy
    default: Are you sure you want to destroy the 'default' VM? [y/N] y
==> default: Forcing shutdown of VM...
==> default: Destroying VM and associated drives...

$ exit
```

The Vagrantfile

When working with Vagrant environments, you configure Vagrant per project, with each project having its own specific work environment. Each of the vagrant projects has a single Vagrantfile that contains its configuration. The Vagrantfile is s text file that Vagrant reads to find out the configuration of your working environments, such as:

- The operating system,
- The CPUs and RAM per VM
- How the VM can be accessed
- Provisioning of various software on the VM

It's a good idea to put the Vagrantfile under version control so all members of a team get the same software, settings and configuration environment for their work environment.

When I ran the vagrant init command in the previous section, it created an initial Vagrantfile. The file is in the directory where I ran the vagrant init command. Read the Vagrantfile since it gives a very good idea of how to configure the essentials of a vagrant project. In my case, everything is commented out except the following:

```
Vagrant.configure(2) do |config|
  config.vm.box = "hashicorp/precise64"
end
```

In my case the file essentially contains a block of Vagrant configuration that contains a configuration value for config.vm.box – this is all it took to create the new VM! Since Vagrantfile are portable, you can take this file to any platform that Vagrant supports, such as Linux, Windows and Mac OS X,

Vagrant Box

As I explained in the previous section, the simple textile named Vagrantfile contains the configuration for spinning up your new VM. But you also need something from which to create this VM from – and that something is called a Vagrant box, A box is simply a base image for an OS that Vagrant will clone to quickly create a functioning VM. You can look at the box file as a template for a Vagrant-created and managed virtual machine. Using the box is the key strategy that makes it possible to spin up

VMs in seconds – imagine all the work involved in creating even the simplest of VMs from scratch.

Vagrant uses base boxes to clone new VMs

In the Vagrantfile, the value config.vm.box specifies the name of the box that Vagrant will use to create your new VM. In our case, it was:

```
config..vm.box = “precise64”
```

Multiple environments can share the same underlying box, with each environment using that box as the template for creating new VMs. However, you use a separate Vagrantfile for each project.

You can view all the Vagrant boxes you have on a server with the following command:

```
$ vagrant box list
centos/7           (virtualbox, 1602.02)
hashicorp/precise64 (virtualbox, 1.1.0)
rails-v1.0.0       (virtualbox, 0)

~
$
```

Vagrant VMs use the shared folders feature, which means that your development teams can continue to use the development tools (IDEs etc) they’re most familiar, and hence the most productive with.

Vagrant Networking Vagrant automatically configures networking with the VMs you create by letting teams communicate with the VM. There are several networking options as I explain shortly, but here’s an example that illustrates the basics of how Vagrant handles networking.

In the example. I use a forwarded port. A forwarded port exposes a port on the VM as a port on the host server. As you know, port 80 is the default port for web services, so let me expose port 80 so I can access any web services. In my Vagrant file, then, I add the following line:

```
config.vm.network "forwarded_port", guest: 80, host: 8080
```

I then issue the command vagrant reload to restart the VM with the new network port settings:

```
$ vagrant reload
==> default: Attempting graceful shutdown of VM...
```

```
==> default: Clearing any previously set forwarded ports...
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
==> default: Forwarding ports...
    default: 80 (guest) => 8080 (host) (adapter 1)
    default: 22 (guest) => 2222 (host) (adapter 1)
==> default: Booting VM...
…
```

You can see how Vagrant has forwarded port 80. You can test the forwarded port by starting a simple web server from within the VM and connecting to it from a browser on the host server:

```
$ vagrant ssh
vagrant@precise64:-$ cd /vagrant
vagrant@precise64:/vagrant$ sudo python -m SimpleHTTPServer 80
Servicing HTTP on 0.0.0.9 port 80…
```

I started a basic web server on port 80 with the command shown here. If you now open a browser and point it to localhost:8080 on your laptop (or wherever the VM is running) you'll see the directory listing for the /vagrant directory, served from your new VM.

Vagrant offers three different ways to set up networking:

- Forwarded ports: I've shown an example of this method, which is quite simple to setup you need to enumerate each and every port number, which becomes onerous in a complex environment
- Host-only networking: Creates a network that's private to your host and the VM s running on the host. While this is a secure way to network, it limits access from other machines outside of the host server.
- Bridged networking: Bridges the VM onto a device on the physical host, making the VM appear as another physical machine on the network. It offers you the benefit of having an OIP to access the VM, but you can't specify a static IP for a bridged network as the IP addresses are serviced via DHCP/

Provisioning Vagrant Boxes

In earlier sections, you learned how to spin up a VM with Vagrant's help, but it was a bare base box. You can use Vagrant boxes that already have software installed, such as Hadoop and Ruby on Rails, for example. You can also automatically install software as part of the creation of your development environments – this process is called provisioning.

While you can install any software you need after booting up a new VM. It's inefficient to do when you've a bunch of machines to configure. Vagrant lets you automate provisioning with shell scripts, or a CM tool such as Chef or Puppet.

How Automated Provisioning Helps You

Automated provisioning removes one of the biggest issues in setting up consistent development environments – configuration drift. Automated provisioning of software is easy, repeatable and also helps keep development and production systems in sync, avoiding unpleasant surprises when you move apps form development to production settings.

As a system administrator or an operations engineer, you can use Vagrant for quickly testing infrastructure changes before moving the changes to production

You can use Vagrant as a good training ground for learning CM tools such as Puppet and Chef, but if you're a novice, with CM tools, just start with simple shell scripts to see how to provision software with Vagrant. Once you become adept at provisioning, it's a simple jump to a full-fledged CM tool.

You can perform Vagrant provisioning with Chef through Chef Solo or the Chef client. Chef Solo is great for testing and for small environments. If you already have a Chef Server, Chef Client is going to be what you want to use. Similarly, you can do provisioning both with and without a master when using Puppet, and the master less approach is simpler and better when starting out with Puppet.

Besides Puppet/Chef as build-in provisioners you can extend vagrant to use additional provisioners through easily available plug-ins.

An Automated Provisioning Example

Let me show you how to configure a provisioner to set up an Apache web server. Before I start automating the installation of Apache, I must first set it up manually on a Vagrant box, to capture the correct procedure to create the web server.

In this example, I use the precise64 Ubuntu 64 bit Linux distribution, and I want to export port 80, as explained in an example earlier on. My Vagrantfile for this box will look as follows:

```
Vagrant::Config.run.do [config]
   config.vm.box = "precise64"
    config.vm.forward_port 80, 8000
end
```

Use SSH to log into the VM, so you can install and configure Apache on the Ubuntu Linux server.

```
$ vagrant ssh
```

Install Apache on the Ubuntu server

```
$ sudo apt-get update
$ sudo apt-get instll apache2
```

Vagrant runs all commands in a provisioning shell script as the user root, so there's no need to use sudo. If for any reason you think you need to be root in a Vagrant environment, do the following to set the root password, and then login as root:

$ sudo passwd root

By default Apache servers web content from /var/www directory. To avoid having to modify the Apache configuration files I 'm going to modify /var/www directory to be a symbolic link to /vagrant which is the default shared folder directory.

```
vagrant@precise64:/vagrant$ sudo rm -rf /var/www
vagrant@precise64:/vagrant$ sudo ln -fs /vagrant /var/www
```

Now, Apache will by default serve any files you placed in the shared folder. At this point, if you go to *http://localhost:8080*, you'll see the directory listing of the shared folder:

Next, create an index.html (default file that Apache serves) file in the shared directory (/vagrant), as shown here:

```
vagrant@precise64:/vagrant$ logout
Connection to 127.0.0.1 closed.
$ echo "<strong>Hello<strong>" > index.html
```

If you refresh your browser now, you'll see the following:

```
Hello
```

Each time you bring up the VM with the vagrant up command, you'll need to perform all these steps, which is a waste of time. With automated provisioning, you can work much smarter.

Automated Provisioning

You saw how to set up and configure Apache in the previous section. Let's see how you can automate the same through a shell script, a Chef Recipe and a Puppet manifest.

Automating Provisioning with Shell Scripts

Using a shell script simply means that you put all the commands you had to issue manually into a single file. Here, I call the file provison.sh, but you can name it any-

thing you want. Add the following lines to the provision.sh file and save it in the Project directory.

You've your provisioning shell script ready – all you need to do now is to let Vagrant know where to find it. In order to do this, stick the following line in the Vagrantfile you created earlier:

```
config.vm.provision "shell", path: "provision.sh"
```

I didn't have to specify a path to the provision.sh file since relative paths are relative to the Project root directory.

In order to test whether you've correctly automated the setup and configuration of Apache as part of a new VM creation, firsts destroy the current VM and run the command vagrant up, to start with a clean state:

```
$ vagrant destroy
$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'hashicorp/precise64'...
…
==> default: Running provisioner: shell...
    default: Running: C:/cygwin64/tmp/vagrant-shell20160424-6872-1xaawtl.sh
Installing Apache and setting it up…
$
```

As you can see, I've created a basic web development environment with no human interaction at all. You can use more complex scripts to create and configure databases, cron jobs, and whatever else you need to set up a full-fledged development environment.

Automated Provisioning with Chef

You'll learn a lot about Chef (and Puppet) in Chapter 7. For now, just let me say that you can use either chef-solo or chef-client to provision Vagrant environments. Chef-solo is easier, so I use that to show how to provision software in Vagrant VMs.

In order to provision our Apache web server using Chef, you need to create a cookbook and a Chef Recipe. By default Vagrant will look in the cookbooks directory relative to the Vagrant project directory for its cookbooks. So, first create the cookbooks directory under the Project directory and then create the following Chef recipe:

```
execute "apt-get update"
package "apache2"
execute "rm -rf /var/www"
link "/var/www" do
   To "/vagrant"
end
```

Save the recipe file to cookbooks/mydir/recipes/default.rb.

Once you do this, stick the following line in your Vagrantfile:

```
config.vm.provision “chef-solo”, run-list: [“mydir”]
```

Now that you’re all set up, run the vagrant destroy command to remove all traces of the previous installation and then execute the vagrant up command to create a new VM with automatic provisioned of the Apache server by Chef.

Automated Provisioning with Puppet

Automated provisioning with Puppet is quite similar to provisioning with Chef. I use a simple Puppet set up without a master. Just as Chef uses cookbooks and recipes, Puppet uses manifests. In order to let Puppet handle the provisioning of our Apache server, you must create a manifest folder under the project root directory. Next, create the Puppet manifest that’ll do the job for us:

```
exec  { “apt-get update”:
   Command => “/usr/bin/apt-get update”,
}

package { “apache2”:
   Require => Exec[“apt-get update”],
}

file { “/var/www”:
   Ensure => link,
   Target => “/vagrant”,
    Force => true,
}
```

Remove the VM with the vagrant destroy command and run vagrant up again.

Finally, you don’t necessarily have to choose among the available provisioners, in the sense that you can use more than one provisioner by simply specifying multiple config.vm .provisioner directives in your Vagrantfile, as shown here:

```
Vagrant::Config.run.do [config]
   Config.vm.box = “precise64”
…
   Config.vm.provision “shell”, inline, “apt-get update”
Config.vm.provision “puppet”
…
End
```

Vagrant will provision your new M by installing and configuring the software provided by each provisioner, in the order you specify the provisioners in the Vagrantfile.

Creating Vagrant Base Boxes

A Vagrant box that contains just the minimum software that allows Vagrant to function, but nothing more, is called a base box. A base box, such as the Ubuntu box pro-

vided by the Vagrant project *"precise64") that I used earlier on, isn't repackaged from any other Vagrant environment, and that's why it's called the "base box".

The Vagrant project and others offer numerous base boxes. If you're starting out, get familiar with Vagrant by using the available base boxes. Later on, once you get really comfortable, you can create your own base boxes to serve as a starting point for fresh development environments.

Components of a Base Box

Vagrant base boxes include a bare set of binaries. Here's a list of the minimum components in a base box:

- Vagrant package manager
- SSH
- An SSH user to allow Vagrant to connect

Note that the provider you're going to use also determines what goes into a base box. For example, if you're using VirtualBox as the provider, you'll need the highly useful Guest Additions so that you can use the shared folders capability of the VM.

It's quite easy to build a custom box from a preexisting Vagrant box such as ubuntu/trust64. Just boot the predefined box and modify its configuration per your requirements. Export the box to a new file (with the .box extension) by executing the vagrant package command.

Packaging a base box into a box file differs, based on the provider. In the following example, I'm using Virtual Box, and I need do the following to package the new base box:

```
$ vagrant package --base Ubuntu-14.04-64-Desktop
```

The package command creates a file named package.box.

There are no rules as to how you should distribute the base boxes you create. However, Vagrant recommends that you add the new box to HashiCorp's Atlas to supporting versioning, push updates and lots of other reasons.

You test the new base box by spinning up the new VM:

```
$ vagrant up
```

If the VM is spun up correctly, your new base box is good!

Using Packer and Atlas for creating base boxes

In the previous section, I showed how to create a Vagrant base box from scratch using a manual method. Vagrant strongly recommends using HashiCorp's Packer (and chef/bento or boxcutter templates) to create reproducible builds for a Vagrant base box. It also recommends that you use HashiCorp's Atlas to automate the builds.

Packer is a command line tool that lets you automate the creation of VMS by generating Vagrant boxes. Packer is the recommended way to create Vagrant boxes. In a nutshell, here's how Packer works:

- It downloads the ISO CD or DVD image of the OS you want to use
- It executes the installation program and sets up the server with the default configuration
- It runs provisioners such as shell, Chef, Ansible and Puppet to customize the new system
- It exports the custom system and packages it into a .box file.

Packer not only lets you create VMs of different operating systems, such as Ubuntu and CentOS and Windows 7, it also lets you create boxes for various providers such as VirtualBox and VMware.

Using Packer, you can specify the configuration pf your Vagrant boxes (memory, disk size etc) in a JSON file. Once you specify the needed automation parameters (such as Ubuntu preseed file), Packer will do the initial OS deployment for you.

You can find details about Packer at *http://www.packer.io*. You can find examples of Packer definition files by going to the repository (called bento) maintained by Chef at Chef's Github account. *https://github.com/chef/bento*

You can download Packer distributions from *https://www.packer.io/downloads.html*. The chef/bento project contains numerous Packer definitions. You can clone the the chef/bento project from GitHub (*https://github.com/chef/bento*). Once you clone the project, in the bento/packer subdirectory, you'll find a number of JSON files that contains definitions for building various operating systems. You can use Packer and these JSON files to build base boxes for Ubuntu, Fedora, etc. For example, you can build an Ubuntu 14.04 box for the VirtualBox provider by running the following packer command:

```
$ packer build -only=virtualbox-iso ubuntu-14.04-i386.json
```

Note that this packer command creates a new Ubuntu box without using a preexisting Vagrant box. It does this by downloading an ISO image from the Ubuntu distribution site. When the packer command finishes executing, you'll see the following file:

```
bento/builds/virtualbox/opscode_ubuntu-14.04-i386_chef-provisionerless.box
```

What you have here is a bare bones system with just enough software for the system to function. You can customize this box to your heart's content.

You can test this new Vagrant box created by Packer by using the same procedures as in the previous section where I showed the steps for manually creating a base box. Here's how to do it for our example:

```
$ vagrant box add bento-ubuntu opscode_ubuntu-14.04-i386_chef-provisionerless.box    # install the
$ vagrant init -m bento-ubuntu         # initialize the new VM
$ vagrant up                            # boot the new VM
$ vagrant destroy       # destroy the box
```

Parallel Job-Execution and Server Orchestration Systems

In a large environment especially, it's critical that you use techniques other than old-fashioned shell scripts to simultaneously execute tasks across a number of servers. Remote command execution tools help significantly in performing near real-time parallel execution of commands.

Automated server provisioning tools such as Razor and Cobbler let you automate the installation of servers and the management of those servers with configuration tools. Using one of these tools, it's easy to go from a bare metal or VM server with nothing on it and come up with a fully configured system in a very short time.

Working with Remote Command Execution Tools

When dealing with a set of servers, administrators typically write programs that can parallelly execute their commands. It's easier to manage these types of operations by using tools and frameworks explicitly designed for this type of work. In this section, I explain some of the more popular tools for server orchestration and parallel job execution. Specifically, the following are the tools I discuss:

- Parallel Execution Processors (PDSH, DSH, CSSH)
- Fabric
- Mcollective

Parallel (SSH) Execution Processors

If you're handling just a handful of servers and don't think that you need the overhead of a configuration management (CM) tool such as Puppet/Chef/Ansible/Salt, you can look at other simpler tools that help you manage multiple Linux servers. Most of these tools also work very well with Puppet and other CM tools, to help make your life easier.

The tools I've in mind are all based on SSH, so you must first setup SSH key based logins on the servers (please see Chapter 2). Let's quickly review these remote execution tools in the following sections.

PSSH (Parallel SSH)

Pssh is a program for executing ssh in parallel in a set of nodes. It can send input to all processes, passwords to ssh and save output to files. Following is an example that shows how to copy a file to my home folder on two servers.

```
pscp.pss -v -H "alapati@host1 alapati@host3" test.gz /home/alapati
```

DSH (DISTRIBUTED SHELL)

You can install DSH (distributed shell) or pdsh (parallel distributed shell) with apt-get or yum. In order to execute a command on multiple nodes, you add the node list to a file such as the following example:

```
vi /tmp/test.list
host01
host02
host03
```

Once you have your server list ready in a file, pass the node list file to pdsh when executing a command:

```
# pdsh -aM -c uptime
```

This command will show you the uptime on each server in your node list file (test.list in this example).

CSSH

CSSH (ClusterSSH) is somewhat different from the pssh and pdsh (or dsh) tools – it opens an xterm terminal to all the hosts you specify, pus an admin console. When you type a command in the admin console, it's automatically replicated to all windows. You can see the command actually execute on the various windows for separate servers. Here's an example showing how to use the CSSH tool.

Create a cluster configuration file named clusters in /tmp and define your servers in that file.

```
vi  /tmp/clusters
clusters = 1204servers
1204servers = host1  host2
```

Run the following command.

```
# cssh alapati@host1 alapati@host2
```

You'll now see two terminal windows and the as you type your commands in the admin console, you'll see the command executing in both terminals.

Fabric

Fabric is a command-line tool that uses SSH to help you orchestrate various configuration operations. Fabric automates communications with remote servers by working on clusters of machines. It can help start and stop services on the clusters. You can use it for both application development and systems administration. Fabric is handy for uploading and downloading files. It can also prompt users for input and cancel execution of the tasks.

MCollective

MCollective (Marionette Collective) is a sophisticated framework that helps you build server orchestration and parallel job-execution systems. The purpose of Mcollective and similar orchestration frameworks is to allow you to parallelly execute configuration changes across multiple systems in a controlled fashion. MCollective is a tool designed to orchestrate and synchronize changes across a large number of servers simultaneously. The name Marionette alludes to a classic marionette controlling puppets.

Later in this book, you'll learn a lot about configuration management (CM) tools such as Puppet, Chef and Salt. How does a server orchestration tool such as Mcollective relate to those tools? Here's the difference: CM tools are for what they say they do – achieving consistent configuration in your data center. A tool such as MCollective has a far narrower scope – it lets you orchestrate changes in a parallel fashion.

How Mcollective differs from traditional tools

System administrators are used to rigging up scripts to perform simultaneous updates of a cluster of servers. However the scripting approach suffers from the following drawbacks:

- They work through a list of servers, one at a time
- They can't handle unexpected outcomes or responses
- They can't handle fatal error messages output to the screen
- They don't integrate well with or complement other management tools

MCollective contains the following features that make it quite different from the other parallel execution tools:

- You can use custom authentication and authorization mechanisms
- You can parallelly execute changes on thousands of servers without a master
- Make it possible to diagnose result codes because full data sets are returned as result codes
- Actions can be taken on the responses by processors
- In addition, Mcollective integrates very well with CM tools such as Puppet and Chef.

Often, an administrator runs something on a bunch of servers at the same time by running code that looks like the following:

```
$ for host in bunch of hosts
  do
     scp config-file $host:/some/path
     ssh $host "service apache restart"
  done
```

When you run this code for a large number of hosts, several issues might crop up – for example, you can't easily keep up with the output of the commands, and errors in the middle of the sequence can be missed. Your goal is to make sure that you do this thing fast and know for sure that the commands worked on all the target servers.

While Puppet and Chef and similar CM tools can help you make changes in systems, they really aren't tools designed to perform massive simultaneous deployments. The tools ensure eventual compliance of systems over a period of time. Something like puppetmaster can process only a few systems at once. Mcollective is a good complementary tool for CM tools such as Puppet (Mcollective ships as part of Puppet 4), and helps you achieve true parallel execution with consistent results.

The key difference between Mcollective and a tool such as Puppet is that Mcollective is a tool with a narrow focus – it lets you perform small changes across a huge number of nodes at precisely the same time. Puppet can perform numerous changes to ensure consistent configuration among a bunch of nodes, but it takes its time to get this done.

How Mcollective Works

Mcollective avoids two key drawbacks of other parallel execution tools:

- First, there's no central master, which helps you eliminate potential bottlenecks with a master/server centralized architecture for single server managing deployments over a bunch of servers.

- Secondly, it avoids drift among the servers it's updating, since it doesn't process the clients in an ordered loop as a shell script might, for example.

MCollective uses a unique concept of what the terms server and clients mean. A node you want to control through Mcollective is deemed a Mcollective server and it runs mcollectived. A node with the mco command-line client installed is deemed a MCollective client which is capable of issuing commands. You install the client software just on those systems from where you want to send requests. You can use management hosts or a bastion host (or your own laptop) for this.

You can use the mco command-line client in scripts or interactively. You can also write your own custom clients in Ruby to use as back ends for GUI applications.

The key to the scalable and fast parallel execution capability of MCollective is the Publish-Subscribe infrastructure it uses to carry requests to the servers. The mcollectived process on each server registers with the middleware broker (ActiveMQ or RabbitMQ for example) and immediately grabs and evaluates requests sent to the middleware broker by the clients. The mcollctived process uses an agent with which it immediately executes the command to satisfy the client requests. It's the agents that you install which process the requests and perform actions. The Publish-Subscribe mode permits simultaneous execution on hundreds or thousands of servers at the exact same time.

Puppet Labs (it maintains both Puppet and Mcollective) recommends ActiveMQ as the best middleware for high performance and scalability

You can control which systems execute specific commands using filters on hostnames, operating systems, installed packages, and similar criteria. The Mcollective agents report back with the status of the process initiated by them. MCollective agents are available for Puppet, Chef, CFEngine, Ansible and Salt.

Mcollective and Puppet

You can use MCollective to control Puppet agents on various nodes. You can use MCollective to:

- Start and stop the Puppet agents
- Run the Puppet agent with various command-line options
- Make changes to nodes using Puppet resources

- Select the node to modify based on the Puppet classes or facts (key-value pairs with information about the nodes – Puppet's facter program is the most common way to get the facts) on the node

Installing and Configuring Mcollective

You can manually install the various binaries that you need to make MCollective work (including the middleware broker), but the recommended way to install and configure MCollective is through a CM tool such as Chef, Puppet, Salt, CFengine or Ansible. The reason for this is that it's tedious to install MCollective manually on numerous servers, and using a CM tool makes it easy to maintain MCollective over time, as well as customize its settings on some servers.

Now that you've seen how parallel command job execution frameworks can help you, let's take a look at two highly useful automatic server provision tools – Razor and Cobbler.

Server Provisioning with Razor

Razor is a popular automatic provisioning tool that lets you install servers across your data center. Razor also integrates very well with CM tools, so you can easily go from a bare metal server or VM with nothing on it to a fully configured server with the help of Razor and a CM tool.

You can install Razor by itself, and Puppet Enterprise also bundles it with its installation media. Razor's broker component also supports Chef, and you can write plugins for other CM tools such as Ansible and Salt as well.

How Razor Works

Razor uses TFTP, DHCP and DNS to support automated server deployment. Razor, written in Ruby, runs as a TorqueBox web application and uses PostgreSQL as its backend, and it has its own network. When you connect a new server that's configured for network boot to the Razor network, Razor does the following:

- The new server does a network boot and connects to Razor and loads its microkernel
- It detects the new node and learns about its characteristics by booting up with Razor's microkernel and collecting information about it
- The new node gets the microkernel and registers with Razor, and will be bootstrapped by Razor by loading the installation files
- If the new server's "facts" match a tag, the tag is compared to policies stored in the Razor server and the first matching policy is applied to the new server

- The new server is configured with an operating system, based on what the policy specifies
- If the policy contains the relevant broker information, Razor performs a handoff to the Chef or Puppet CM tool

Razor's Architecture

You can look at Razor as consisting of the following components:

- The Razor server
- The Razor microkernel
- The Razor Client

The Razor Server

The Razor server is Razor's main component. TorqueBox, which is application platform based on the JBoss application server, hosts the Razor server. You can interact with the Razor Server through the Razor client or through RESTful APIs.

The Razor Microkernel

The microkernel is just what it sounds like – it's a tiny Linux image that is used to boot on the new nodes and inventory the nodes. During the discovery stage, if the microkernel can't find matching policies for the new server, the server will present the microkernel's login screen, but normally you don't long into the microkernel.

The Razor Client

The client lets you access the Razor server and it's probably a good idea to run it on the same machine where you run the server, since the client provides no authentication by itself.

Working with Razor

If you're using Puppet Enterprise (PE), you already have Razor – however you must still configure the required database (PostGres), DHCP server, DNS server and an image repository.

To install Razor when you've Puppet running, you need to do very little:

```
puppet module install puppetlabs/razor
puppet apply -e 'include razor'
```

There are a number of prebuilt Vagrant environments that you can download and use. Or, you can manually install Razor yourself.

In addition to the PostgreSQL database as a backed server, Razor needs both DHCP and DNS servers. Razor assigns IP addresses to the new nodes that it installs through DHCP.

If you're dealing with a small test environment, you can use dnsmasq for both DHCP and DNS.

Server Provisioning with Cobbler

Cobbler is a Linux installation server that helps you simplify server provisioning by centralizing and automating tasks involved in the configuration and administration of an installation server. Cobbler helps you quickly setup a network installation environment and also serves as a tool that helps you automate tasks in Linux environments. Originally it was bundled with Fedora, but since 2011, it's being also bundled with Ubuntu. Cobbler glues together several related Linux tasks so you don't have to worry about each of them when performing a new installation or modifying an existing installation.

Cobbler has a built-in configuration system and integrates well with other systems such as Pallet, for example. You mostly use the commands cobbler check and cobbler import to perform the initial setup from the command line but can use a web application later on.

Cobbler s ideal for network installs which you can configure for PXE, media based network installations and virtualized installations with Xen, KVM, configuration management orchestration, etc.

Cobbler is also helpful in managing DHCP, DNS and the yum package mirroring infrastructure. It has its own lightweight CM system and you can integrate it with Puppet and other systems as well.

Where Cobbler Can Help You

As I showed earlier in this chapter, when performing a network environment, for server installations you must perform all the following tasks:

- Configure DHCP, TFTP, DNS, HTTP, FRP, NFS and other services
- Customize the DHCP and TFTO configuration files
- Create the automatic deployment files such as the Kickstart file
- Extract the installation binaries to the HTTP/FTP/NFS repositories

This sequence of steps involves a lot of manual work: you must manually register each client machine, and any change in the provisioning of a server means a manual change in the configurations and the automatic deployment files. The TFTP directory and other files can get quite complex when dealing a large number of machines.

Cobbler was designed to tackle the system related issues head on by acting as the central management point for all machine provisioning tasks. It lets you install machines without manual intervention. You simply run a command such as add new repository or change client machine operating system, and Cobbler will:

- Reconfigure services
- Create the repositories
- Extract OS media to newly created directories
- Control power management
- Restart servers

What Cobbler Offers

Cobbler sets up a PXE boot environment and controls everything related to the installation. When you use Cobbler to create a new machine it:

- Uses a template to configure the DHCP server
- Mirrors a repository or extracts a media to register the new OS
- Creates an entry in the DHCP configuration file with the IP and MAC addresses you specify for the new machine
- Creates necessary PXE files under the TFTP service directory
- Restarts the machine to begin the installation, if you enable power management

Cobbler knows how to extract necessary files from the distro ISO files and adjust the network services to boot up the new machines.

Kickstart

Kickstart templates let Red Hat and Fedora based systems automate the installation process. Cobbler works with Kickstart. You can use the Kickstart file to install machines in different domains and with different machines, by using configuration parameters (variables) such as $domain and $machine-name. A Cobbler profile can then specify, for example, domain-mydomain.com and specify the names of the machines that use this profile with the machine-name variables. All new machines in this Cobbler configuration will be installed with the same Kickstart configuration, and configured for the domain mydomain.com.

Fence Scripts

Cobbler can connect to power management environments such as blade center and ipmitool through fence scripts. When you reboot a new system, Cobbler runs the appropriate fence script for you.

Cobbler Architecture

Cobbler uses a set of registered objects to perform its work. Each registered object points to another object, inheriting the data of the object it points to. Cobbler uses the following types of objects.

- Distribution: represents an OS and contains information related to the kernel and initrd
- Profile: points to the distribution, a Kickstart file and other repositories
- System: this object represents the machine you're provision. The system object points to a profile or an image and includes specialized information such as the IP and MAC addresses and power management.
- Repository: this object stores the mirroring information (mirror URL) for a repository such as yum
- Image; this image can replace a distribution object for files that don't fit in that category. An example is where you can't device the files into a kernel and intird.

Deploying an Operating System using Cobbler and PXE Boot Cobbler comes with a great deal of functionality out of the box, although it can get complex due to the many technologies it can manage. You need to know PXE (see in this chapter) and the procedures for automatically installing the Linux distro you want to install. Installing Cobbler is quite easy through the yum utility (yum install cobbler).

Installing and configuring the Cobbler web interface is a good strategy if you need to perform regular activities with Cobbler.