O'REILLY®

Network Programmability and Automation

SKILLS FOR THE NEXT-GENERATION NETWORK ENGINEER

Jason Edelman,
Scott S. Lowe & Matt Oswalt

O'REILLY®

Network Programmability and Automation

SKILLS FOR THE NEXT-GENERATION NETWORK ENGINEER

Jason Edelman,
Scott S. Lowe & Matt Oswalt

Network Programmability and Automation

by Jason Edelman , Scott S. Lowe , and Matt Oswalt

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 .

- Editors: Mike Loukides and 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

- 2015-12-17: First Early Release

See http://oreilly.com/catalog/errata.csp?isbn=9781491931219 for release details.

978-1-491-93121-9

[FILL IN]

Chapter 1. Software Defined Networking

Are you new to Software Defined Networking (SDN)? Have you been living up in the SDN crazy for the past several years? Whichever bucket you fall into, do not worry. Even though this book is centered around network automation and programmability, this chapter is going to highlight and provide an introduction on several of the major tenets throughout the network industry that often end up in conversations on the topic of SDN. We'll get started by reviewing how Software Defined Networking made it into the main stream.

The Rise of Software Defined Networking

If there was one person that could be credited with all the change that is occurring in the network industry, it would be Martin Casado, who is currently a VMware Fellow, Senior Vice President, and General Manager in the Networking and Security Business Unit at VMware. He has had a profound impact on the industry, not just from his direct contributions including OpenFlow and Nicira, but by opening the eyes of large network incumbents and showing that network operations, agility, and manageability must change. Let's take a look at this in a little more detail.

OpenFlow

What is it?

For better or for worse, OpenFlow has served as the Hello World of the Software Defined Networking (SDN) movement. OpenFlow is the protocol that Martin Casado worked on while he was achieving his PhD at Stanford University under the supervision of Nick McKeown. It is simply a protocol that allows for the de-coupling of a network device's control plane from the data plane. In simplest terms, the control plane can be thought of as the brains of a network device and the data plane can be thought of as the hardware or ASICS that actually perform packet forwarding.

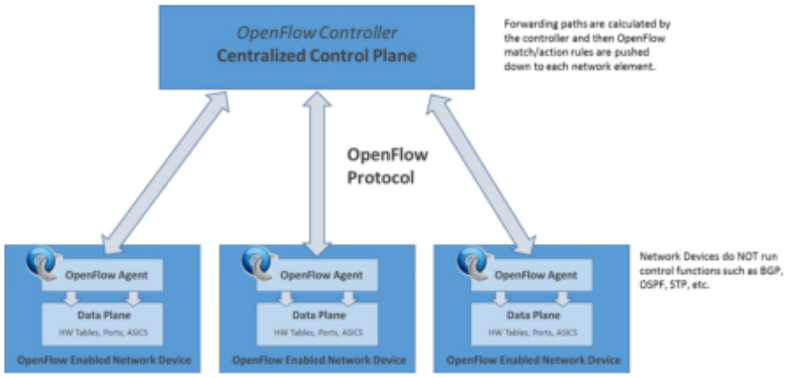


Figure 1-1. De-coupling the Control Plane and Data Plane with OpenFlow

Running OpenFlow in Hybrid Mode

The figure above depicts the network elements having no control plane. This represents a pure OpenFlow-only deployment. Many devices also support OpenFlow hybrid mode meaning OpenFlow can be deployed on a given port, VLAN, or even within a normal packet forwarding pipeline such that if there is not a match in the OpenFlow Table, then use the existing forwarding tables (MAC, Routing, etc.) making it more analogous to Policy Based Routing (PBR).

History of Programmable Networks

OpenFlow was not the first protocol or technology used to de-couple control functions and intelligence from network devices. There is a long history of technology and research that predates OpenFlow, although OpenFlow is the technology that started the SDN revolution. A few of the technologies that predated OpenFlow includes ForCes (Forwarding and Control Element Separation), Active Networks, Routing Control Platform (RCP), and Path Computation Element (PCE). For a more in-depth look at this history, take a look at a paper titled The Road to SDN: An Intellectual History of Programmable Networks by Jen Rexford, Nick Feamster, and Ellen Zegura.

What this means is OpenFlow is a low-level protocol that is used to directly interface with the hardware tables (example: Forwarding Information Base, or FIB) that exist on networking devices instructing the device how to forward traffic, e.g. traffic is destination 1 should egress port 45.

Because the tables OpenFlow uses support more than destination address as compared to traditional routing protocols, there is more granularity (matching fields in the packet) to determine the forwarding path, so the actual outcome is analogous Policy Based Routing (PBR) in hardware. Policy Based Routing is also a good example to use that shows what OpenFlow's true value strives to be — it is that to create an abstraction layer for the diverse set of hardware that exists in that new features can be rolled out that would be optimized in hardware independent of having purpose-built ASICS. As we know, the industry waited for years to get PBR in hardware as well as other features such as GRE in hardware. With OpenFlow abstracting out the underlying hardware, it becomes possible to enhance the capabilities of the network infrastructure without waiting for the next version of hardware from the manufacturer.

Note

Policy Based Routing has been around for years and allowed network administrators to create route policies based on source/destination address, protocol, etc., but until recent years, the caveat was all policy based network traffic ended up being software switched hitting the CPU, not to mention what it does to the complexity of the network. Leveraging OpenFlow simplifies this.

Note

OpenFlow is a low level protocol manipulating flow tables directly impacting packet forwarding. OpenFlow was not built to be a management plane protocol to configure general parameters on devices such as passwords, SNMP, AAA, etc. although there are some ramblings of a new spec called OF-config that may take some of this into consideration

Why OpenFlow?

While it's important to understand what OpenFlow is, it's even more important to understand the reasoning behind the research and development effort of the original OpenFlow spec that led to the rise of Software Defined Networking.

Martin Casado had a job working for the national government while he was attending Stanford. During his time working for the government, there was a need to react to security attacks on the IT systems (after all, this is the US government). Casado quickly realized that he was able to program and manipulate the computers and servers as he needed. The actual use cases were never publicized, but it was this type of control over endpoints that made it possible to react, analyze, and potentially re-program a host or group of hosts when and if needed.

When it came to the network, it was near impossible to do this in a clean and programmatic fashion. After all, each network device only had a Command Line Interface (CLI). Although the CLI was and is still very well known and even preferred by network administrators, it was clear to Casado that it did not offer the flexibility required to truly manage, operate, and secure the network.

In reality, the way networks were managed had almost changed ever nearly 20 years minus adding CLI commands for new features. The biggest change was the migration from telnet to SSH, which is often a joke SDN startup Big Switch Networks uses in their slides as you can see in the Figure below.

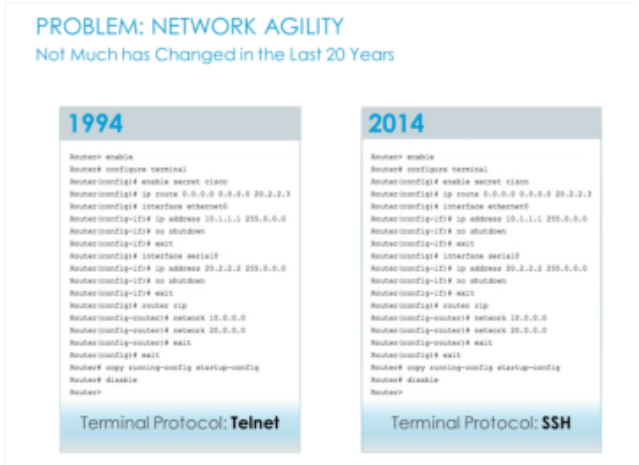


Figure 1-2. What's Changed? From Telnet to SSH. Source: Big Switch Networks

All joking aside, the management of networks has lagged behind other technologies quite drastically, and this is what Martin eventually set out to change over the next several years. This lack in manageability is often better understood when other technologies are examined. Other technologies almost always have more modern ways of managing a large number of devices for both configuration management and data gathering and analysis. For example, hypervisor managers, wireless controllers, IP PBXs, PowerShell, DevOps tools, and the list can go on. Some of these are tightly coupled from a certain vendor, but others are more loosely aligned to allow for multi-platform management, operations, and agility.

If we go back to the scenario while Martin was working for the government, was it possible to re-direct traffic based on application? Did network devices have an API? Was there a single point of communication to the network? The answers were largely no across the board. How could it be possible to program the network to dynamically control packet forwarding, policy, and configuration as easily as it was to write a program and have it execute on an end host machine?

The initial OpenFlow spec was the result of experiencing these types of problems first hand by Martin Casado. While it is now 2015 and the hype around OpenFlow has died down quite a bit, since the industry is starting to finally focus more on use cases and solutions than low level protocols, it is this initial work that was the catalyst for the entire industry to do a re-think on how networks are built, managed, and operated. Thank you, Martin.

This also means if it wasn't for Martin Casado, this book would probably not have been written, but we'll never know now!

What is Software Defined Networking?

We had an introduction to OpenFlow, but what is Software Defined Networking? Are they the same thing, different things, or neither? To be honest, Software Defined Networking is just like Cloud: was nearly a decade ago before we knew about different types of Cloud like Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

Having reference examples and designs streamline the understanding of what cloud was and is — but even before these terms did exist, it could be defended that when you saw Cloud, you know it. That's kind of where we are with Software Defined Networking. There are public definitions that exist that state white-box networking is SDN or that having an API on a network device is SDN. Are they REALLY SDN? Not really.

Rather than attempt to provide a definition of SDN, the ten technologies and trends that are very often thought of as SDN will be covered. They include:

- OpenFlow
- Network Functions Virtualization
- Virtual Switching
- Network Virtualization
- Device APIs
- Network Automation
- Bare-Metal Switching
- Data Center Network Fabrics
- SD-WAN
- Controller Networking

Of these trends, the rest of the book will focus on network automation, APIs, and peripheral technologies that are critical in understanding how all of the pieces come together when using network devices that expose programmatic interfaces with modern automation tools and instrumentation.

OpenFlow

Even though we introduced OpenFlow earlier, we'll quickly provide another quick summary since it has played such a major part in shaping the SDN market.

Separating the control plane from the data plane with OpenFlow is how an SDN purist would still define SDN (not that it is right or wrong), but it is where the SDN movement began. This type of architecture largely revolves around the evolved OpenFlow standard that has come along way from the days of the initial testing that occurred at Stanford.

One of the major benefits that was supposed to be an outcome of using a protocol like OpenFlow between a controller and network devices was that there would be true vendor independence of the controller software, sometimes referred to as a network operating system, and the underlying virtual and physical network devices. What has actually happened though is that vendors who use OpenFlow in their solution, i.e. Big Switch Networks, HP, and NEC have developed OpenFlow extensions due to the pace of standards and the need to provide unique value-added features that the off-the-shelf version of OpenFlow does not offer. It is yet to be seen if all of the extensions end up making it into future version of the OpenFlow standard.

When OpenFlow is used, you do gain the benefit to getting more granular with how traffic traverses the network, but with great power comes great responsibility. This is great if you have a team developers. For example Google has rolled out an OpenFlow based WAN called B4 that increases efficiency of their WAN to nearly 100%. For most other organization, the use of OpenFlow or any other given protocol will be less important than what an overall solution offers to the business you are supporting.

Note

While this particular section is called OpenFlow architecturally it's about de-coupling the control plane from the data plane. OpenFlow is just the main protocol being used to accomplish this functionality.

Network Functions Virtualization (NFV)

Network Functions Virtualization, known as NFV, is a simple concept. It is being able to take functions that have been traditionally deployed as hardware, but now deploy them as software. The most common examples of this are virtual machines that operate as routers, firewalls, load balancers, IDS/IPS, VPN, application firewalls, and any other service/function.

With NFV, it becomes possible to breakdown a monolithic piece of hardware that may have been 10s or 100s of thousands of dollars with 100s to 1000s lines of commands to get it configured into N pieces of software, namely virtual appliances. These smaller devices become much more manageable from an individual device perspective.

Note

The example above uses virtual appliances as the form factor for NFV enabled devices. This is merely an example. Deploying network functions as software could come in many forms including embedded in a hypervisor, as a container, or as an application.

It's not uncommon to deploy hardware that *may* be needed in 3-5 years just in case because it's too complicated and even more expensive to have gradual upgrades. So not only is hardware an intensive capital cost, it's only used for the what-if scenarios if growth occurs. Deploying software based, or NFV, solutions, offer a better way to scale out and minimize the failure domain of a network or particular application while using a *pay-as-you-grow* model.

If NPV could offer so much benefit, why haven't there been more solutions and products that fit into this category deployed in production? There are actually a few different reasons. First, it requires a re-think in how the network is architected. When there is a single monolithic firewall (as an example), everything goes through that firewall. All applications and all users. If not all, a defined set that you are aware of. In the modern NPV model where there could be many virtual FWs deployed, there is a FW per application or tenant as opposed to a single big box FW. This makes the failure domain per FW, or any other services appliances, fairly small, and if a change is being made or a new application is being rolled out, no change is required for the other per-application (per-tenant) based FWs.

On the other hand, in the more traditional world of having monolithic devices, there is essentially a single pane of management for security policy — single CLI or GUI. This could make the failure domain immense, but it does offer administrators streamlined policy management since it's only a single device being managed. Based on the team or staff supporting these devices, they may opt still for a monolithic approach. That is the reality, but hopefully over time with improved tools that can help with the consumption and management of software centric solutions, as an industry, we'll see more deployments leveraging this type of technology.

Aside from management, another factor that plays into this is that many vendors are not actively selling their *virtual appliance* edition. If a vendor has had a hardware business for the past several years, it's a drastic shift to a software-led model. Because of this, many of these vendors are limiting the performance or features on their virtual appliance based technology.

As will be seen in many of these technology areas, a major value of NFV is in agility too. By eliminating hardware, the time to provision new services are decreased by removing the time needed to rack, stack, cable, and integrate into an existing environment. Leveraging a software approach, it becomes as fast as deploying a new virtual machine into the environment and an inherent benefit of this approach is being able to clone and backup the virtual appliance for further testing, for example in Disaster Recovery (DR) environments.

Virtual Switching

The more common virtual switches on the market these days include the VMware standard switch (VSS), VMware distributed switch (VDS), Cisco Nexus 1000V, Cisco Application Virtual Switch (AVS), and the open source Open vSwitch (OVS).

These switches every so often get wrapped into the SDN discussion, but in reality they are software-based switches that reside in the hypervisor kernel providing local network connectivity between virtual machines (and now containers). They provide functions such as MAC learning and features like Link Aggregation, SPAN, and sFlow just like their physical switch counterparts have been doing for years. While these virtual switches are often found in more comprehensive SDN and Network Virtualization solutions, by themselves they are a switch that just happens to be running in software. While virtual switches are not **technically** SDN on their own, they are extremely important as we move forward as an industry. They've created a new access layer, or new edge, within the data center. No longer is the network edge the physical top of rack (TOR) switch that is hardware-defined with limited flexibility (in terms of feature/function development). Since the new edge is software-based through the use of virtual switches, it offers the ability to more rapidly create new network functions in software, and because of this, it is possible to distribute policy more easily throughout the network. As an example, security policy can be deployed to the virtual switch port, that is nearest point to the actual host, be it a virtual machine or container, to further enhance the security of the network.

Network Virtualization

Solutions that are categorized as Network Virtualization have become synonymous with Software Defined Networking (SDN) solutions. For purposes of this section, network virtualization refers to software only overlay-based solutions. The popular solutions that fall into this category some from vendors like VMware, Nuage, PLUMgrid, and Midokura.

A key characteristic of these solutions is that an overlay-based protocol such as Virtual eXtensible LAN (VXLAN) is used to build connectivity between hypervisor-based virtual switches. This connectivity and tunneling approach provides Layer 2 adjacency between virtual machines that exist on different physical hosts independent of the physical network meaning the physical network could be Layer 2, Layer 3, or a combination of both. The result is a virtual network that is de-coupled from the physical network and that is meant to provide choice and agility.

While the overlay is just minor implementation detail of network virtualization solutions, these solutions are much more than just virtual switches being stitched together by overlays. These solutions are usually comprehensive offering security, load balancing, and integrations back into the physical network all with a single point of a management, i.e. the controller. Often times these solutions offer integrations with the best of breed Layer 4-7 services companies as well offering choice as to which technology could be deployed within network virtualization platforms.

Agility is also achieved because of the use of the central controller platform that is used to dynamically configure each virtual switch and services appliances as needed. If you recall, the network has lagged behind operationally due to the CLI that is pervasive across all vendors in the physical world. In network virtualization, there is no need to configure virtual switches manually as each solution simplifies this process by providing a central GUI, CLI, but also an API where changes can be made programmatically.

In Chapter 7, [Link to Come] where we cover APIs, we'll be taking a look at controller based APIs that exist in some of the network virtualization based solutions.

Device APIs

Over the past several years, vendors have begun to realize that just offering a standard CLI was not going to cut it anymore and that using a CLI has severely held back operations. If you have ever worked with any programming or scripting language, you can probably understand that. For those that haven't, we'll also be taking a look at what it has been like writing scripts that communicate via SSH and SNMP in Chapter 7, [Link to Come].

The major pain point is that scripting with legacy, or CLI based network devices, is not object oriented and does not returned structured data. This meant data would be returned from the device to a script in a raw text format, i.e. the output of a show version, and then the individual writing the script would need to parse that text manually to extract attributes such as uptime or operating system version. While this approach is all administrators have had, vendors are gradually migrating to API driven network devices.

Offering an API eliminates the need to parse raw text as data is returned back from a network device significantly reducing the time it takes to write a script. Rather than parsing through text to find the uptime or any other attribute, an object is returned providing exactly what is needed. Not only does it reduce the time to write a script lowering the barrier to entry for network engineers (non-programmers), but it also provides a cleaner interface such that professional software developers can rapidly develop and test code must like how they operate using other APIs on non-network devices. By testing code, this could mean testing new topologies, certifying new network features, validating particular network configurations, and the list can go on. These are all things that are done manually today and are very time consuming and error prone.

One of the first APIs into the network scene was that by Arista Networks. It's API is called eAPI, which is a REST & JSON based API. Don't worry, REST and JSON will be covered in chapters to follow! Since Arista, we've seen Cisco announce APIs such as Nexus NX-API and onePK and a vendor like Juniper who has had an extensible NETCONF interface all along, but hasn't publicly drawn as much attention to it. It's worth noting that nearly every vendor out there has some sort of API these days.

This topic will be covered in much more detail in Chapter 7, [Link to Come].

Network Automation

As APIs in the network world continue to evolve, more interesting use-cases for taking advantage of them will also continue to emerge. In the near term, network automation is a prime candidate for taking advantage of the programmatic interfaces being exposed my modern network devices that offer an API.

To put it in greater context, network automation is **NOT** just about automating the configuration of network devices. It is true that is the most common perception of network automation, but using APIs and programmatic interfaces can automate and offer much more than pushing configuration parameters.

Leveraging an API streamlines the access to all of the data bottled up in network devices. Think about data such as flow level data, routing tables, FIB tables, interface statistics, MAC tables, VLAN tables, serial numbers, and the list can go on and on. Using modern automation techniques that in turn leverage an API can quickly aid in the day to day operations of managing networks for data gathering and automated diagnostics. On top of that, since an API is being used that returns structured data, as an administrator, you will have the ability to display and analyze the exact data set you want and need, even coming from various aforementioned disparately reducing the time it takes to debug and troubleshoot issues on the network. Rather than connecting to N routers running BGP trying validate a configuration or troubleshoot an issue, automation techniques can be used to simplify this process.

Additionally, leveraging automation techniques leads to a more predictable and uniform network as a whole. This can been seen by automating the creation of configuration files, automating the creation of a VLAN, or automating the process of troubleshooting. It streamlines the process for all users supporting a given environment instead of having each network administrator having their own best practice.

The various types of network automation will be covered in the next chapter, Chapter 1 in much greater depth.

Bare Metal Switching

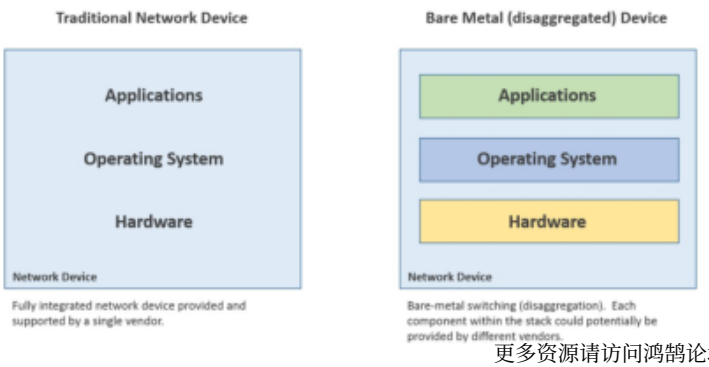
The topic of bare-metal switching is often thought of as Software Defined Networking (SDN), but it's not. Really, it isn't! That said, in the effort of trying to give an introduction to the various technology trends that are perceived as SDN, it needs to be covered. If we re-wind back to 2014 (and even earlier), the phrase used to describe bare-metal switching was white-box or commodity switching. The phrase has changed and that is not without good reason.

Before we cover the change from white-box to bare metal, it's important to understand the high level of what this means since it's a massive change in how network devices are thought of. Network devices for the last 20 years were always bought as a physical device — these physical device came as a hardware appliances, an operating system, and features/applications that you can use on the system. These components all came from the same vendor.

In the white-box and bare-metal network devices, the device looks more like an x86 server. It offers the user to disaggregate each of the required components meaning it is possible to purchase hardware from one vendor, an operating system from another, and then load features/apps from other vendors or even the open source community.

White-box switching was a hot topic for a period of time during the OpenFlow hype since the intent was to commoditize hardware and centralize the brains of the network in an OpenFlow controller, otherwise now known as a SDN controller. And in 2013, Google announced they had built their own switches and were contesting them with OpenFlow! This was the topic of a lot of industry conversations of the time, but in reality, not every end user is Google, so not every user will be building their own hardware and software platforms.

In parallel to these efforts, we saw the emergence of a few companies that were solely focused on providing solutions around white-box switching. They include Big Switch Networks, Cumulus Networks, and Pica8. Each of them offer software-only solutions, so they still need hardware that their software will run on to provide an end-to-end solution. Initially, these white-box hardware platforms came from Original Direct Manufacturers (ODMs) such as Quanta, Super Micro, and Accton. If you've been in the network industry, it is more than likely you've never even heard of those vendors.



Fully integrated network device provided and supported by a single vendor.

Bare-metal switching (disaggregation). Each component within the stack could potentially be provided by different vendors.

Figure 1-3. A Look at Traditional and Bare-Metal Switching Stacks

It wasn't until Cumulus and Big Switch announced partnerships with companies including HP and Dell where the industry started to shift from selling this trend white-box to bare-metal since now name-brand vendors were supporting 3rd party operating systems from the likes of Big Switch and Cumulus Networks on their hardware platforms.

There still may be confusion on why bare-metal is technically not SDN since a vendor like Big Switch plays in both worlds. The answer is simple. If there is a controller integrated with the solution using a protocol such as OpenFlow (it does not **have** to be OpenFlow), and it is programmatically communicating with the network devices, that gives it the flavor of Software Defined Networking. This is what Big Switch does — they need to load software on the bare-metal/white-box hardware running an OpenFlow agent that then communicates with the controller as part of their solution.

On the other hand, Cumulus Networks provides a Linux distribution purpose-built for network switches. This distribution, or operation system, runs traditional protocols such as LLDP, OSPF, and BGP, with no controller requirement whatsoever making it more comparable, and compatible, to non-SDN based network architectures.

With this description it should be evident that Cumulus is a network operating system (OS) company that runs their software on bare-metal switches while Big Switch is a bare-metal based SDN company requiring the use of their SDN controller, but also leverages 3rd party bare-metal switching infrastructure.

During the time of this writing , Canonical just announced a version of Ubuntu to serve as a network operating system as well. Just shows how dynamic this area is, but having a specific version of Ubuntu built for network switches makes it a little classier on what bare-metal/white-box switching is all about — it's about disaggregation and having the ability to purchase network hardware from one vendor and load software from another should you choose to do so. In this case, administrators are offered the flexibility to change designs, architectures, and software, without swapping out hardware, just the underlying operating system. That is pretty slick..

Data Center Network Fabrics

Have you ever had the concern that you could not easily interchange the various network devices in a network if they were all running standard protocols such as Spanning Tree or OSPF. If you have or haven't, you are not alone! Imagine having a data center network with a collapsed core and individual switches at the top of each rack. Now think about the process that needs to happen when it's time for an upgrade.

There are many ways to upgrade networks like this, but what if it was just the top of rack (TOR) switches that needed to be upgraded and in the evaluation process for new TOR switches, it was decided a new vendor or platform would be used. This is 100% normal and has been done time and time again. The process is simple - interconnect the new switches to the existing core (of course, we are assuming there are available ports in the core) and properly configure 802.1Q trunking if it's a Layer 2 interconnect or configure your favorite routing protocol if it's a Layer 3 interconnect.

Enter Data Center Network Fabrics.

Data Center Network Fabrics aim to change the mindset of network operators from managing individual boxes one at a time to managing a system in its entirety. If uses the example from above, it would not be possible to swap out a TOR switch, which is just a single component of a data center network. Rather, when the network is deployed and managed as a system, it needs to be thought of as a system. This means the upgrade process would be to migrate from system to system, or fabric to fabric. In the world of fabrics, fabrics can be swapped out when it's time for an upgrade, but the individual components within the fabric cannot be – at least most of the time. It may be possible when a specific vendor is providing a migration or upgrade path and when bare-metal switching (only replacing HW) is being used.

In addition to treating the network as a system, a few other common attributes of data center networking fabrics are:

- They offer a single interface to manage or configure the fabric
- They offer distributed default gateways across the fabric
- They offer multi-pathing using Layer 2 and/or Layer 3 protocols
- They use some form of SDN controller to manage the system

Note

Not all attributes may exist on all fabrics.

Note

This is not an all encompassing list of attributes that define a network fabric.

SD-WAN

One of the hottest trends in Software Defined Networking right now (mid-2015) is Software Defined Wide Area Networking (SD-WAN). Over the past 18 months there has been a growing number of companies that have launched to tackle the problem of Wide Area Networking. A few of these vendors include Viptela, CloudGenix, VeloCloud, Cisco, Glue Networks, and Silverpeak.

The WAN has not seen a radical shift in technology since the migration from Frame Relay to MPLS. With broadband and Internet costs being a fraction of what costs are for equivalent private line circuits, there has been an increase in leveraging site to site VPN tunnels over the years, laying the ground work for the next big thing in Wide Area Networking.

Common designs for remote offices typically include a private (MPLS) circuit and/or a public Internet connection. When both exist, Internet is usually used as backup only, specifically for guest traffic, or for general data riding issue over a VPN to corporate while the MPLS circuit is used for low latency applications such as voice or video communications. When traffic starts to get divided between circuits, this increases the complexity of the routing protocol configurations and also limits the granularity of how to route to the destination address. The source address, application, and real time performance of the network is usually not taken into consideration when deciding the best path to take.

A common SD-WAN architecture that many of the modern solutions are is similar to that of network virtualization used in the data center in that an overlay protocol is used to inter-connect the SD-WAN edge devices. These overlays are used, the solution is agnostic to the underlying physical transport making SD-WAN functional over the Internet or a private WAN. These solutions often ride over 2 or more Internet circuits at branch sites fully encrypting traffic using IPSec. Additionally, many of these solutions constantly measure the performance of each circuit in use being able to rapidly fail over between circuits for specific applications even during brown outs. Since there is application layer visibility, administrators can easily pick and choose which application should take a particular route, not having to solely use destination based routing that increases the complexity using OSPF or BGP on the WAN routers.

Note

While many of the SD-WAN vendors leverage overlay technology, not every vendor does. For example, Cisco and Glue Networks do not use overlay technology in their solutions.

From an architecture standpoint, the SD-WAN solutions also typically offer various forms of Zero Touch Provisioning (ZTP) and centralized management with a portal that exists on-premises or in the cloud as a SaaS based application.

A valuable byproduct of using SD-WAN technology is that it offers more **choice** for end users since basically any carrier or type of connection can be used over the WAN and across the Internet. In doing so, it also simplifies the configuration and complexity of carrier networks, which in turn will allow carriers to simplify their internal design and architecture, hopefully reducing their costs. Going one step further from a technical perspective, all logical network constructs such as Virtual Routing Forwarding (VRFs) would be managed via the controller platform's User Interface (UI) that the SD-WAN vendor provides, again, eliminating the need to wait weeks for carriers to respond back to you when changes are required.

Controller Networking

When it comes to several of these trends, there is some overlap as you may have realized. That is one of the confusing parts when trying to understand all of the new technology and trends that have emerged over the last few years.

For example, popular network virtualization platforms use a controller as do several solutions that fall into the data center network fabric, SD-WAN, and bare-metal switch solutions too. Confusing? You may be wondering why controller-based networking has been broken out by itself. In reality, it often times is just characteristic and a mechanism to deliver modern solutions, but not all of the previous trends cover all of what controllers can deliver from a technology perspective.

For example, a very a popular open source Software Defined Networking (SDN) controller is OpenDaylight (ODL). ODL, as with many other controllers, are platforms, not products. They are platforms that can offer specialized applications such as network virtualization, but they can also be used for network monitoring, visibility, tap aggregation, or any other function in conjunction with applications that sit on top of the controller platform.

This is the core reason why it's important to understand what controllers can offer above and beyond being used for more traditional applications such as fabrics, network virtualization, and SD-WAN.

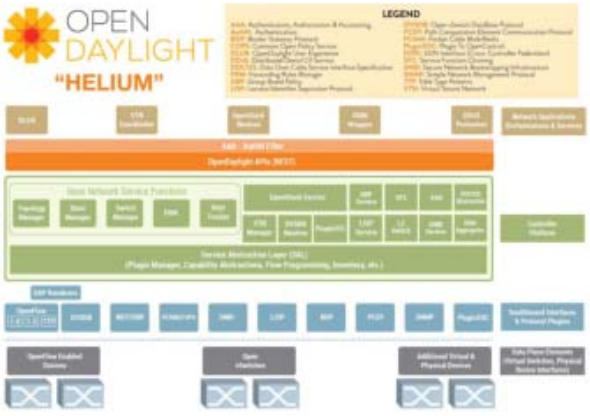


Figure 1-4. OpenDaylight Architecture

Summary

There you have it. That is an introduction to the trends and technologies that are most often categorized as Software Defined Networking (SDN). Dozens of SDN start-ups were created over the past 7 years, millions of VC money invested, and billions spent on acquisitions of these companies. It's been unreal, and if we break it down one step further, it's all with the common goal of leveraging software principles and technology to offer greater power, control, agility, and choice to the users of the technology while increasing the operational efficiencies.

In the next chapter, Chapter 1, we'll take a look at Network Automation and start to dive deeper into the various types of automation, some common protocols and APIs, and also how automation has started to evolve in the last several years.

Chapter 2. Linux

Linux in a Network Automation Context

You might be wondering why we've included a chapter about Linux in a book on network automation and programmability. After all, what in the world does Linux, a UNIX-like operating system, have to do with network automation and programmability? There are several reasons why we felt this content was important.

First, several modern network operating systems (NOSes) are based on Linux, although some use a custom command line interface (CLI) that means they don't look or act like Linux. Others, however, do expose the Linux internals and/or use a Linux shell such as `bash`.

Second, some new companies and organizations are bringing to market full Linux distributions that are targeted at network equipment. For example, the OpenCompute Project (OCP) recently selected Open Network Linux (ONL) as the default Linux distribution that will power their open source network hardware. Cumulus Networks is another example, offering their Debian-based Cumulus Linux as a NOS for supported hardware platforms. As a network engineer, the possibility is growing that you'll need to know Linux in order to configure your network.

Third, and finally, many of the tools that we discuss in this book have their origins in Linux, or require that you run them from a Linux system. For example, Ansible (a tool we'll discuss in Chapter 9, [Chapter to Come]) requires Python (a topic we'll discuss in Chapter 4, Chapter 3). For a few different reasons we'll cover in Chapter 9, when automating network equipment with Ansible you'll run Ansible from a network-attached system running Linux, and *not* on the network equipment directly. Similarly, when you're using Python to gather and/or manipulate data from network equipment, you'll often do so from a system running Linux.

For these reasons, we felt it was important to include a chapter that seeks to accomplish the following goals:

- Provide a bit of background on the history of Linux
- Briefly explain the concept of Linux distributions
- Introduce you to Bash, one of the most popular Linux shells available
- Discuss Linux networking basics
- Dive into some advanced Linux networking functionality

Keep in mind that this chapter is not intended to be a comprehensive treatise on Linux or the Bash shell; rather, it is intended to get you "up and running" with Linux in the context of network automation and network programmability. Having said that, let's start our discussion of Linux with a very brief look at the history and origins of Linux.

A Brief History of Linux

The story of Linux is a story with a number of different threads. As such, the story varies depending on the perspective from which it is told.

Some say that Linux started out in the early 1980s, when Richard Stallman launched the GNU Project as an effort to provide a free UNIX-like operating system (OS). GNU, by the way, stands for "GNU's Not UNIX," a recursive acronym Stallman created to describe the free UNIX-like OS he was attempting to create. Stallman's GNU General Public License (GPL) came out of the GNU Project's efforts. Although the GNU Project was able to create free versions of a wide collection of UNIX utilities and applications, the kernel—known as GNU Hurd—for the GNU Project's new OS never gained momentum.

Others, therefore, point to Linus Torvalds' efforts to create a MINIX clone in 1991 as the start of Linux. Driven by the lack of a free OS kernel, his initial work rapidly gained support, and in 1992 was licensed under the GNU GPL with the release of version 0.99. Since that time, the kernel he wrote (named Linux) has been the default OS kernel for the software collection created by the GNU Project.

Because Linux originally referred only to the OS kernel and needed the GNU Project's software collection to form a full operating system, some people suggested that the full OS should be called "GNU/Linux," and some organizations still use that designation today (Debian, for example). By and large, however, most people just refer to the entire OS as Linux, and so that's the convention that we will follow in this book.

Linux Distributions

As you saw in the previous section, the Linux operating system is made up of the Linux kernel plus a large collection of open source tools primarily developed as part of the GNU Project. The bundling together of the kernel plus a collection of open source software led to the creation of Linux *distributions*. A distribution is the combination of the Linux kernel plus a selection of open source utilities, applications, and software packages that are bundled together and distributed together (hence the name *distribution*). Over the course of Linux's history, a number of Linux distributions have come and gone (anyone remember Slackware?), but as of this writing there are two major branches of Linux distributions: the Red Hat/CentOS branch and the Debian and Debian derivative branch.

Red Hat Enterprise Linux, Fedora, and CentOS

Red Hat was an early Linux distributor who became a significant influencer and commercial success in the Linux market, so it's perfectly natural that one major branch of Linux distributions is based on Red Hat.

Red Hat offers a commercial distribution, known as Red Hat Enterprise Linux (RHEL), in addition to offering technical support contracts for RHEL. Many organizations today use RHEL because it is backed by Red Hat, focuses on stability and reliability, offers comprehensive technical support options, and is widely supported by other software vendors.

However, the fast-moving pace of Linux development and the Linux open source community is often at odds with the slower and more methodical pace required to maintain stability and reliability in the RHEL product. To help address this dichotomy, Red Hat has an upstream distribution known as Fedora. We refer to Fedora as an "upstream distribution" because much of the development of RHEL and RHEL-based distributions occurs in Fedora, then flows "down" to these other products. In coordination with the broader open source community, Fedora sees new kernel versions, new kernel features, new package management tools, and other new developments first; these new things are tested and vetted in Fedora before being migrated to the more enterprise-focused RHEL distribution at a later date. For this reason, you may see Fedora used by developers and other individuals who need the "latest and greatest", but you won't often see Fedora used in production environments.

Although RHEL and its variants are only available from Red Hat through a commercial arrangement, the open source license (the GNU GPL) under which Linux is developed and distributed *requires* that the source of Red Hat's distribution be made publicly available. A group of individuals who wanted the stability and reliability of RHEL but without the corresponding costs imposed by Red Hat took the RHEL sources and created CentOS. (CentOS is a named formed out of "Community Enterprise OS.") CentOS is freely available without cost, but—like many open source software packages—does not come with any form of technical support. For many organizations and many use cases, the support available from the open source community is sufficient, so it's not uncommon to see CentOS used in a variety of environments, including enterprise environments.

One of the things that all of these distributions (RHEL, Fedora, and CentOS) share is a common *package format*. When Linux distributions first started emerging, one key challenge that had to be addressed was the way in which software was packaged with the Linux kernel. Due to the breadth of free software that was available for Linux, it wasn't really effective to ship *all* of it in a distribution, nor would users necessarily *want* all of the various pieces of software installed. If not all of the software was installed, though, how would the Linux community address dependencies? A *dependency* is a piece of software required to run another piece of software on Linux. For example, some software might be written in Python, which of course would require Python to be installed. To install Python, however, might require other pieces of software to be installed, and so on. As an early distributor, Red Hat came up with a way to combine the files needed to run a piece of software along with additional information about that software's dependencies into a single package—a *package format*. That package format is known as an RPM, perhaps so named after the tool originally used to work with said packages: RPM Manager (formerly Red Hat Package Manager), whose executable name was simply `rpm`. All of the Linux distributions we've discussed so far—RHEL, CentOS, and Fedora—leverage RPM packages as their default package format, although the specific tool used to work with such packages has evolved over time.

RPM's successors

We mentioned that RPM originally referred to the actual package manager itself, which was used to work with RPM packages. Most RPM-based distributions have since replaced the `rpm` utility with newer package managers that do a better job of understanding dependencies, resolving conflicts, and installing (or removing) software from a Linux installation. For example, RHEL/CentOS/Fedora moved first to a tool called `yum` (short for "Yellowdog Updater, Modified), and are now migrating again to a tool called `dnf` (which stands for "Dandified YUM").

Other distributions also leverage the RPM package format, such as Oracle Linux, Scientific Linux, and various SUSE Linux derivatives.

RPM portability

You might think that because a number of different Linux distributions all leverage the same package format (RPM), that RPM packages are portable across these Linux distributions. In theory, this is possible, but in practice it rarely works. This is usually due to slight variations in package names and package versions across the distributions, which makes resolving dependencies and conflicts practically impossible.

Debian, Ubuntu, and Other Derivatives

Debian GNU/Linux is a distribution produced and maintained by The Debian Project. The Debian Project was officially founded by Ian Murdock

on August 16, 1993, and the creation of Debian GNU/Linux was funded by the Free Software Foundation's GNU Project from November 1994 through November 1995. To this day, Debian remains the only major distribution of Linux that is not backed by a commercial entity. All Debian GNU/Linux releases since version 1.1 have used a code name taken from a character in one of the *Toy Story* movies. Debian GNU/Linux 1.1, released in June 1996, was code-named "Buzz." The most recent version of Debian GNU/Linux, version 8.0, was released in April 2015 and is code-named "Jessie."

Debian GNU/Linux offers three branches: Stable, Testing, and Unstable. The Testing and Unstable branches are rolling releases that will, eventually, become the next Stable branch. This approach results in a typically very high-quality release, and could be one of the reasons that a number of other distributions are based on (*derived* from) Debian GNU/Linux.

One of the more well-known Debian derivatives is Ubuntu Linux, started in April 2004 and funded in large part by Canonical Ltd., a company founded by Mark Shuttleworth. The first Ubuntu release was in October 2004, was released as version 4.10 (the "4" denotes the year, and the "10" denotes the month of release), and was code-named "Warty Warthog." All Ubuntu code-names are built using an adjective and an animal with the same first letter (Warty Warthog, Hoary Hedgehog, Breezy Badger, etc.). Ubuntu was initially targeted as a usable desktop Linux distribution, but now offers both desktop-, server-, and mobile-focused versions. Ubuntu uses time-based releases, releasing a new version every 6 months and a long-term support (LTS) release every two years. LTS releases are supported by Canonical and the Ubuntu community for a total of 5 years after release. All releases of Ubuntu are based on packages taken from Debian's unstable branch, which is why we refer to Ubuntu as a Debian derivative.

Speaking of packages: like RPM-based distributions, the common thread across the Debian and Debian derivatives—probably made clear by the term "Debian derivatives"—is that they share a common package format, known as the Debian package format (and noted by a `.deb` extension on the files). The founders of the Debian Project created the DEB package format and the `dpkg` tool to solve the same problems that Red Hat attempted to solve with the RPM package format. Also like RPM-based distributions, Debian-based distributions evolved past the use of the `dpkg` tool directly, first using a tool called `dselect` and then moving on to the use of the `apt` tool (and programs like `apt-get` and `aptitude`).

Debian package portability

Just as with RPM packages, the fact that multiple distributions leverage the Debian package format (typically noted with a `.deb` extension) doesn't mean that Debian packages are necessarily portable between distributions. Slight variations in package names, package versions, file paths, and other details will typically make this very difficult, if not impossible.

A key feature of the `apt`-based tools is the ability to retrieve packages from one or more remote *repositories*, which are on-line storehouses of Debian packages. The `apt` tools also feature better dependency determination, conflict resolution, and package installation (or removal).

Other Linux Distributions

There are other distributions in the market, but these two branches—the Red Hat/Fedora/CentOS branch and the Debian/Ubuntu branch—cover the majority of Linux instances found in organizations today. For this reason, we'll focus only on these two branches throughout the rest of this chapter. If you're using a distribution not from one of these two major branches—perhaps you're working with SuSE Enterprise Linux, for example—keep in mind there may be slight differences between the information contained here and your specific distribution. You should refer to your distribution's documentation for the details.

Now that we've provided an overview of the history of Linux and Linux distributions, let's shift our focus to interacting with Linux, focusing primarily on interacting via the shell.

Interacting with Linux

As a very popular server OS, you could use Linux in a variety of ways across the network. This could take the form of receiving IP addresses via a Linux-based DHCP server, accessing a Linux-powered web server running the Apache HTTP server or Nginx, or by utilizing a Domain Name System (DNS) server running Linux in order to resolve domain names to IP addresses. There are, of course, *many* more examples; these are just a few. In the context of our discussion of Linux, though, we're going to focus primarily on interacting with Linux via the *shell*.

The shell is what provides the command-line interface (CLI) by which most users will interact with a Linux system. Linux offers a number of shells, but the most common shell is Bash, the Bourne Again Shell (a pun on the name of one of the original UNIX shells, the Bourne Shell). In the vast majority of cases, unless you've specifically configured your system to use a different shell, when you're interacting with Linux you're using Bash. In this section, we're going to provide you with enough basic information to get started interacting with a Linux system's console, and we'll assume that you're using Bash as your shell. If you are using a different shell, please keep in mind that some of the commands and behaviors we describe below might be slightly different.

A good Bash reference

Bash is a topic about which an entire book could be written. In fact, one already has—and is now in its third edition. If you want to learn more about Bash than we have room to talk about in this book, we *highly* recommend O'Reilly's *Learning the bash Shell, 3rd Edition*.

We've broken our discussion of interacting with Linux into 4 major areas:

- Navigating the file system
- Manipulating files and directories
- Running programs
- Working with background services, known as *daemons*

Let's start with navigating the file system.

Navigating the File System

Linux uses what's known as a single-root file system, meaning that all of the drives and directories and files in a Linux installation fall into a single namespace, referred to quite simply as `/`. (When you see `/` by itself, say "root" in your head.) This is in stark contrast to an OS like Microsoft Windows, where each drive has its own root (the drive letter, like `C:\` or `D:\`).

Everything is treated like a file

Linux follows in UNIX's footsteps in treating everything like a file. This includes storage devices (which are treated as block devices), ports on the computer (like serial ports), or even input/output devices. Thus, the importance of a single-root file system—which encompasses devices as well as storage—becomes even more important.

Like most other OSes, Linux uses the concept of directories (known as *folders* in some other OSes) to group files in the file system. Every file resides in a directory, and therefore every file has a unique *path* to its location. To denote the path of a file, you start at the root and list all the directories it takes to get to that file, separating the directories with a forward slash. For example, the command `ping` is often found in the `bin` directory off the root directory. The path, therefore, to `ping` would be noted like this:

```
/bin/ping
```

In other words, start at the root directory (`/`), continue into the `bin/` directory, and find the file named `ping`. Similarly, on Debian Linux 8.1, the `arp` utility for viewing and manipulating Address Resolution Protocol (ARP) entries is found at (in other words, its *path* is) `/usr/sbin/arp`.

This concept of path becomes important when we start considering that Bash allows you to navigate, or move around, within the file system. The *prompt*, or the text that Bash displays when waiting for you to input a command, will tell you where you are in the file system. Here's the default prompt for a Debian 8.1 system:

```
vagrant@jessie:~$
```

Do you see it? Unless you're familiar with Linux, you may have missed the tilde (`~`) following `vagrant@jessie:` in the figure. In the Bash shell, the tilde is a shortcut that refers to the user's *home directory*. Each user has a home directory that is their personal location for storing files, programs, and other content for only that user. To make it easy to refer to one's home directory, Bash uses the tilde as a shortcut for the home directory. So, looking back at the sample prompt, you can see that this particular prompt tells you a few different things:

1. The first part of the prompt, before the `@` symbol, tells you the current user (in this case, `vagrant`).
2. The second part of the prompt, directly after the `@` symbol, tells you the current hostname of the system on which you are currently operating (in this case, `jessie` is the hostname).
3. Following the colon is the current directory, noted in this case as `~` meaning that this user (`vagrant`) is currently in his or her home directory.
4. Finally, even the `$` at the end has meaning—in this particular case, it means that the current user (`vagrant`) does not have root permissions. (The `$` will change to an octothorpe—also known as a hash sign, `\#`—if the user has root permissions.)

The default prompt on a CentOS 7.1 system looks something like this:

```
[vagrant@centos ~]$
```

About the environments we're using

Throughout this chapter, you'll see various Linux prompts similar to ones we just showed you. We're using a tool called Vagrant to simplify the creation of multiple different Linux environments—in this case, Debian GNU/Linux 8.1 (also known as "Jessie"), Ubuntu Linux 14.04 LTS (named "Trusty Tahr"), and CentOS 7.1. The Vagrant environments we use in this book are available from this book's GitHub repository

As you can see, it's very similar, and it conveys the same information as the other example prompt we showed, albeit in a slightly different format. Like the earlier example, this prompt shows us the current user (`vagrant`), the hostname of the current system (`centos`), the current directory (`~`), and the effective permissions of the logged-in user (`$`).

The use of the tilde is helpful in keeping the prompt short when you're in your home directory, but what if you don't know the path to your home directory? In other words, what if you don't know where on the system your home directory is located? In situations like this where you need to determine the full path to your current location, Bash offers the `pwd` (print working directory) command, which will produce output something like this:

```
vagrant@jessie:~$ <userinput>pwd</userinput>
/home/vagrant
vagrant@jessie:~$
```

The `pwd` command simply returns the directory where you're currently located in the file system (the working directory).

Now that you know where you are located in the file system, you can begin to move around the file system using the `cd` (change directory) command along with a path to a destination. For example, if you were in your home directory and wanted to change into the `bin` subdirectory, you'd simply type **`cd bin`** and press Enter (or Return).

Note the lack of the leading slash here. This is because `/bin` and `bin` might be two *very* different locations in the file system:

- Using `bin` (no leading slash) tells Bash to change into the `bin` subdirectory of the current working directory.
- Using `/bin` (with a leading slash) tells Bash to change into the `bin` subdirectory of the root (`/`) directory.

See how, therefore, `bin` and `/bin` might be very different locations? This is why understanding the concept of a single-root file system and the path to a file or directory is important. Otherwise, you might end up performing some action on a different file or directory than what you intended! This is particularly important when it comes to manipulating files and directories, which we'll discuss in the next section.

Before moving on, though, there are a few more navigational commands we need to discuss.

To move up one level in the file system (for example, to move from `/usr/local/bin/` to `/usr/local/`), you can use the `..` shortcut. Every directory contains a special entry, named `..` (two periods), that is a shortcut entry for that directory's parent directory (the directory one level above it). So, if your current working directory is `/usr/local/bin`, you can simply type **`cd ..`** and press Enter (or Return) to move up one directory.

```
vagrant@jessie:/usr/local/bin$ <userinput>cd ..</userinput>
vagrant@jessie:/usr/local$
```

Note that you can combine the `..` shortcut with a directory name to move laterally between directories. For example, if you're currently in `/usr/local` and need to move to `/usr/share`, you can type **`cd ../share`** and press Enter. This moves you to the directory whose path is up one level (`..`) and is named `share`.

```
vagrant@jessie:/usr/local$ <userinput>cd ../share</userinput>
vagrant@jessie:/usr/share$
```

You can also combine multiple levels of the `..` shortcut to move up more than one level. For example, if you are currently in `/usr/share` and need to move to `/` (the root directory), you could type **`cd ../../`** and press Enter. This would put you into the root (`/`) directory.

```
vagrant@jessie:/usr/share$ <userinput>cd ../..</userinput>
vagrant@jessie:/$
```

All these examples are using *relative paths*, i.e., paths that are relative to your current location. You can, of course, also use *absolute paths*; that is, paths that are anchored to the root directory. As we mentioned earlier, the distinction is the use of the forward slash (`/`) to denote an absolute path starting at the root versus a path relative to the current location. For example, if you are currently located in the root directory (`/`) and need to move to `/media/cdrom`, you don't need the leading slash (because `media` is a subdirectory of `/`). You can use **`cd media/cdrom`** and press Enter. This will move you directory to `/media/cdrom`, because you used a relative path to your destination.

```
vagrant@jessie:/$ <userinput>cd media/cdrom</userinput>
vagrant@jessie:/media/cdrom$
```

From here, though, if you needed to move to `/usr/local/bin`, you'd want to use an absolute path. Why? Because there is no (easy) relative path between these two locations that doesn't involve moving through the root (see the "More than one path" sidebar for a bit more detail.). Using an absolute path, anchored with the leading slash, is the quickest and easiest approach.

```
vagrant@jessie:/media/cdrom$ <userinput>cd /usr/local/bin</userinput>
vagrant@jessie:/usr/local/bin$
```

More than one path

If you're thinking that you could have also used the command **`cd ../../usr/local/bin`** to move from `/media/cdrom` to `/usr/local/bin`, you've mastered the relationship between relative paths and absolute paths on a Linux system.

Finally, there's one final navigation trick we want to share. Suppose you're in `/usr/local/bin`, but you need to switch over to `/media/cdrom`. So you enter **`cd /media/cdrom`**, but after switching directories realize you needed to be in `/usr/local/bin` after all. Fortunately, there is a quick fix. The notation `cd -` tells Bash to switch back to the last directory you were in before you switched to the current directory. (If you need a shortcut to get back to your home directory, just enter `cd` with no parameters.)

```
vagrant@jessie:/usr/local/bin$ <userinput>cd /media/cdrom</userinput>
vagrant@jessie:/media/cdrom$ <userinput>cd -</userinput>
/usr/local/bin
vagrant@jessie:/usr/local/bin$ <userinput>cd -</userinput>
/media/cdrom
vagrant@jessie:/media/cdrom$ <userinput>cd -</userinput>
/usr/local/bin
vagrant@jessie:/usr/local/bin$
```

Here are all of these file system navigation techniques in action.

```
vagrant@jessie:/usr/local/bin$ <userinput>cd ..</userinput>
vagrant@jessie:/usr/local$ <userinput>cd ../share</userinput>
vagrant@jessie:/usr/share$ <userinput>cd ../..</userinput>
vagrant@jessie:/$ <userinput>cd media/cdrom</userinput>
vagrant@jessie:/media/cdrom$ <userinput>cd /usr/local/bin</userinput>
vagrant@jessie:/usr/local/bin$ <userinput>cd -</userinput>
/media/cdrom
vagrant@jessie:/media/cdrom$ <userinput>cd -</userinput>
/usr/local/bin
vagrant@jessie:/usr/local/bin$
```

Now you should have a pretty good grasp on how to navigate around the Linux file system. Let's build on that knowledge with some information on manipulating files and directories.

Manipulating Files and Directories

Armed with a basic understanding of the Linux file system, paths within the file system, and how to move around the file system, let's take a quick look at manipulating files and directories. We'll cover four basic tasks:

- Creating files and directories
- Deleting files and directories
- Moving, copying, and renaming files and directories
- Changing permissions

Let's start with creating files and directories.

Creating Files and Directories

To create files or directories, you'll work with one of two basic commands: `touch`, which is used to create files, and `mkdir` (make directory), which is used—not surprisingly—to create directories.

Other ways exist

There are other ways of creating files, such as echoing command output to a file or using an application (like a text editor, for example). Rather than trying to cover all the possible ways to do something, we want to focus on getting you enough information to get started.

The `touch` command just creates a new file with no contents (it's up to you to use a text editor or appropriate application to add content to the file after it is created). Let's look at a few examples:

```
[vagrant@centos ~]$ <userinput>touch config.txt</userinput>
```

Here's an equivalent command (we'll explain why it's equivalent in just a moment):

```
[vagrant@centos ~]$ <userinput>touch ./config.txt</userinput>
```

Why this command is equivalent to the earlier example may not be immediately obvious. In the previous section, we talked about the `..` shortcut for moving to the parent directory of the current directory. Every directory also has an entry noted by a single period (`.`) that refers to the *current directory*. Therefore, using `touch config.txt` and `touch ./config.txt` will *both* create a file named `config.txt` in the current working directory.

If both syntaxes are correct, why are there two different ways of doing it? In this case, both commands produce the same result—but *this isn't the case for all commands*. When you want to be sure that the file you're referencing is the file in the current working directory, use `./` to tell Bash you want the file in the current directory.

```
[vagrant@centos ~]$ <userinput>touch /config.txt</userinput>
```

In this case, we're using an absolute path, so this command creates a file named `config.txt` in the root directory, assuming your user account has permission. (We'll talk about permissions later in this chapter in the section titled "Changing Permissions".)

When ./ is useful

One thing we haven't discussed in detail yet is the idea of Bash's *search paths*, which are paths (locations) in the file system that Bash will automatically search when you type in a command. In a typical configuration, paths such as `/bin`, `/usr/bin`, `/sbin/`, and similar locations are included in the search path. Thus, if you specify a filename from a file in one of those directories without using the full path, Bash will find it for you by searching these paths. This is one of the times when being specific about a file's location (by including `./` or the absolute path) might be a good idea, so that you can be sure which file is the file being found and used by Bash.

The `mkdir` command is very simple: it creates the directory specified by the user. Let's look at a couple quick examples.

```
[vagrant@centos ~]$ <userinput>mkdir bin</userinput>
```

This command creates a directory named `bin` in the current working directory. It's different than this command (relative versus absolute paths!):

```
[vagrant@centos ~]$ <userinput>mkdir /bin</userinput>
```

Like most other Linux commands, `mkdir` has a lot of options that modify its behavior, but one you'll use frequently is the `-p` parameter. When used with the `-p` option, `mkdir` will not report an error if the directory already exists, and will create parent directories along the path as needed.

For example, let's say you had some files you needed to store, and you wanted to store them in `/opt/sw/network`. If you were in the `/opt` directory and entered **`mkdir sw/network`** when the `sw` directory didn't already exist, the `mkdir` command would report an error. However, if you simply added the `-p` option, `mkdir` would then create the `sw` directory if needed, then create `network` under `sw`. This is a *great* way to create an entire path all at once without failing due to errors if a directory along the way already exists.

Creating files and directories is one half of the picture; let's look at the other half (deleting files and directories).

Deleting Files and Directories

Similar to the way there are two commands for creating files and directories, there are two commands for deleting files and directories. Generally, you'll use the `rm` command to delete (remove) files, and you'll use the `rmdir` command to delete directories. There is also a way to use `rm` to delete directories, as we'll show you in this section.

To remove a file, you simply use `rm` *`filename`*. For example, to remove a file named `config.txt` in the current working directory, you'd use one of the two following commands (do you understand why?):

```
vagrant@trusty:~$ <userinput>rm config.txt</userinput>
vagrant@trusty:~$ <userinput>rm ./config.txt</userinput>
```

You can, of course, use absolute paths (`/home/vagrant/config.txt`) as well as relative paths (`./config.txt`).

To remove a directory, you use `rmdir` *`directory`*. Note, however, that the directory has to be empty; if you attempt to delete a directory that has files in it, you'll get this error message:

```
rmdir: failed to remove 'src': Directory not empty
```

In this case, you'll need to first empty the directory, then use `rmdir`. Alternately, you can use the `-r` parameter to the `rm` command. Normally, if you try to use the `rm` command on a directory and you fail to use the `-r` parameter, Bash will respond like this (in this example, we tried to remove a directory named `bin` in the current working directory):

```
rm: cannot remove 'bin': Is a directory
```

When you use `rm -r` *`directory`*, though, Bash will remove the entire directory tree. Note that, by default, `rm` *isn't* going to prompt for confirmation—it's simply going to delete the whole directory tree. No Recycle Bin, no Trash Can…it's gone. (If you want a prompt, you can add the `-i` parameter.)

Warning

When you delete a file using `rm` *without* the `-i` parameter, there is no Recycle Bin and no Trash Can. The file is gone. The same goes for the `mv` and `cp` commands we'll discuss in the next section—without the `-i` parameter, these commands will simply overwrite files in the destination without any prompt. Be sure to exercise the appropriate level of caution when using these commands.

Creating and deleting files and directories aren't the only tasks you might need to do, though, so let's take a quick look at moving (or copying) files and directories.

Moving, Copying, and Renaming Files and Directories

When it comes to moving, copying, and renaming files and directories, the two commands you'll need to use are `cp` (for copying files or directories) and `mv` (for moving and renaming files and directories).

Check the man pages!

The basic use of all the Linux commands we've shown you so far is relatively easy to understand, but—as the saying goes—the Devil is in the details. If you need more information on any of the options, parameters, or the advanced usage of just about any command in Linux, use the `man` (manual) command. This command will show you a more detailed explanation of how to use the command.

To copy a file, it's just `cp` *`source destination`*. Similarly, to move a file you would just use `mv` *`source destination`*. Renaming a file, by the way, is considering moving it from one name to a new name (typically in the same directory).

Moving a directory is much the same; just use `mv` *`source-dir destination-dir`*. This is true whether the directory is flat (containing only files) or a tree (containing both files as well as subdirectories).

Copying directories is only a bit more complicated. Just add the `-r` option, like `cp -r` *`source-dir destination-dir`*. This will handle most use cases for copying directories, although some less-common use cases may require some additional options. We recommend you read and refer to the man (manual) page for `cp` for additional details.

The final topic we'd like to tackle in our discussion of manipulating files and directories is the topic of permissions.

Changing Permissions

Taking a cue from its UNIX predecessors (keeping in mind that Linux rose out of efforts to create a free UNIX-like operating system), Linux is a multi-user OS that incorporates the use of permissions on files and directories. In order to be considered a multi-user OS, Linux had to have a way to make sure one user couldn't view/see/modify/remove other users' files, and so file- and directory-level permissions were a necessity.

Linux permissions are built around a couple of key ideas:

- Permissions are assigned based on the user (the user who owns the file), group (other users in the file's group), and others (other users not in the file's group)
- Permissions are based on the action (read, write, and execute)

Here's how these two ideas come together. Each of the actions (read, write, and execute) are assigned a value; specifically, read is set to 4, write is set to 2, and execute is set to 1. (Note that these values correspond exactly to binary values.) To allow multiple actions, add the values for each underlying action. For example, if you wanted to allow both read and write, the value would be 6 (read = 4, write = 2, so read-write = 6).

These values are then assigned to user, group, and others. For example, to allow the file's owner to read and write to a file, you'd assign the value

6 to the user's permissions. To allow the file's owner to read, write, and execute a file, you'd assign the value 7 to the user's permissions. Similarly, if you wanted to allow users in the file's group to read the file but not write or execute it, you'd assign the value 2 to the group's permissions. User, group, and other permissions are listed as an octal number, like this:

644 (user = read+write, group = read, others = read)
755 (user = read+write+execute, group = read+execute, others = read+execute)
600 (user = read+write, group = none, others = none)
620 (user = read+write, group = write, others = none)

You may also see these permissions listed as a string of characters, like `rwxr-xr-w`. This breaks down to the read (r), write (w), and execute (x) permissions for each of the three entities (user, group, and others). Here's the same examples as earlier, but written in alternate format:

644 = `rw-r--r--`
755 = `rwxr-xr-w`
600 = `rw-------`
620 = `rw-w----`

The read and write permissions are self-explanatory, but execute is a bit different. For a file, it means just what it says: the ability to execute the file as a program (something we'll discuss in more detail in the next section, "Running Programs"). For a directory, though, it means the ability to look into and list the contents of the directory. Therefore, if you want members of a directory's group to see the contents of that directory, you'll need to grant the execute permission.

A couple of different Linux tools are used to view and modify permissions. The `ls` utility, used for listing the contents of a directory, will show permissions when used with the `-l` option, and is most likely the primary tool you'll use to view permissions. Figure [Link to Come], below, contains the output of `ls -l /bin` on a Debian 8.1 system, and clearly show permissions assigned to the files in the listing.

[[4.3]] .Permissions in a file listing image::images/linux/file-listing.png["Permissions shown in a file listing"]

To change or modify permissions, you'll need to use the `chmod` utility. This is where the explanation of octal values (755, 600, 644, etc.) and the `rwxr-xr-w` notation (typically referred to as *symbolic notation*) come in handy, because that's how `chmod` expects the user to enter permissions. As with relative paths vs. absolute paths, the use of octal values versus symbolic notation is really a matter of what you're trying to accomplish:

- If you need (or are willing to) set all the permissions at the same time, use octal values. Even if you omit some of the digits, you'll still be changing the permissions because `chmod` assumes missing digits are leading zeroes (and thus you're setting permissions to none).
- If you need to set only one part (user, group, or others) of the permissions while leaving the rest intact, use symbolic notation. This will allow to you only modify one part of the permissions (for example, only the user permissions, or only the group permissions).

Here are a few quick examples of using `chmod`. First, let's set the `bin` directory in the current working directory to mode 755 (owner = read/write/execute, all others = read/execute):

```
[vagrant@centos ~]$ <userinput>chmod 755 bin</userinput>
```

Next, let's use symbolic notation to add read/write permissions to the user that owns the file `config.txt` in the current working directory, while leaving all other permissions intact:

```
[vagrant@centos ~]$ <userinput>chmod u+rw config.txt</userinput>
```

Here's an even more complex example—this adds read/write permissions for the file owner, but removes write permission for the file group:

```
[vagrant@centos ~]$ <userinput>chmod u+rw,g-w /opt/share/config.txt</userinput>
```

The `chmod` command also supports the use of the `-R` option to act *recursively*, meaning the permission changes will be propagated to files and subdirectories (obviously this works only when using `chmod` against a directory).

Modifying ownership and file group

Given that file ownership and file group play an integral role in file permissions, it's natural that Linux also provides tools to modify file ownership and file group (the `ls` command is used to view ownership and group, as shown earlier in Figure [Link to Come]). You'll use the `chown` command to change ownership, and the `chgrp` command to change the file group. Both commands support the same `-R` option as `chmod` to act recursively.

We're now ready to move on from file and directory manipulation to our next major topic in interacting with Linux, which is running programs.

Running Programs

Running program is actually pretty simple, given the material we've already covered. In order to run a program, here's what's needed:

1. A file that is actually an executable file (you can use the `file` utility to help determine if a file is executable)
2. Execute permissions (either as the file owner, as a member of the file's group, or with the execute permission given to others)

We discussed the second requirement (execute permissions) in the previous section on permissions, so we don't need to cover that again here. If you don't have execute permissions on the file, use the `chmod`, `chown`, and/or `chgrp` commands as needed to address it. The first requirement (an executable file) deserves a bit more discussion, though.

What makes up an "executable file"? It could be a binary file, compiled from a programming language such C or C+. However, it could also be an executable text file, such as a Bash shell script (a series of Bash shell commands) or a script written in a language like Python or Ruby. (We'll be talking about Python extensively in the next chapter.) The `file` utility (which may or may not be installed by default; use your Linux distribution's package management tool to install it if it isn't already installed) can help here.

Here's the output of the `file` command against various types of executable files.

```
vagrant@jessie:~$ <userinput>file /bin/bash</userinput>
/bin/bash: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=a8ff57737fe60fba639d91d603253f4cdc6eb9f7, stripped
vagrant@jessie:~$ <userinput>file docker</userinput>
docker: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, for GNU/Linux 2.6.24, BuildID[sha1]=3d4e8c5339180d462a7f43e62ede4f231d625f71, not stripped
vagrant@jessie:~$ <userinput>file shellscript.sh</userinput>
script.sh: Bourne-Again shell script, ASCII text executable
vagrant@jessie:~$ <userinput>file testscript.py</userinput>
script.py: Python script, ASCII text executable
vagrant@jessie:~$ <userinput>file testscript-2.rb</userinput>
script.rb: Ruby script, ASCII text executable
```

Scripts and the Shebang

You'll note that the `file` command can identity text files as a Python script, a Ruby script, or a shell (Bash) script. This might sound like magic, but in reality it's relying upon a Linux construct known as the *shebang*. The *shebang* is the first line in a text-based script and it starts with the characters `#!`, followed by the path to the interpreter to the script (the *interpreter* is what will execute the commands in the script). For example, on a Debian 8.1 system the Python interpreter is found at `/usr/bin/python`, and so the shebang for a Python script would look like `\#!/usr/bin/python`. A Ruby script would have a similar shebang, but pointing to the Ruby interpreter. A Bash shell script's shebang would point to Bash itself, of course.

Once you've satisfied both requirements—you have an executable file and you have execute permissions on the executable file—running a program is as simple as entering the program name on the command line. *That's it.* Each program may, of course, have certain options and parameters that need to be supplied. The only real "gotcha" here might be around the use of absolute paths; for example, if multiple programs named "testnet" exist on your Linux system and you simply enter `testnet` at the shell prompt, which one will it run? This is where an understanding of Bash search paths (which we covered earlier) and/or the use of absolute paths can help you ensure that you're running the intended program.

Let's expand on this potential "gotcha" just a bit. Earlier in this chapter, in the "Navigating the File System" section, we covered the idea of relative paths and absolute paths. We're going to add to the discussion of paths now by introducing the concept of a *search path*. Every Linux system has a search path, which is a list of directories on the system that it will search when the user enters a filename. You can see the current search path by entering `echo $PATH` at your shell prompt, and on a CentOS 7 system you'd see something like this:

```
[vagrant@centos ~]$ <userinput>echo $PATH</userinput>
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/vagrant/.local/bin:/home/vagrant/bin
[vagrant@centos ~]$
```

What this means is that if you had a script named `testscript.py` stored in `/usr/local/bin`, you could be in *any* directory on the system and simply enter the script's name (`testscript.py`) to execute the script. The system would search the directories in the search path (in order) for the filename you'd entered, and execute the first one it found (which, in this case, would typically be the one in +/usr/local/bin because that's the first directory in the search path).

You'll note, by the way, that the search path does *not* include the current directory. Let's say you've created a `scripts` directory in your home directory, and in that directory you have a shell script you've written called `shellscript.sh`. Take a look at the behavior from the following set of commands:

```
[vagrant@centos ~]$ <userinput>pwd</userinput>
/home/vagrant/scripts
[vagrant@centos ~]$ <userinput>ls</userinput>
```

```
shellscript.sh
[vagrant@centos ~]$ <userinput>shellscript.sh</userinput>
-bash: /home/vagrant/bin/shellscript.sh: No such file or directory
[vagrant@centos ~]$ <userinput>./shellscript.sh</userinput>
This is a shell script.
[vagrant@centos ~]$
```

Because the shell script wasn't in the search path, we had to use an absolute path—in this case, the absolute path was telling Bash (via the `./` notation) to look in the current directory.

Therefore, the "gotcha" with running programs is that any program you run—be it a compiled binary or an ASCII text script that will be interpreted by Bash, Python, Ruby, or some other interpreter—needs to be in the search path, or you'll have to explicitly specify the absolute path (which may include the current directory) to the program. In the case of multiple programs with the same name in different directories, it also means that the program Bash finds first will be the program that gets executed, and the search order is determined by the search path.

Note

You can, of course, change and customize the search path. The search path is controlled by what is known as an *environment variable* whose name is PATH. (By convention, all environment variables are specified in uppercase letters.) Modifying this environment variable will modify the search order that Bash uses to locate programs.

There's one more topic we're going to cover before moving on to a discussion of networking in Linux, and that's working with background programs, also known as daemons.

Working with Daemons

In the Linux world, we use the term *daemon* to refer to a process that runs in the background. (You may also see the term *service* used to describe these types of background processes.) Daemons are most often encountered when you're using Linux to provide network-based functionality. Examples—some of which we discussed earlier when we first introduced the section on interacting with Linux—might include a DHCP server, an HTTP server, a DNS server, or an FTP server. On a Linux system, each of these network services is provided by a corresponding daemon (or service). In this section, we're going to talk about how to work with daemons: start daemons, stop daemons, restart a daemon, or check on a daemon's status.

It used to be that working with daemons on a Linux system varied pretty widely between distributions. Startup scripts, referred to as *init scripts*, were used to start, stop, or restart a daemon. Some distributions offered utilities—often nothing more than Bash shell scripts—such as `service` command to help simplify working with daemons. For example, on Ubuntu 14.04 LTS and CentOS 7.1 systems, the `service` command (found in `/usr/sbin/`) allowed you to start, stop, or restart a daemon. Behind the scenes, these utilities are calling distribution-specific commands (such as `initctl` on Ubuntu or `systemctl` on CentOS) to actually perform their actions.

In recent years, though, the major Linux distributions have converged on the use of systemd as their init system: RHEL/CentOS 7.x uses systemd, Debian 8.0 and later use systemd, and Ubuntu 15.04 and later use systemd. As such, working with daemons (background services) should become easier in the future, although there are (and probably will continue to be) slight differences in each distribution's implementation and use of systemd.

Note

If you are interested in more details on systemd, we recommend having a look at the systemd website.

In the meantime, though, let's look at working with daemons across the three major Linux distributions we've selected for use in this chapter: Debian GNU/Linux 8.1 ("Jessie"), Ubuntu "Trusty Tahr" 14.04 LTS, and CentOS 7.1. We'll start with Debian GNU/Linux 8.1.

Working with Background Services in Debian GNU/Linux 8.1

Starting with version 8.0, Debian GNU/Linux uses systemd as its init system, and therefore the primary means by which you'll work with background services is via the `systemctl` utility (found on the system as `/bin/systemctl`). Unlike some other distributions, Debian does not offer any sort of "wrapper" commands that in turn call `systemctl` on the back end, instead preferring to have users use `systemctl` directly.

There's much more to systemd

There's a great deal more to systemd, which Debian GNU/Linux 8.x uses as its init system, than we have room to discuss here. When we provide examples on how to start, stop, or restart a background service using systemd, we assume that the systemd unit file has already been installed and enabled, and that it is recognized by systemd.

To start a daemon using systemd, you'd call `systemctl` with the `start` command:

```
vagrant@jessie:~$ <userinput>systemctl start <replaceable>service name</replaceable></userinput>
```

To stop a daemon using systemd, replace the `start` parameter with `stop`, like this:

```
vagrant@jessie:~$ <userinput>systemctl stop <replaceable>service name</replaceable></userinput>
```

Similarly, use the `restart` command to stop and then start a daemon:

```
vagrant@jessie:~$ <userinput>systemctl restart <replaceable>service name</replaceable></userinput>
```

And use the `restart` parameter to `systemctl` to check the current status of a daemon. Figure [Link to Come], below, shows the output of running `systemctl status` on a Debian 8.1 virtual machine.

[[4.4]] .Output of a systemctl status command image::images/linux/systemctl-output-debian.png["Output from systemctl status on a Debian GNU/Linux system"]

What if you don't know the service name? `systemctl list-units` will give you a paged list of all the loaded and active units.

Note

Prior to version 8.0, Debian did not use systemd. Instead, Debian used an older init system known as *System V init* (or *sysv-rc*).

Now let's shift and take a look at working with daemons on Ubuntu Linux 14.04 LTS. Although Ubuntu Linux is a Debian derivative, you'll see that there are significant differences between Debian 8.x and this latest LTS release from Ubuntu.

Working with Background Services in Ubuntu Linux 14.04 LTS

Unlike Debian 8.x and CentOS 7.x, Ubuntu 14.04 LTS (recall that the LTS denotes a long-term support release that is supported for 5 years after release) does *not* use systemd as its init system. Instead, Ubuntu 14.04 uses a system developed by Canonical and called Upstart.

The primary command you'll use to interact with Upstart for the purpose of stopping, starting, restarting, or checking the status of background services (also referred to as "jobs" in the Upstart parlance) is `initctl`, and it is used in a fashion very similar to `systemctl`.

For example, to start a daemon you'd use `initctl` like this:

```
vagrant@trusty:~$ <userinput>initctl start <replaceable>service name</replaceable></userinput>
<replaceable>service name</replaceable> start/running
```

Likewise, to stop a daemon you'd replace "start" in the previous command with "stop", like this:

```
vagrant@trusty:~$ <userinput>initctl stop <replaceable>service name</replaceable></userinput>
<replaceable>service name</replaceable> stop/waiting
```

The `restart` and `status` parameters to `initctl` work in much the same way:

```
vagrant@trusty:~$ <userinput>initctl restart vmware-tools</userinput>
vmware-tools start/running
vagrant@trusty:~$ <userinput>initctl status vmware-tools</userinput>
vmware-tools start/running
```

And, like with `systemctl`, there is a way to get the list of service names, so that you know the name to supply when trying to start, stop, or check the status of a daemon:

```
vagrant@trusty:~$ <userinput>initctl list</userinput>
```

Ubuntu 14.04 LTS also comes with some "shortcuts" to working with daemons:

- There are commands named `start`, `stop`, `restart`, and `status` that are symbolic links to `initctl`. Each of these commands works as if you had typed `initctl command`, so using `stop vmware-tools` would be the same as `initctl stop vmware-tools`. These symbolic links are found in the `/sbin` directory.
- Ubuntu also has a shell script, named `service`, that calls `initctl` on the back end. The format for the `service` command is `service service action`, where `service` is the name of the daemon (which you can obtain via `initctl list`) and action is one of start, stop, restart, or status. Note that this syntax is opposite of `initctl` itself, which is `initctl action service`, which may cause some confusion if you switch

back and forth between using the `service` script and `initctl`.

Note

You may have noticed us mentioning something called a *symbolic link* in our discussion of managing daemons on Ubuntu 14.04. Symbolic links are pointers to a file that allow the file to be referenced multiple times (using different names in different directories) even though the file exists only once on the disk. Symbolic links are not unique to Ubuntu, but are common to all the Linux distributions we discuss in this book.

Next, we'll look at working with background services in CentOS 7.1.

Working with Background Services in CentOS 7.1

CentOS 7.1 uses systemd as its init system, so it is largely similar to working with daemons on Debian GNU/Linux 8.x. In fact, the core `systemctl` commands are completely unchanged, although you will note differences in the unit names when running `systemctl list-units` on the two Linux distributions. Make note of these differences when using both CentOS 7.x and Debian 8.x in your environment.

One difference between Debian and CentOS is that CentOS includes a wrapper script named `service` that allows you to start, stop, restart, and check the status of daemons. It's likely that this wrapper script (we call it a "wrapper script" because it acts as a "wrapper" around `systemctl`, which does the real work on the back end) was included for backward compatibility, as previous releases of CentOS did *not* use systemd and also featured this same command. Note that although it shares a name with the `service` command from Ubuntu, the two scripts are *not* the same and are not portable between the distributions.

The syntax for the `service` command is `service service action`. Like on Ubuntu, where the syntax of the `service` script is opposite of `initctl`, you'll note that the `service` script on CentOS also uses a syntax that is opposite of `systemctl` (which is `systemctl action service`).

Before we wrap up this section on working with daemons and move into a discussion of Linux networking, there are a few final commands we think you might find helpful.

Other Daemon-Related Commands

We'll close out this section on working with daemons with a quick look at a few other commands that you might find helpful. For full details on all the various parameters for these commands, we encourage you to read the man pages (use `man command` at a Bash prompt).

- To show network connections to a daemon, you can use the `ss` command. One particularly helpful use of this command is to show listening network sockets, which is one way to ensure that the networking configuration for a particular daemon (background service) is working properly. Use `ss -lnt` to show listening TCP sockets, and use `ss -lnu` to show listening UDP sockets.
- The `ps` command is useful for presenting information on the currently running processes.

Before we move on to the next section, let's take a quick moment and review what we've covered so far:

- We provided some background and history for Linux.
- We've supplied information on basic file system navigation and paths.
- We've shown you how to perform basic file manipulations (create files and directories, move/copy files and directories, remove files and directories).
- We've discussed how to work with background services, also known as daemons.

Our next major topic is networking in Linux, which will build on many of the areas we've already touched on so far in this chapter.

Networking in Linux

We stated earlier in this chapter that our coverage of Linux was intended to get you "up and running" with Linux in the context of network automation and programmability. Naturally, this means that our discussion of Linux would not be complete without also discussing networking in Linux. This is, after all, a networking-centric book!

Working with Interfaces

The basic building block of Linux networking is the *interface*. Linux supports a number of different types of interfaces; the most common of these are physical interfaces, VLAN interfaces, veth pairs, and bridge interfaces. As with most other things in Linux, you configure these various types of interfaces through command-line utilities executed from the Bash shell and/or using certain plain-text configuration files. Making interface configuration changes persistent across a reboot typically requires modifying a configuration file. Let's look first at using the command-line utilities, then we'll discuss persistent changes using interface configuration files.

Interface Configuration via the Command Line

Just as the Linux distributions have converged on systemd as the primary init system, most of the major Linux distributions have converged on a single set of command-line utilities for working with network interfaces. These commands are part of the "iproute2" set of utilities, available in the major Linux distributions as either "iproute" or "iproute2" (CentOS 7.1 uses "iproute"; Debian 8.1 and Ubuntu 14.04 use "iproute2" for the package name). This set of utilities uses a command called `ip` to replace the functionality of earlier (and now deprecated) commands such as `ifconfig` and `route` (both Ubuntu 14.04 and CentOS 7.1 include these earlier commands, but Debian 8.1 does not).

More information on iproute2

If you're interested in more information on iproute2, visit the iproute2 working group's page.

For interface configuration, two forms of the `ip` command will be used: `ip link`, which is used to view or set interface link status, and `ip addr`, which is used to view or set IP addressing configuration on interfaces. (We'll look at some other forms of the `ip` command later in this section.)

Let's look at a few task-oriented examples of using the `ip` commands to perform interface configuration.

Listing Interfaces

You can use either the `ip link` or `ip addr` command to list all the interfaces on a system, although the output will be slightly different for each command.

If you want a listing of the interfaces along with the interface status, use the `ip link list` command, like this:

```
[vagrant@centos ~]$ <userinput>ip link list</userinput>
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ens32: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT qlen 1000
    link/ether 00:0c:29:d7:28:17 brd ff:ff:ff:ff:ff:ff
3: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT qlen 1000
    link/ether 00:0c:29:d7:28:21 brd ff:ff:ff:ff:ff:ff
[vagrant@centos ~]$
```

Note

The default "action", so to speak, for most (if not all) of the `ip` commands is to list the items with which you're working. Thus, if you want to list all the interfaces, you can just use `ip link` instead of `ip link list`, or if you wanted to list all the routes you can just use `ip route` instead of `ip route list`. We will specify the full commands here for clarity.

As you can tell from the prompt, this output was taken from a CentOS 7.1 system. The command syntax is the same across the three major distributions we're discussing in this chapter, and the output is largely identical.

You'll note that this output shows you the current list of interfaces (note that CentOS assigns different names to the interfaces than Debian and Ubuntu), the current Maximum Transmission Unit (MTU), the current administrative state (UP), and the Ethernet Media Access Control (MAC) address, among other things.

The output of this command also tells you the current state of the interface (note the information in brackets immediately following the interface name):

- UP: Indicates that the interface is enabled.
- LOWER_UP: Indicates that interface link is up.
- NO_CARRIER (not shown above): The interface is enabled, but there is no link.

If you're accustomed to working with network equipment, you're probably familiar with an interface being "down" versus being "administratively down". If an interface is down because there is no link, you'll see "NO_CARRIER" in the interface state. If the interface is administratively down, then you won't see "UP", "LOWER_UP", or "NO_CARRIER", and state will be listed as "DOWN". In the next section ("Enabling/Disabling Interfaces") we'll show you how to use the `ip link` command to disable an interface (set an interface administratively down).

You can also list interfaces using the `ip addr list` command, like this (this output is taken from Ubuntu 14.04 LTS):

```
vagrant@trusty:~$ <userinput>ip addr list</userinput>
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:33:99:f6 brd ff:ff:ff:ff:ff:ff
    inet 192.168.70.205/24 brd 192.168.70.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe33:99f6/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:33:99:00 brd ff:ff:ff:ff:ff:ff
    inet 192.168.100.11/24 brd 192.168.100.255 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe33:9900/64 scope link
       valid_lft forever preferred_lft forever
vagrant@trusty:~$
```

As you can see, the `ip addr list` command also lists the interfaces on the system, along with some link status information and the IPv4/IPv6 addresses assigned to the interface.

For both the `ip link list` and `ip addr list` commands, you can filter the list to only a specific interface by adding the interface name. The final command then becomes `ip link list` *interface* or `ip addr list` *interface*, like this:

```
vagrant@jessie:~$ <userinput>ip link list eth0</userinput>
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT group default qlen 1000
    link/ether 00:0c:29:bf:af:1a brd ff:ff:ff:ff:ff:ff
vagrant@jessie:~$
```

Listing interfaces is very useful, of course, but perhaps even more useful is actually modifying the configuration of an interface. In the next section, we'll show you how to enable or disable an interface.

Enabling/Disabling an Interface

In addition to listing interfaces, you also use the `ip link` command to manage an interface's status. To disable an interface, for example, you set the interface's status to "down" using the `ip link set` command:

```
[vagrant@centos ~]$ <userinput>ip link set ens33 down</userinput>
[vagrant@centos ~]$ <userinput>ip link list ens33</userinput>
3: ens33: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast state DOWN mode DEFAULT qlen 1000
    link/ether 00:0c:29:d7:28:21 brd ff:ff:ff:ff:ff:ff
[vagrant@centos ~]$
```

Note "state DOWN" and the lack of "NO_CARRIER", which tells you the interface is administratively down (disabled) and not just down due to a link failure.

To enable (or re-enable) the `ens33` interface, you'd simply use `ip link set` again, this time setting the status to "up":

```
[vagrant@centos ~]$ <userinput>ip link set ens33 up</userinput>
[vagrant@centos ~]$ <userinput>ip link list ens33</userinput>
3: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT qlen 1000
    link/ether 00:0c:29:d7:28:21 brd ff:ff:ff:ff:ff:ff
[vagrant@centos ~]$
```

Setting the MTU of an Interface

If you need to set the MTU of an interface, you'd once again turn to the `ip link` command, using the `set` subcommand like this:

```
[vagrant@centos ~]$ <userinput>ip link set mtu <replaceable>new MTU</replaceable> <replaceable>interface name</replaceable></userinput>
[vagrant@centos ~]$
```

As a specific example, let's say you wanted to run jumbo frames on the ens33 interface on your CentOS 7.x Linux system. Here's the command:

```
[vagrant@centos ~]$ <userinput>ip link set mtu 9000 ens33</userinput>
[vagrant@centos ~]$
```

As with all the other `ip` commands we've looked it, this change is immediate but not persistent—you'll have to edit the interface's configuration file to make the change persistent. We discuss configuring interfaces via configuration files in the next section, titled "Interface Configuration via Configuration Files."

Assigning an IP Address to an Interface

To assign (or remove) an IP address to an interface, you'll use the `ip addr` command. We've already shown you how to use `ip addr list` to see a list of the interfaces and their assigned IP address(es); now we'll expand the use of `ip addr` to add and remove IP addresses.

To assign (add) an IP address to an interface, you'll use the command `ip addr add` *address* `dev` *interface*. For example, if you want to assign (add) the address 172.31.254.100/24 to the eth1 interface on a Debian 8.1 system, you'd run this command:

```
vagrant@jessie:~$ <userinput>ip addr add 172.31.254.100/24 dev eth1</userinput>
vagrant@jessie:~$
```

If an interface already has an IP address assigned, the `ip addr add` command simply *adds* the new address, leaving the original address intact. So, in this example, if the eth1 interface already had an address of 192.168.100.10/24, running the command above would result in this configuration:

```
vagrant@jessie:~$ <userinput>ip addr list eth1</userinput>
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:bf:af:24 brd ff:ff:ff:ff:ff:ff
    inet 192.168.100.10/24 brd 192.168.100.255 scope global eth1
       valid_lft forever preferred_lft forever
    inet 172.31.254.100/24 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:febf:af24/64 scope link
       valid_lft forever preferred_lft forever
vagrant@jessie:~$
```

To remove an IP address from an interface, you'd use `ip addr del` *address* `dev` *interface*. Here we are removing the 172.31.254.100/24 address we assigned earlier to the eth1 interface:

```
vagrant@jessie:~$ <userinput>ip addr del 172.31.254.100/24 dev eth1</userinput>
vagrant@jessie:~$ <userinput>ip addr list eth1</userinput>
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:bf:af:24 brd ff:ff:ff:ff:ff:ff
    inet 192.168.100.10/24 brd 192.168.100.255 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:febf:af24/64 scope link
       valid_lft forever preferred_lft forever
vagrant@jessie:~$
```

As with the `ip link` command, the syntax for the `ip addr add` and `ip addr del` commands is the same across the three major Linux distributions we're discussing in this chapter. The output is also largely identical, although there may be variations in interface names.

So far, we've only shown you how to use the `ip` commands to modify the configuration of an interface. If you're familiar with configuring network devices (and if you're reading this book, you probably are), this could be considered analogous to modifying the running configuration of a network device. However, what we haven't done so far is make these configuration changes permanent. In other words, we haven't changed the startup configuration. To do that, we'll need to look at how Linux uses interface configuration files.

Interface Configuration via Configuration Files

To make changes to an interface persistent across system restarts, using the `ip` commands alone isn't enough. You'll need to edit the interface configuration files that Linux uses on startup so perform those same configurations for you automatically. Unfortunately, while the `ip` commands are pretty consistent across Linux distributions, interface configuration files across different Linux distributions can be quite different.

For example, on RHEL/CentOS/Fedora and derivatives, interface configuration files are found in separate files located in `/etc/sysconfig/network-scripts`. The interface configuration files are named `ifcfg-`*<interface>*, where the name of the interface (like eth0, ens32, or whatever) is embedded in the name of the file. An interface configuration file might look something like this (this example is taken from CentOS 7.1):

```
NAME="ens33"
DEVICE="ens33"
ONBOOT=yes
NETBOOT=yes
IPV6INIT=yes
```

```
BOOTPROTO=dhcp
TYPE=Ethernet
```

Some of the most commonly-used directives in RHEL/CentOS/Fedora interface configuration files are:

- **NAME:** A friendly name for users to see, typically only used in graphical user interfaces (this name wouldn't show up in the output of `ip` commands).
- **DEVICE:** This is the name of the physical device being configured.
- **IPADDR:** The IP address to be assigned to this interface (if not using DHCP or BootP).
- **PREFIX:** If you're statically assigning the IP address, this setting specifies the network prefix to be used with the assigned IP address. (You can alternately use NETMASK instead, but the use of PREFIX is recommended.)
- **BOOTPROTO:** This directive specifies how the interface will have its IP address assigned. A value of "dhcp", as shown earlier, means the address will be provided via Dynamic Host Configuration Protocol (DHCP). The other value typically used here would be "none", which means the address is statically defined in the interface configuration file.
- **ONBOOT:** Setting this directive to "yes" will activate the interface at boot time; setting it to "no" means the interface will not be activated at boot time.
- **MTU:** Specifies the default MTU for this interface.
- **GATEWAY:** This setting specifies the gateway to be used for this interface.

There are many more settings, but these are the ones you're likely to see most often. For full details, check the contents of `/usr/share/doc/initscripts-*<version>*/sysconfig.txt` on your CentOS system.

For Debian and Debian derivatives like Ubuntu, on the other hand, interface configuration is handled by the file `/etc/network/interfaces`. Here's an example network interface configuration file from Ubuntu 14.04 LTS (we're using the `cat` command here to simply output the contents of a file to the screen):

```
vagrant@trusty:~$ <userinput>cat /etc/network/interfaces</userinput>
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eth0
iface eth0 inet dhcp

auto eth1
iface eth1 inet static
      address 192.168.100.11
      netmask 255.255.255.0
vagrant@trusty:~$
```

You'll note that Debian and Ubuntu use a single file to configure all the network interfaces; each interface is separated by a configuration stanza starting with `auto` *`interface`*. In each configuration stanza, the most common configuration options are (you can view all the options for configuring interfaces on a Debian or Ubuntu system by running `man 5 interfaces`):

- Setting the address configuration method: You'll typically use either `inet dhcp` or `inet static` to assign IP addresses to interfaces. In the example shown earlier, the eth0 interface was set to use DHCP while eth1 was assigned statically.
- The `netmask` option provides the network mask for the assigned IP address (when the address is being assigned statically via `inet static`). However, you can also use the prefix format (like "192.168.100.10/24") when assigning the IP address, which makes the use of the `netmask` directive unnecessary.
- The `gateway` directive in the configuration stanza assigns a default gateway when the IP address is being assigned statically (via `inet static`).

If you prefer using separate files for interface configuration, it's also possible to break out interface configuration into per-interface configuration files, similar to how RHEL/CentOS handle it, by including a line like this in the `/etc/network/interfaces` file:

```
source /etc/network/interfaces.d/*
```

This line instructs Linux to look in the `/etc/network/interfaces.d/` directory for per-interface configuration files, and process them as if they were directly incorporated into the main network configuration file. The `/etc/network/interfaces` file on Debian 8.1 includes this line by default (but the directory is empty, and the interface configuration takes place in the `/etc/network/interfaces` file). In the case of using per-interface configuration files, then it's possible that this might be the *only* line found in the `/etc/network/interfaces` file.

A Use Case for Per-Interface Configuration Files

Per-interface configuration files may give you some additional flexibility when using a configuration management tool such as Chef, Puppet, Ansible, or Salt. (We'll discuss these tools in more detail in Chapter 9, [Chapter to Come].) In such situations, it may be easier to use one of these configuration management tools to generate per-interface configuration files instead of having to manage different sections within a single file.

When you make a change to a network interface file, the configuration changes are *not* immediately applied. (If you want an immediate change, use the `ip` commands we described earlier in addition to making changes to the configuration files.) To put the changes into effect, you'll need to "restart" the network interface.

On Ubuntu 14.04, you'd use the `initctl` command, described earlier in the section titled "Working with Daemons," to restart the network interface:

```
vagrant@trusty:~$ <userinput>initctl restart network-interface INTERFACE=<replaceable>interface</replaceable></userinput>
```

On CentOS 7.1, you'd use the `systemctl` command:

```
[vagrant@centos ~]$ <userinput>systemctl restart network</userinput>
```

And on Debian 8.1, you'd use a very similar command:

```
vagrant@jessie:~$ <userinput>systemctl restart networking</userinput>
```

You'll note the systemd-based distributions (CentOS and Debian 8.x) lack a way to do per-interface restarts.

Once the interface is restarted, then the configuration changes are applied and in effect (and you can verify this through the use of the appropriate `ip` commands).

In addition to configuring and managing interfaces, another important aspect of Linux networking is configuring and managing the Linux host's IP routing tables. The next section provides more details on what's involved.

Routing as an End Host

In addition to configuring network interfaces on a Linux host, we also want to show you how to view and manage routing on a Linux system. Interface and routing configuration go hand-in-hand, naturally, but there are times when some tasks for IP routing need to be configured separately from interface configuration. First, though, let's look at how interface configurations affect host routing configuration.

Although the `ip route` command is your primary means of viewing and/or modifying the routing table for a Linux host, the `ip link` and `ip addr` commands may also affect the host's routing table.

First, if you wanted to view the current routing table, you could simply run `ip route list`:

```
vagrant@trusty:~$ <userinput>ip route list</userinput>
default via 192.168.70.2 dev eth0
192.168.70.0/24 dev eth0  proto kernel  scope link  src 192.168.70.205
192.168.100.0/24 dev eth1  proto kernel  scope link  src 192.168.100.11
vagrant@trusty:~$
```

The output of this command tells use a few things:

- The default gateway is 192.168.70.2. The `eth0` device will be used to communicate with all unknown networks via the default gateway. (Recall from the previous section that this would be set via DHCP or via a configuration directive such as `GATEWAY` on a RHEL/CentOS/Fedora system or `gateway` on a Debian/Ubuntu system).
- The IP address assigned to eth0 is 192.168.70.205, and this is the interface that will be used to communicate with the 192.168.70.0/24 network.
- The IP address assigned to eth1 is 192.168.100.11/24, and this is the interface that will be used to communicate with the 192.168.100.0/24

network.

If we disable the eth1 interface using `ip link set eth1 down`, then the host's routing table changes automatically:

```
vagrant@trusty:~$ <userinput>ip link set eth1 down</userinput>
vagrant@trusty:~$ <userinput>ip route list</userinput>
default via 192.168.70.2 dev eth0
192.168.70.0/24 dev eth0  proto kernel  scope link  src 192.168.70.205
vagrant@trusty:~$
```

Now that eth1 is down, the system no longer has a route to the 192.168.100.0/24 network, and the routing table updates automatically. This is all fully expected, but we wanted to show you this interaction so you could see how the `ip link` and `ip addr` commands affect the host's routing table.

For less automatic changes to the routing table, you'll use the `ip route` command. What do we mean by "less automatic changes"? Here are a few use cases:

- Adding a static route to a network over a particular interface
- Removing a static route to a network
- Changing the default gateway

Here are some concrete examples of these use cases.

Let's assume the same configuration we've been showing off so far—the eth0 interface has an IPv4 address from the 192.168.70.0/24 network, and the eth1 interface has an IPv4 address from the 192.168.100.0/24 network. In this configuration, the output of `ip route list` would look like this:

```
vagrant@trusty:~$ <userinput>ip route list</userinput>
default via 192.168.70.2 dev eth0
192.168.70.0/24 dev eth0  proto kernel  scope link  src 192.168.70.205
192.168.100.0/24 dev eth1  proto kernel  scope link  src 192.168.100.11
vagrant@trusty:~$
```

If we were going to model this configuration as a network diagram, it would look something like what's shown in Figure [Link to Come] below:

[[4.4]] .Sample network topology image::images/linux/original-topology.png["Sample network topology"]

Now let's say that a new router is added to the 192.168.100.0/24 network, and a network with which this host needs to communicate (using the subnet address 192.168.101.0/24) is placed beyond that router. Figure [Link to Come] shows the new network topology:

[[4.5]] .Updated network topology image::images/linux/updated-topology.png["Updated network topology after adding new network"]

The host's existing routing table won't allow it to communicate with this new network—since it doesn't have a route to the new network, Linux will direct traffic to the default gateway, which doesn't have a connection to the new network. To fix this, we add a route to the new network over the host's eth1 interface like this:

```
vagrant@jessie:~$ <userinput>ip route add 192.168.101.0/24 via 192.168.100.2 dev eth1</userinput>
vagrant@jessie:~$ <userinput>ip route list</userinput>
default via 192.168.70.2 dev eth0
192.168.70.0/24 dev eth0  proto kernel  scope link  src 192.168.70.204
192.168.100.0/24 dev eth1  proto kernel  scope link  src 192.168.100.10
192.168.101.0/24 via 192.168.100.2 dev eth1
vagrant@jessie:~$
```

This command tells the Linux host (a Debian system, in this example) that it can communicate with the 192.168.101.0/24 network via the IP address 192.168.100.2 over the eth1 interface. Now the host has a route to the new network via the appropriate router, and is able to communicate with systems on that network. If the network topology were updated again with another router and another new network, as shown in Figure [Link to Come] below, we'd need to add yet another route.

[[4.6]] .Final network topology image::images/linux/final-topology.png["Final network topology with multiple networks"]

To address this final topology, you'd run this command:

```
vagrant@jessie:~$ <userinput>ip route add 192.168.102.0/24 via 192.168.100.3 dev eth1</userinput>
vagrant@jessie:~$ <userinput>ip route list</userinput>
default via 192.168.70.2 dev eth0
192.168.70.0/24 dev eth0  proto kernel  scope link  src 192.168.70.204
192.168.100.0/24 dev eth1  proto kernel  scope link  src 192.168.100.10
192.168.101.0/24 via 192.168.100.2 dev eth1
192.168.102.0/24 via 192.168.100.3 dev eth1
vagrant@jessie:~$
```

To make these routes persistent (remember that using the `ip` commands *don't* typically make configuration changes persistent), you'd add these commands to the configuration stanza in `/etc/network/interfaces` for the eth1 device, like this:

```
auto eth1
iface eth1 inet static
      address 192.168.100.11
      netmask 255.255.255.0
      up ip route add 192.168.101.0/24 via 192.168.100.2 dev $IFACE
      up ip route add 192.168.102.0/24 via 192.168.100.3 dev $IFACE
```

The `$IFACE` listed on the commands in this configuration stanza refer to the specific interface being configured, and the `up` directive instructs Debian/Ubuntu systems to run these commands after the interface comes up. With these lines in place, the routes will automatically be added to the routing table every time the system is started.

If, for whatever reason, you need to *remove* routes from a routing table, then you can use the `ip route` command for that as well:

```
[vagrant@centos ~]$ <userinput>ip route del 192.168.103.0/24 via 192.168.100.3</userinput>
```

Finally, changing the default gateway is also something you might need to do using the `ip route` command. (We will note, however, that changing the default gateway can also be accomplished—and made persistent—by editing the interface configuration files. Using the `ip route` command will change it immediately, but the change will not be persistent.) To change the default gateway, you'd use a command somewhat like this (this assumes a default gateway is already present):

```
vagrant@trusty:~$ <userinput>ip route del default via 192.168.70.2 dev eth0</userinput>
vagrant@trusty:~$ <userinput>ip route add default via 192.168.70.1 dev eth0</userinput>
```

Linux also supports what is known as *policy routing*, which is the ability to support multiple routing tables along with rules that instruct Linux to use a specific routing table. For example, perhaps you'd like to use a different default gateway for each interface in the system. Using policy routing, you could configure Linux to use one routing table (and thus one particular gateway) for eth0, but use a different routing table (and a different default gateway) for eth1. Policy routing is a bit of an advanced topic so we won't cover it here, but if you're interested in seeing how this works read the man pages or help screens for the `ip rule` and `ip route` commands for more details.

The focus so far in this section has been around the topic of IP routing from a host perspective, but it's also possible to use Linux as a full-fledged IP router. As with policy routing, this is a bit of an advanced topic; however, we are going to cover the basic elements in the next section.

Routing as a Router

By default, virtually all modern Linux distributions have IP forwarding *disabled*, since most Linux users don't need IP forwarding. However, Linux has the ability to perform IP forwarding so that it can act as a *router*, connecting multiple IP subnets together and passing (routing) traffic among multiple subnets. To enable this functionality, you must first enable IP forwarding.

To verify if IP forwarding is enabled or disabled, you would run this command (it works on Debian, Ubuntu, and CentOS, although the command might be found at different paths on different systems):

```
vagrant@trusty:~$ <userinput>/sbin/sysctl net.ipv4.ip_forward</userinput>
net.ipv4.ip_forward = 0
vagrant@trusty:~$ <userinput>/sbin/sysctl net.ipv6.conf.all.forwarding</userinput>
net.ipv6.conf.all.forwarding = 0
vagrant@trusty:~$
```

Tip

In situations where a command is found in a different file system location among different Linux distributions, you might find the `which` command to be helpful.

In both cases, the output of the command indicates the value is set to 0, which means it is disabled. You can enable IP forwarding on the fly (without a reboot)—but non-persistently, meaning it will disappear after a reboot—using this command:

```
[vagrant@centos ~]$ <userinput>sysctl -w net.ipv4.ip_forward=1</userinput>
```


This is like the `ip` commands we discussed earlier in that the change takes effect immediately, but the setting will not survive a reboot of the Linux system. To make the change permanent, you must edit `/etc/sysctl.conf` or put a configuration file into the `/etc/sysctl.d` directory. Using individual configuration files in `/etc/sysctl.d` is probably the better approach, especially if you are using a configuration management tool, but

either approach will work. Either way, add this value to either `/etc/sysctl.conf` or to a configuration file in `/etc/sysctl.d`:

```
net.ipv4.ip_forward = 1
```

Or, to enable IPv6 forwarding, add this value:

```
net.ipv6.conf.all.forwarding = 1
```

You can then either reboot the Linux host to make the changes effective, or you can run `sysctl -p <path to file with new setting>`.

Once IP forwarding is enabled, then the Linux system will act as a router. At this point, the Linux system is only capable of performing static routing, so you would need to use the `ip route` command to provide all the necessary routing instructions/information so that traffic could be routed appropriately. However, dynamic routing protocol daemons do exist for Linux that would allow a Linux router to participate in dynamic routing protocols such as BGP or OSPF. Two popular options for integrating Linux into dynamic routing environments are Quagga and BIRD.

Using features like IPTables (or IPTables' successor, NFTables), you can also add functionality like Network Address Translation (NAT) and access control lists (ACLs).

In addition to being able to route traffic at Layer 3, Linux also has the ability to bridge traffic—that is, the ability to connect multiple Ethernet segments together at Layer 2. The next section covers the basics of Linux bridging.

Bridging (Switching)

The Linux bridge offers you the ability to connect multiple network segments together in a protocol-independent way—that is, a bridge operates at Layer 2 of the OSI model instead of at Layer 3 or higher. Bridging—specifically, multiport transparent bridging—is widely used in data centers today in the form of network switches, but most uses of bridging in Linux are centered around various forms of virtualization (either via the KVM hypervisor or via other means like Linux containers). For this reason, we'll only briefly cover the basics of bridging here, and only in the context of virtualization.

Practical Use Case for Bridging

Before we get into the details of creating and configuring bridges, let's look at a practical example of how a Linux bridge would be used.

Let's assume that you have a Linux host with two physical interfaces (we'll use `eth0` and `eth1` as the names of the physical interfaces). Immediately after you create a bridge (a process we'll describe in the following section), your Linux host looks something like the following diagram, Figure [Link to Come]:

[[4.7]] .A Linux bridge with no interfaces image::images/linux/bridge-no-eth.png["A Linux bridge with no physical interfaces"]

The bridge has been created and it exists, but it can't really *do* anything yet. Recall that a bridge is designed to join network segments—without any segments attached to the bridge, there's nothing it can (or will) do. You need to add some interfaces to the bridge.

Let's say you add `eth1` to the bridge `br0`. Now your configuration looks something like Figure [Link to Come]:

[[4.8]] .A Linux bridge with a physical interface image::images/linux/bridge-with-eth.png["A Linux bridge with a physical interface"]

If we were now to attach a virtual machine (VM) to this bridge (a topic which is outside the scope of this book, but is typically accomplished via the use of KVM and Libvirt), then your configuration would look something like Figure [Link to Come]>:

[[4.9]] .A Linux bridge with a physical interface and a VM image::images/linux/bridge-eth-vm.png["A Linux bridge with a physical interface and a VM"]

In this final configuration, the bridge `br0` connects—or *bridges*--the network segment to the VM and the network segment to the physical interface, providing a single Layer 2 broadcast domain from the VM to the NIC (and then on to the physical network). Providing network connectivity for VMs is a very common use case for Linux bridges, but not the only use case. You might also use a Linux bridge to join a wireless network (via a wireless interface on the Linux host) to an Ethernet network (connected via a traditional NIC).

Now that you have an idea of what a Linux bridge can do, let's take a look at creating and configuring Linux bridges.

Creating and Configuring Linux Bridges

To configure Linux bridges, you must first install the correct package. All three of the Linux distributions we're including in this chapter provide support for Linux bridging via a package named "bridge-utils", which you would install using your distribution's package management tool of choice (`yum` for RHEL/CentOS/Fedora, `apt-get` for Debian and Ubuntu).

Once the "bridge-utils" package is installed, you'll have access to the `brctl` command, which is the primary means whereby you'll create, configure, and remove Linux bridges. Like the `ip` commands we discussed earlier, changes made using `brctl` take effect immediately but typically are not persistent.

To create a bridge, you'd use `brctl` with the "addbr" parameter, like this:

```
vagrant@jessie:~$ <userinput>brctl addbr br0</userinput>
```

This would create a bridge named "br0" that contains no interfaces (a configuration similar to Figure [Link to Come] earlier). You can verify this using the `brctl show` command:

```
vagrant@jessie:~$ <userinput>brctl show</userinput>
bridge name     bridge id       STP enabled     interfaces
br0         8000.000000000000     no
vagrant@jessie:~$
```

To add an interface to the bridge, once again use the `brctl` command, this time with the "addif" parameter and the name of the interface to be added to the bridge:

```
vagrant@jessie:~$ <userinput>brctl addif br0 eth1</userinput>
vagrant@jessie:~$
```

Your configuration now looks similar to Figure [Link to Come] earlier.

To remove an interface from a bridge, you'll use `brctl` with the "delif" parameter:

```
[vagrant@centos ~]$ <userinput>brctl delif br0 eth1</userinput>
[vagrant@centos ~]$
```

Any time you're working with bridges, the possibility of a bridging loop is something that must be considered (you are, after all, joining network segments together at Layer 2). The Linux bridge supports Spanning Tree Protocol (STP), which can be enabled with the command `brctl stp`, like this:

```
vagrant@jessie:~$ <userinput>brctl set br0 on</userinput>
```

Replacing "on" with "off" will disable STP. The `brctl showstp` command will display current STP information for the specified bridge.

Finally, to remove a bridge, the command is `brctl delbr` along with the name of the bridge to be removed:

```
vagrant@trusty:~$ <userinput>brctl delbr br0</userinput>
vagrant@trusty:~$
```

Note that there is no need to remove interfaces from a bridge before removing the bridge itself.

All the commands we've shown you so far create non-persistent configurations. In order to make these configurations persistent, you'll need to go back to the interface configuration files we discussed earlier in the section titled "Interface Configuration via Configuration Files." Why? Because Linux treats a bridge as a type of interface—in this case, a *logical* interface as opposed to a *physical* interface.

Because Linux treats bridges as interfaces, you'd use the same types of configuration files we discussed earlier: in RHEL/CentOS/Fedora, you'd use a file in `/etc/sysconfig/network-scripts`, while in Debian/Ubuntu you'd use a configuration stanza in the file `/etc/network/interfaces` (or a standalone configuration file in the `/etc/network/interfaces.d` directory). Let's look at what a bridge configuration would look like in both CentOS and in Debian (Ubuntu will look very much like Debian).

In CentOS 7.1, you'd create an interface configuration file in `/etc/sysconfig/network-scripts` for the bridge in question. So, for example, if you wanted to create a bridge named "br0", you'd create a file named `ifcfg-br0`. Here's a sample interface configuration file for a bridge:

```
DEVICE=br0
TYPE=Bridge
ONBOOT=yes
BOOTPROTO=none
IPV6INIT=no
IPV6_AUTOCONF=no
DELAY=5
STP=yes
```

This creates a bridge named `br0` that has STP enabled. To add interfaces to the bridge, you'd then modify the interface configuration files for the interfaces that should be part of the bridge. For example, if you wanted the interface named `ens33` to be part of the `br0` bridge, your interface configuration file for `ens33` might look like this:

```
DEVICE=ens33
ONBOOT=yes
HOTPLUG=no
BOOTPROTO=none
TYPE=Ethernet
BRIDGE=br0
```

The `BRIDGE` parameter in this configuration file is what ties the `ens33` interface into the `br0` bridge.

One thing you'll note is that neither `br0` nor `ens33` have an IP address assigned. It's best, perhaps, to reason about this in the following way: on a typical network switch, a standard switch port that is configured for Layer 2 only isn't addressable via an IP address. That's the configuration we've replicated here—`br0` is the switch, and `ens33` is the Layer 2-only port that is part of the switch.

If you *did* want an IP address assigned (perhaps for management purposes, or perhaps because you also want to leverage Layer 3 functionality in Linux), then you can assign an IP address to the *bridge*, but *not* to the member interfaces in the bridge. Again, you can make an analogy to traditional network hardware here—it's like giving the switch a management IP address, but the individual Layer 2-only switch ports still aren't addressable by IP. To provide an IP address to the bridge interface, just add the `IPADDR`, `NETMASK`, and `GATEWAY` directives in the bridge's interface configuration file.

Debian (and therefore Ubuntu) are similar. In the case of setting up a bridge on Debian, you would typically add a configuration stanza to the `/etc/network/interfaces` file to configure the bridge itself, like this:

```
iface br0 inet manual
  up ip link set $IFACE up
  down ip link set $IFACE down
  bridge-ports eth1
```

This would create a bridge named "br0" with the `eth1` interface as a member of the bridge. Note that no configuration is needed in the configuration stanzas for the interfaces that are named as members of the bridge.

If you wanted an IP address assigned to the bridge interface, simply change the `inet manual` to `inet dhcp` (for DHCP) or `inet static` (for static address assignment). When using static address assignment, you'd also need to include the appropriate configuration lines to assign the IP address (specifically, the `address`, `netmask`, and optionally the `gateway` directives).

Once you have configuration files in place for the Linux bridge, then the bridging configuration will be restored when the system boots, making it persistent. (You can verify this using `brctl show`.)

That wraps up our discussion of bridging, which in turn wraps up the section on Linux networking and this chapter on Linux.

Summary

In this chapter, we've provided a brief history of Linux, and why it's important to understand a little bit of Linux as you progress down the path of network automation and programmability. We've also supplied some basic information on interacting with Linux, working with Linux daemons, and how Linux networking is configured. Finally, we discussed using Linux as a router as well as explored the functionality of the Linux bridge.

In our introduction to this chapter, we mentioned that one of the reasons we felt it was important to include some information on Linux was because some of the tools we'd be discussing have their roots in Linux or are best used on a Linux system. In the next chapter, Chapter 3, we'll be discussing one such tool: the Python programming language.

Chapter 1: Python

Should Beginners Programmers Learn Python?

Python Interactive Interpreter

Data Types

Strings

Chapter 4. Data Formats

If you've done any amount of exploration into the world of APIs, you've likely heard about terms like JSON, or XML. You may have heard the term "markup language" when discussing one of these.

In the same way that routers and switches require standardized protocols in order to communicate, applications need to be able to agree on some kind of syntax in order to communicate data between them. In this chapter, we'll discuss some of the most commonly used formats within this space, and how you as a network developer can leverage these tools to accomplish tasks.

Introduction to Data Formats

A computer programmer typically uses a wide variety of tools to store and work with data in the programs they build. They may use simple variables (single value), arrays (multiple values), hashes (key/value pairs) or even custom objects built in the sytax of the language they're using.

This is all perfectly standard within the confines of the software being written. However, sometimes a more abstract, portable format is required. For instance, a non-programmer may need to move data in and out of these programs. Another program may have to communicate with this program in a similar way, and they may not even be written in the same language! We need a standard format to allow a diverse set of software to communicate with each other, and for humans to interface with it.

It turns out we have quite a few. With respect to data formats, what we're talking about is text-based representation of data that would otherwise be represented as internal software constructs in memory. All of the data formats that we'll discuss in this chapter have broad support over a multitude of languages and operating systems. In fact, many languages have built-in tools that make it easy to import and export data to these formats, either on the filesystem, or on the network.

So as a network engineer, how does all this talk about software impact you? For one thing, this level of standardization is already in place from a raw network protocol perspective. Protocols like BGP, OSPF, and TCP/IP were concieved out of a necessity to speak a single language across a globally distributed system - the internet! The data formats in this chapter were concieved for very similar reasons - they just operate a higher level in the stack.

Every device you have installed, configured, or upgraded, was given life by a software developer that considered these very topics. Some network vendors saw fit to provide mechanisms that allows operators to interact with a network device using these widely supported data formats - others did not. The goal of this chapter is to help you to understand the value of standardized and simplified formats like these, so that you can use them to your advantage on your Network Automation journey.

For example some configuration models are friendly to automated methods, by representing the configuration model in these data formats like XML or JSON. It is very easy to see the XML representation of a certain dataset in JunOS, for example:

```
# *Source* block
# Use: highlight code listings
# (require `source-highlight` or `pygmentize`)
root#vsrx01> show interfaces | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.1X47/junos">
    <interface-information xmlns="http://xml.juniper.net/junos/12.1X47/junos-interface" junos:style="normal">
        <physical-interface>
            <name>ge-0/0/0</name>
            <admin-status junos:format="Enabled">up</admin-status>
            <oper-status>up</oper-status>
            <local-index>134</local-index>
            <snmp-index>507</snmp-index>
            <link-level-type>Ethernet</link-level-type>
            <mtu>1514</mtu>
            <source-filtering>disabled</source-filtering>
            <link-mode>Full-duplex</link-mode>
            <speed>1000mbps</speed>

< ... output truncated ... >
```

Now, of course this is not very easy on the eyes, but that's not the point. From a programmatic perspective, this is ideal, since each piece of data is given it's own easily parseable field. A piece of software doesn't have to guess where to find the name of the interface - it's located at the well-known and documented tag "name". This is a key difference in understanding the different needs that a software system may have when interacting with infrastructure components, as opposed to a human being on the CLI.

When thinking about data formats at a high-level, it's important to first understand exactly what we intend to do with the various data formats at our disposal. Each was created for a different use case, and understanding these use cases will help you decide which is appropriate for you to use.

Types of Data

Now, we've discussed the use case for data formats, it's important to briefly talk about what kind of data might be represented by these formats. After all - the purpose of these formats is to communicate things like words, numbers, and even complex objects between software instances. If you've taken any sort of programming course, you've likely heard of most of these.

Note

Note that since this chapter isn't about any specific programming languages, these are just generic examples. These datatypes may be represented by different names, depending on their implementation.

The previous chapter, Chapter 3 goes over Python specifically, so be sure to go back and refer to that chapter for Python-specific definitions and usage.

- "*String*" - Arguably, the most fundamental data type is the String. This is a very common way of representing a sequence of numbers, letters, or symbols. If you wanted to represent an English sentence in one of the data formats we'll discuss in this chapter, or in a programming language, you'd probably use a string to do so. In Python, you may see "str", or "unicode" to represent these.
- "*Integer*" - Another is the Integer. There are actually a number (get it?) of data types that have to do with numerical values, but the integer seems to be the first that comes to mind when discussing numerical datatypes. The Integer is exactly what you learned in math class - a whole number, positive or negative. There are other data types use like "float" or "decimal" that you might use to describe non-whole values. Python represents integers using the "int" type.
- "*Boolean*" - One of the simplest data types is Boolean. This is a simple value that is either True or False. This is a very popular type used when a programmer wishes to know the result of an operation, or whether two values are equal to each other, for example. This is known as the "bool" type in Python.
- "*Advanced Data Structures*" - Data types can be organized into complex structures as well. All of the formats we'll discuss in this language support a basic concept known as an Array, or a List in some cases. This is a list of values or objects that can be represented and referenced by some kind of index. There are also key/value pairs, known by many names, such as Dictionaries, Hashes, HashMaps, Hash Tables, or Maps. This is similar to the Array, but the values are organized according to key/value pairs, where both the key or the value can be one of several types of data, like String, Integer, etc. An array can take many forms in Python - the "set", "tuple", and "list" types are all used to represent a sequence of items, but are different from each other in what sort of flexibility they offer. Key/value pairs are represented by the "dict" type.

This is not a comprehensive list, but covers the vast majority of use cases in this chapter. Again, the implementation-specific details for these data types really depends on the context in which they appear. The good news is that all of the data formats we'll discuss in this chapter have wide, and very flexible support for all of these and more.

Now that we've established what data formats are all about, and what types of data may be represented by each of them, let's dive in to some specific examples, and see these concepts written out.

YAML

What is YAML?

If you're reading this book because you've seen some compelling examples of Network Automation online or in a presentation, and you want to learn more, you may have heard of YAML. This is because YAML is a particularly human-friendly data format, and for this reason, it is being discussed before any other in this chapter.

Note

YAML stands for "YAML Ain't Markup Language", which seems to tell us that the creators of YAML desired that it not become just some new markup standard, but a unique attempt to represent data in a human-readable way. Also, the acronym is recursive!

If you compare YAML to the other data formats that we'll discuss like XML or JSON, it seems to do much the same thing: it represents constructs like lists, key/value pairs, strings, integers. However, as you'll soon see, YAML does this is an exceptionally human-readable way. YAML is very easy to read and write if you understand the basic datatypes discussed in the last section.

This is a big reason that an increasing number of tools (see Ansible) are using YAML as a method of defining an automation workflow, or providing a dataset to work with (like a list of VLANs). It's very easy to use YAML to get from zero to a functional automation workflow, or to define the data you wish to push to a device.

At the time of this writing, the latest YAML specification is YAML 1.2, published at http://www.yaml.org/. Also provided on that site is a list of software projects that implement YAML - typically for the purpose of being read in to language-specific data structures and doing something with them. If you have a favorite language, it might be helpful to follow along with the YAML examples in this chapter, and try to implement them using one of these libraries.

Let's take a look at some examples. Let's say we want to use YAML to represent a list of network vendors. If you paid attention in the last section, you'll probably be thinking that we want to use Strings to represent each vendor name - and you'd be correct! This example is very simple:

```
# *Source* block
# Use: highlight code listings
# (require `source-highlight` or `pygmentize`)
---
- Cisco
- Juniper
- Brocade
- VMware
```

Note

You'll notice three hyphens (---) at the top of every example in this section; this is a YAML convention that indicates the beginning of our YAML document.

The YAML specification also states that an ellipsis (....) is used to indicate the end of a document, and that you can actually have multiple instances of triple hyphens (---) to indicate multiple documents within one file or data stream. These methods are typically only used in communication channels (e.g. for termination of messages), which is not a very popular use case, so we won't be using either of these approaches in this chapter.

This YAML document contains three items. We know that each item is a String - one of the nice features of YAML is that we usually don't need quote or double-quote marks to indicate a string, this is something that is usually automatically discovered by the YAML parser (e.g. PyYAML). Each of these items has a hyphen in front of it. Since all three of these Strings are shown at the same level (no indentation), we can say that these Strings compose a list, with a length of 4.

YAML very closely mimics the flexibility of Python's data structures, so we can take advantage of this flexibility without having to write any Python. A good example of this flexibility is shown when we mix data types in this list (not every language supports this):

```
# *Source* block
# Use: highlight code listings
# (require `source-highlight` or `pygmentize`)
---
- Core Switch
- 7700
- False
- ['switchport', 'mode', 'access']
```

In example 6.3, we have another list, this time with a length of four. However, each item is a totally unique type. The first [illegible] String type. The second, 7700 becomes an Integer. The third becomes a Boolean. This "interpretation" is performed [illegible] as PyYAML - which happens to do a pretty good job of inferring what kind of data the user is trying to communicate.

Note

YAML Boolean types are actually very flexible, and accept a wide variety of values here that really end up meaning the same thing when interpreted by a YAML parser.

For instance, you could write "False", as in example 6.3, or you could write "no", "off", or even simply "n". They all end up meaning the same thing - a "False" boolean value. This is a big reason that YAML is often used as a human interface for many software projects.

The fourth is actually itself a list, containing three String items - we've seen our first example of nested data structures in YAML! We've also seen an example of the various ways that some data can be represented. Our "outer" list is shown on separate lines - each item prepended by a hyphen. The inner list is shown on one line, using brackets and commas. These are two ways of writing the same thing - a list.

Note

Note that sometimes it's possible to help the parser figure out the type of data we wish to communicate. For instance, if we wanted the second item to be recognized as a String instead of an Integer, we could enclose it in quotes ("7700"). Another reason to enclose something in quotes would be if a String contained a character that was part of the YAML syntax itself, such as a colon (:).

Refer to the documentation for the specific YAML parser you're using for more information on this.

Early on in this chapter we also briefly talked about key/value pairs, (or dictionaries, as they're called in Python). YAML supports this quite simply. Let's see how we might represent a dictionary with four key/value pairs:

```
# *Source* block
# Use: highlight code listings
# (require `source-highlight` or `pygmentize`)
---
Juniper: Also a plant
Cisco: 6500
Brocade: True
VMware:
  - esxi
  - vcenter
  - nsx
```

Here, our keys are shown as Strings to the left of the colon, and the corresponding value for those keys are shown to the right. If we wanted to look up one of these values in a Python program for instance, we would reference the corresponding key for the value we are looking for.

Similar to lists, dictionaries are very flexible with respect to the data types stored as values. In example 6.4, we are storing a myriad of different data types as the values for each key/value pair.

It's also worth mentioning that - like lists - YAML dictionaries can be written in multiple ways. From a data representation standpoint, Example 6.4 is identical to this:

```
# *Source* block
# Use: highlight code listings
# (require `source-highlight` or `pygmentize`)
---
{Juniper: Also a plant, Cisco: 6500, Brocade: True, VMware: ['esxi', 'vcenter', 'nsx']}
```

Most parsers will interpret these two YAML documents precisely the same, but the first is obviously far more readable. That brings us to the crux of this argument - if you are looking for a more human-readable document, use the more verbose options. If not, you probably don't even want to be using YAML in the first place, and you may want something like JSON or XML. For instance, in an API, readability is nearly irrelevant - the emphasis is on speed and wide software support.

Finally, you can use a hash sign (#) to indicate a comment. This can be on it's own line, or after existing data.

```
# *Source* block
# Use: highlight code listings
# (require `source-highlight` or `pygmentize`)
---
- Cisco    # ocsiC
- Juniper  # repinuJ
- Brocade  # edacorB
- VMware   # erawMV
```

Anything after the hash sign is ignored by the YAML parser.

So as you can see, YAML can be used to provide a friendly way for human beings to interact with software systems. However, YAML is fairly new as far as data formats go. With respect to communication directly between software elements (i.e. no human interaction), other formats like XML and JSON are much more popular, and have much more mature tooling that is conducive to that purpose.

Let's narrow in on a single example to see how exactly a YAML interpreter will read in the data we've written in a YAML document. Let's re-use some previously seen YAML to illustrate the various ways we can represent certain data types:

```
# *Source* block
# Use: highlight code listings
# (require `source-highlight` or `pygmentize`)
---
Juniper: Also a plant
Cisco: 6500
Brocade: True
VMware:
  - esxi
  - vcenter
  - nsx
```

Let's say this yaml document is saved to our local filesystem as "example.yml". Our objective is to use Python to read this YAML file, parse it, and represent the contained data as some kind of variable.

Fortunately, the combination of native Python syntax and a very well-known, third-party YAML parser "pyaml" makes this very easy:

```
import yaml
with open("example.yml") as f:
    result = yaml.load(f)
    print(result)
    type(result)

{'Brocade': True, 'Cisco': 6500, 'Juniper': 'Also a plant', 'VMware': ['esxi', 'vcenter', 'nsx']}
<type 'dict'>
```

This example shows how easy it is to load a YAML file into a Python dictionary. First, a context manager is used to open the file for reading (a very common method for reading any kind of text file in Python), and the "load()" function in the "yaml" module allows us to load this directly into a dictionary called "result". The following lines show that this has been done successfully.

XML

What is XML?

As mentioned in the last section, while YAML is a suitable choice for human-to-machine interaction, other formats like XML and JSON are tend to be favored as the data representation choice when software elements need to communicate with each other. In this section, we're doing to talk about XML, and why it is suitable for this use case.

Note

XML enjoys wide support in a variety of tools and languages, such as the LXML library (http://lxml.de/) in Python. In fact, the XML definition itself is accompanied by a variety of related definitions for things like schema enforcement, transformations, and advanced queries. As a result, this section will attempt only to whet your appetite with respect to XML. You are encouraged to try some of the tools and formats listed on your own.

XML Basics

XML shares some similarities to what we've seen with YAML. For instance - it is inherently hierarchical. We can very easily embed data within a parent construct:

```
# *Source* block
# Use: highlight code listings
# (require `source-highlight` or `pygmentize`)
<device>
  <vendor>Cisco</vendor>
  <model>Nexus 7700</model>
  <osver>NXOS 6.1</osver>
</device>
```

In example 6.9, the <device> element is said to be the root, as it is not indented at all. It is also the parent of the elements nested within it: <vendor>, <model>, and <osver>. It is said that these are the children of the <device> element, and that they are siblings of each other. This is very conducive to storing metadata about network devices, as you can see in this particular example. In an XML document, there may be multiple instances of the "device" tag (perhaps nested within a broader "devices" tag).

You'll also notice that each child element also contains data within - but whereas the root element contained XML children, these tags contain text data. Thinking back to the section on datatypes, it is likely these would be represented by Strings in a Python program, for instance.

XML elements can also have attributes:

```
# *Source* block
# Use: highlight code listings
# (require `source-highlight` or `pygmentize`)
<device type="datacenter-switch" />
```

When a piece of information may have some associated metadata, it may not be appropriate to use a child element, but rather an attribute.

The XML specification has also implemented a namespace system, which helps to prevent element naming conflicts. Developers can use any name they want when creating XML documents, and when a piece of software leverages XML, it's possible that the software would be given two XML elements with the same name, but with different content and purpose.

For instance, an XML document could implement the following

```
# *Source* block
# Use: highlight code listings
# (require `source-highlight` or `pygmentize`)
<device>Palm Pilot</device>
```

This example uses the "device" element name, but clearly is being used for some purpose other than representing a network device, and therefore has a totally different meaning than our switch definition in example 6.9.

Namespaces can help with this, by defining and leveraging prefixes in the XML document itself, using the "xmlns" designation:

```
# *Source* block
# Use: highlight code listings
# (require `source-highlight` or `pygmentize`)
<root>
  <e:device xmlns:e="http://example.org/enduserdevices" >Palm Pilot</e:device>
  <n:device xmlns:n="http://example.org/networkdevices">
    <n:vendor>Cisco</n:vendor>
    <n:model>Nexus 7700</n:model>
    <n:osver>NXOS 6.1</n:osver>
  </n:device>
</root>
```

There is much more involved with writing and reading a valid XML document - check out the w3schools documentation on XML, located at http://www.w3schools.com/xml/

XML Schema Definition (XSD)

XML doesn't have any sort of built-in mechanism to describe the data type within, like what we saw with YAML. Though some of the constructs are similar, many XML parsers don't make the same assumptions that PyYAML does, for instance.

Think of XML like the foundation for a house - it's absolutely crucial for the house to remain standing, but it doesn't have any impact or influence on what's contained within the house, like furniture. We need something to fill the role of an interior designer, someone to go through the house and ensuring that everything is where it needs to be. This is what XSD is designed to do.

XML Schema Definition (http://www.w3schools.com/schema/) allows us to describe the building blocks of an XML document. Using this language, we're able to place constraints on where data should (or should not) be in our XML document. There were previous attempts to provide this functionality (e.g. DTD) but they were limited in their capabilities. Also, XSD is actually written in XML, which simplifies things greatly.

One very popular use case for XSD - or really any sort of schema or modeling language - is to generate source code data structures that match the schema. We can then use that source code to automatically generate XML that is compliant with that schema, as opposed to writing out the XML by hand.

For a concrete example of how this is done in Python, let's look once more at our XML example.

```
# "Source" block
# User highlight code listings
# Requires 'source-highlight' or 'pygmentize'!
<device>
  <vendor>Cisco</vendor>
  <model>Nexus 7700</model>
  <cover>NXOS 6.1</cover>
</device>
```

Our goal is to print this XML to the console. We can do this by first creating an XSD document, then generating Python code from that document using a 3rd party tool. Then, that code can be used to print the XML we need.

Let's write an XSD schema file that describes the data we intend to write out:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="device">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="vendor" type="xs:string"/>
        <xs:element name="model" type="xs:string"/>
        <xs:element name="cover" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

We can use a Python tool called "pyxb" to create a Python file that contains Class object representations of this schema:

```
~$ pyxbgen -u schema.xsd -m schema
```

This will create "schema.py" in this directory. So, if we open a Python prompt at this point, we can import this schema file, and work with it. In example 6.16, we're creating an instance of the generated object, setting some properties on it, and then rendering that into XML using the "toxml" function:

```
import schema
dev = schema.deviceD
dev.vendor = "Cisco"
dev.model = "Nexus"
dev.cover = "6.1"
dev.toxml("utf-8")
'<?xml version="1.0" encoding="utf-8"?><device><vendor>Cisco</vendor><model>Nexus</model><cover>6.1</cover></device>'
```

This is just one way of doing this; there are other 3rd party libraries that allow for code generation from XSD files. Also take a look at "generateDS", located here: http://pythonhosted.org/generateDS/

Note

Some REST APIs (see [Link to Come]) use XML to encode data between software endpoints. Using XSD allows the developer to generate compliant XML much more accurately, and with fewer steps. So, if you come across a REST API on your network device, ask your vendor to provide schema documentation - it will save you some time.

There is much more information about XSD located on the W3C site at: http://www.w3schools.com/schema/

Transforming XML with XSLT

Given that the majority of physical network devices still primarily use a text-based, human-oriented mechanism for configuration, you might have to familiarize yourself with some kind of template format. There are a myriad of them out there, and templates in general are very useful to performing safe and effective network automation.

The next chapter in this book, [Link to Come], goes into detail on templating languages, especially Jinja2. However, since we're talking about XML, we may as well briefly discuss XSLT.

XSLT is a language for applying transformations to XML data - primarily to convert them into XHTML or other XML documents. As with many other languages related to XML, XSLT is defined on the W3C site, and it is located here: http://www.w3schools.com/xsl/

Let's look at a practical example of how to populate an XSLT template with meaningful data so that a resulting document can be achieved. As with our previous examples, we'll leverage some Python to make this happen.

The first thing we need is some raw data to populate our template with. This XML document will suffice:

```
<?xml version="1.0" encoding="UTF-8"?>
<authors>
    <author>
        <firstName>Jason</firstName>
        <lastName>Edelman</lastName>
    </author>
    <author>
        <firstName>Scott</firstName>
        <lastName>Lowe</lastName>
    </author>
    <author>
        <firstName>Matt</firstName>
        <lastName>Oswalt</lastName>
    </author>
</authors>
```

This amounts to a list of authors, each with "firstName" and "lastName" elements. The goal is to use this data to generate an HTML table that displays these authors, via an XSLT document.

An XSLT template to perform this task might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output indent="yes"/>
<xsl:template match="/">
  <html>
  <body>
  <h2>Authors</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th style="text-align:left">First Name</th>
        <th style="text-align:left">Last Name</th>
      </tr>
      <xsl:for-each select="authors/author">
      <tr>
        <td><xsl:value-of select="firstName"/></td>
        <td><xsl:value-of select="lastName"/></td>
      </tr>
      </xsl:for-each>
    </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

A few notes on the XSLT document in example 6.18: first, you'll notice that there is a basic "for-each" construct embedded in what otherwise looks like valid HTML. This is a very standard practice in template language - the static text remains static, and little bits of logic are placed where needed.

It's also worth pointing out that this for-each statement uses a "coordinate" argument (listed as "authors/author") to state exactly which part of our XML document contains the data we wish to use. This is called "XPath", and it is a syntax used within XML documents and tools to specify a location within an XML tree.

Finally, we use the "value-of" statement to dynamically insert (like a variable in a Python program) a value as text from our XML data.

Assuming our XSLT template is saved as "template.xsl", and our data file as "xmldata.xml", we can return to our trusty Python interpreter to combine these two pieces, and come up with a resulting HTML output.

```
from lxml import etree
xslRoot = etree.fromstring(open("template.xsl").read())
transform = etree.XSLT(xslRoot)
xmlRoot = etree.fromstring(open("xmldata.xml").read())
transRoot = transform(xmlRoot)
print etree.tostring(transRoot)

<html><body><h2>Authors</h2><table border="1"><tr bgcolor="#9acd32"><th style="text-align:left">First Name</th><th style="text-align:left">Last Name</th></tr><tr><td>Jason</td><td>Edelman</td></tr><tr><td>Scott</td><td>Lowe</td></tr><tr><td>Matt</td><td>Oswalt</td></tr></table></body></html>
```

This produces a valid HTML table for us, seen in Figure 6.1:

Authors

First Name	Last Name
Jason	Edelman
Scott	Lowe
Matt	Oswalt

Figure 4-1 HTML Table Produced by XSLT

XSLT also provides some additional logic statements:

- <if> - only output the given element(s) if a certain condition is met
- <sort> - sorting elements before writing them as output
- <choose> - a more advanced version of the <if> statement (allows "else if" or "else" style of logic)

It's possible for us to take this example even further, and use this concept to create a network configuration template, using configuration data defined in XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<interfaces>
    <interface>
        <name>GigabitEthernet0/0</name>
        <ipv4addr>192.168.0.1 255.255.255.0</ipv4addr>
    </interface>
    <interface>
        <name>GigabitEthernet0/1</name>
        <ipv4addr>172.16.31.1 255.255.255.0</ipv4addr>
    </interface>
    <interface>
        <name>GigabitEthernet0/2</name>
        <ipv4addr>10.2.2.1 255.255.254.0</ipv4addr>
    </interface>
</authors>

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.example.org/routerconfig">

<xsl:template match="/">
    <xsl:for-each select="interfaces/interface">
```

```
    interface <xsl:value-of select="name"/><br />
        ip address <xsl:value-of select="ipv4addr"/><br />
    </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

With the XML and XSLT documents shown above in examples 6.20 and 6.21, we can get a rudimentary router configuration in the same way we generated an HTML page:

```
interface GigabitEthernet0/0
ip address 192.168.0.1 255.255.255.0
interface GigabitEthernet0/1
ip address 172.16.31.1 255.255.255.0
interface GigabitEthernet0/2
ip address 10.3.2.1 255.255.254.0
```

As you can see, it's possible to produce a network configuration by using XSLT. However, it is admittedly a bit cumbersome. It's likely that you will find Jinja2 a much more useful templating language for creating network configurations - it has a lot of features that are conducive to network automation. Jinja2 is covered in the next chapter of this book. [Link to Come]

XQuery

In the previous section, we alluded to using XPath in our XSLT documents to very particularly locate specific nodes in our XML document. However, if we needed to perform a more advanced lookup, we need a bit more than a simple coordinate system.

XQuery leverages tools like XPath to find and extract data from an XML document. For instance, if you are accessing the REST API of a router or switch using Python, you may have to write a bit of extra code to get to the exact portion of the XML output that you wish to use. Alternatively you can use XQuery immediately upon receiving this data to present only the relevant data to your Python program.

XQuery is a powerful tool - almost like a programming language unto itself. For more info on XQuery, checkout the W3C specification, located at http://www.w3schools.com/xquery/.

JSON

What is JSON?

So far in this chapter, we've discussed YAML, a tool well suited for human interaction, and easy import into common programming language data structures. We've also discussed XML, which isn't the most attractive format to look at, but has a rich ecosystem of tools and wide software support. In this section, we discuss JSON, which combines a few of these strengths into one data format.

JSON was invented at a time when web developers were in need of a lightweight communication mechanism between web servers and applets embedded within web pages. XML was around at this time of course, but it proved a bit too bloated to meet the needs of the ever-demanding internet.

You may have also noticed that YAML and XML differ in a big way with respect to how these two data formats map to the data model of most programming languages like Python. With libraries like PyYAML, importing a YAML document into source code is nearly effortless. However, with XML there is usually a few more steps needed, depending on what you want to do.

For these and other reasons, Javascript Object Notation (JSON) burst onto the scene in the early 2000s - it aimed to be a lightweight version of XML, more suited to the data models found within popular programming languages.

Note

Note that JSON is widely considered to be a subset of YAML. In fact, many popular YAML parsers can also parse JSON data as if it were YAML. However, some of the details of this relationship are a bit more complicated. See the YAML specification section addressing this (http://yaml.org/spec/1.2/spec.html#id2759572) for more information.

JSON Basics

In the previous section, we showed an example of how three authors may be represented in an XML document:

```
# *Source* block
# Use: highlight code listings
# Requires `source-highlight` or `pygmentize`)
<authors>
    <author>
        <firstName>Jason</firstName>
        <lastName>Edelman</lastName>
    </author>
    <author>
        <firstName>Scott</firstName>
        <lastName>Lowe</lastName>
    </author>
    <author>
        <firstName>Matt</firstName>
        <lastName>Oswalt</lastName>
    </author>
</authors>
```

To illustrate the difference between JSON and XML, specifically with respect to JSON's more lightweight nature, here is an equivalent data model provided in JSON:

```
# *Source* block
# Use: highlight code listings
# Requires `source-highlight` or `pygmentize`)
{"authors":[
    {"firstName":"Jason", "lastName":"Edelman"},
    {"firstName":"Scott", "lastName":"Lowe"},
    {"firstName":"Matt", "lastName":"Oswalt"}
]}
```

As you can see, this occupies a fraction of the space as the XML counterpart. No wonder JSON was more attractive than XML in the early 2000s, when "Web 2.0" was just getting started!

Let's look specifically at some of the features. You'll notice that the whole thing is wrapped in curly braces "{}" - this is very common, and it indicates that JSON objects are contained inside. You can think of "Objects" as key/value pairs, or dictionaries as we discussed in the section on YAML. JSON objects always use Strings when describing the keys in these constructs.

In this case, our key is "authors", and the value for that key is a JSON List. This is also equivalent to the List format we discussed in YAML - an ordered list of zero or more values. This is indicated by the straight brackets "[]".

Contained within this list are three objects (separated by commas and a newline), each with two key/value pairs. The first pair describes the author's first name (key of "firstName") and the second, the author's last name (key of "lastName").

We discussed the basics of data types at the beginning of this chapter, but let's take an abbreviated look at the supported datatypes in JSON - you'll find they match our experience from YAML quite nicely:

- ***Number*** - A signed decimal number
- ***String*** - A collection of characters, such as a word or a sentence
- ***Boolean*** - True or False
- ***Array*** - An ordered list of values; items do not have to be the same type (enclosed in straight braces: [])
- ***Object*** - An unordered collection of key/value pairs; keys must be Strings (enclosed in curly braces: {})
- ***null*** - Empty value. Uses the word "null"

Let's work with JSON in a few different programming languages and see what we can do with it.

Working with JSON in a Language

JSON enjoys wide support across a myriad of languages. In fact, you will often be able to simply import a JSON data structure into constructs of a given language, simply with a one-line command. Let's take a look at some examples.

Our JSON data is stored in a simple text file:

```
{
  "hostname": "IOSXEMU1",
  "vendor": "Cisco",
  "isAlive": true,
  "uptime": 123456,
  "users": {
    "admin": 15,
    "storage": 10,
  },
  "vlans": [
    {
      "vlan_name": "VLAN10",
      "vlan_id": 10
    },
    {
      "vlan_name": "VLAN20",
      "vlan_id": 20
    }
  ]
}
```

Our goal is to import the data found within this file into the constructs used by our language of choice.

First, let's use Python. Python has tools for working with JSON built right in to it's standard library, aptly called the "json" package. In this example, we define a JSON data structure (borrowed from the Wikipedia entry on JSON) within the Python program itself - but this could easily also be retrieved from a file or a REST API. As you can see, importing this JSON is fairly straightforward (see the inline comments):

```
# Python contains very useful tools for working with JSON, and they're
# part of the standard library, meaning they're built into Python itself.
import json

# We can load our JSON file into a variable called "data"
with open("json-example.json") as f:
    data = f.read()

# json_dict is a dictionary, and json.loads takes care of
# placing our JSON data into it.
json_dict = json.loads(data)

# Printing information about the resulting Python data structure
print("The JSON document is loaded as type {0}\n".format(type(json_dict)))
print ("Now printing each item in this document and the type it contains")
for k, v in json_dict.items():
    print(
        "-- The key {0} contains a {1} value.".format(str(k), str(type(v)))
    )
```

Those last few lines are there so we can see exactly how Python views this data once imported. The output that results from running this Python program is as follows:

```
~ $ python json-example.py

The JSON document is loaded as type <type 'dict'>

Now printing each item in this document and the type it contains
-- The key uptime contains a <type 'int'> value.
-- The key isAlive contains a <type 'bool'> value.
-- The key users contains a <type 'dict'> value.
-- The key hostname contains a <type 'unicode'> value.
-- The key vendor contains a <type 'unicode'> value.
-- The key vlans contains a <type 'list'> value.
```

Note

You might be seeing the "unicode" datatype for the first time. It's probably best to just think of this as roughly equivalent to the "str" (string) type, discussed in the Python chapter, Chapter 3.

In Python, the "str" type is actually just a sequence of bytes, whereas unicode specifies an actual encoding.

JSON Schema

We discussed the tools that allow us to enforce a schema within XML - that is, be very particular with the type of data contained within an XML document. JSON also has a mechanism for schema enforcement, and it's aptly named "JSON Schema". This specification is defined at http://json-schema.org/documentation.html, but has also been submitted as an Internet Draft.

A Python implementation of JSON Schema (https://pypi.python.org/pypi/jsonschema) exists, and implementations in other languages can also be found.