Eric Chou

Foreword by:
Rich Groves
Director of R&D at A10 Networks

# Mastering Python Networking

Advanced networking with Python

Packt>

# Mastering Python Networking

Advanced networking with Python

Eric Chou

Packt

**BIRMINGHAM - MUMBAI**

< html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" "http://www.w3.org/TR/REC-html40/loose.dtd">

# Mastering Python Networking

First published: June 2017

Production reference: 1230617

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78439-700-5

www.packtpub.com

# Credits

| | |
|---|---|
| **AuthorÂ Â**<br><br>Eric Chou | **Copy Editor Â**<br><br>Gladson Monteiro |
| **ReviewerÂ Â**<br><br>Allen Su | **Project Coordinator**<br><br>Virginia Dias |
| **Commissioning Editor Â**<br><br>Kartikey Pandey | **Proofreader Â**<br><br>Safis Editing |

| | |
|---|---|
| | |
| **Acquisition Editor** Â<br>Meeta Rajani | **Indexer** Â<br>Rekha Nair |
| **Content Development Editor** Â<br>Sweeny Dias | **Graphics** Â<br>Kirk D'Penha |
| **Technical Editors**<br>Prashant Chaudhari<br>Komal Karne | **Production Coordinator** Â<br>Aparna Bhagat |

# Foreword

Over my 20 years in computer networking, I have been lucky to work on a handful of popular, large-scale systems. If you had asked me in 2001, I would have toldÂ you my favorite project was AOL Instant Messenger. To scale to the size that we did in the early days, we had to create our own APIs for provisioning, security, and control over all aspects of the service--all aspects save the physical network, that is. This was a pain, but we lived with it as compute and storage needs were dynamic while network needs stayed reasonably static. In 2010, while working on what would become Microsoft Azure, it was clear that usage patterns have changed. Compute and storage have now beenÂ disaggregated, so it is more important than ever to have non-blocking connectivity and the ability to utilize any overlay technology required by the workload.Â

Within the last few years, we have seen quite a shift in thinking, from network device vendors adding APIs and Python scripting environments to their devices in the hope of them being driven programmatically. In this book, Eric Chou helps us gain a thorough understanding of interfacing with networks and network devices using Python, from interacting with a single device to large numbers of devices with complex automation using Ansible. Then, he takes us to my favorite topics of network monitoring and security, as well as an array of OpenFlow projects controlled through the Python-based Ryu SDN controller.

Eric and I worked together at Microsoft, where we built *Microsoft DEMon*, an Openflow-based network packet broker. Eric's deep understanding of Python and his love for automation show in every project we work on together. I have had the pleasure to see many of Eric's examples from thisÂ book used in real life and proven in actual projects. In *Mastering Python Networking*, Eric is adding some theory and a set of practical examples taken from real problems that he has solved.

**Rich Groves**

Director of R&D at A10 Networks

# About the Author

**Eric Chou** is a seasoned technologist with over 16 years of experience. He has managed some of the largest networks in the industry while working at Amazon and Microsoft and is passionate about network automation and Python. He shares this deep interest in these fields through his teachings as a Python instructor, blogger, and active contributor to some of the popular Python open source projects. Currently, Eric holds two patents in IP Telephony and is a principal engineer at A10 Networks with a focus on product research and development in the field of security.

*I would like to thank members of the Packt Publishing team--Meeta Rajani, Prashant Chaudhari, and Sweeny Dias--and my technical reviewer, Allen Su, for making my dream of writing this book a reality. Thank you for the opportunity to work with you and for your tireless effort and support.*

*I would also like to thank the open source and Python community members for generously sharing their knowledge and code with the public. Without their contributions, many of the projects referenced in this book would not have been possible.*

*Iâ€™m also grateful for the people who have helped me in my career and shaped my professional path. Iâ€™d like to thank all who have been part of my professional growth, especially my mentors at each stage of my career: Hup Chen, Monika Machado, and Rich Groves. Thank you for inspiring me to be the best I can be.*

*Finally, I would like to thank my wife and my two beautiful daughters for their support. They provided me the freedom and understanding I needed to focus on and complete the book.*

# About the Reviewer

**Allen Su**, CCIE no. 13871 (Routing and Switching, Service Provider, Security), is currently a seniorÂ network andÂ cloud security engineer at Microsoft, where he is driving innovative design and engineering of secure edge services and automation capabilities.

Allen is a networking industry veteran, having spent the last 15 years in various engineering and architectural roles. Prior to Microsoft, Allen was at Arista Networks, where he worked with and helped some marquee cloud providers build their cloud-scale networks and define and develop their network automation framework and strategy.Â  Before his tenure at Arista, Allen spent a significant portion of his career at Cisco, learning the intricacies of networking, which he leveraged to design, architect, and build some of the worldâ€™s largest networks.

*I would like to thank Eric Chou for giving me the opportunity and privilege to review his hard work. It wasÂ a great learning journey for me personally, and I think Iâ€™ve gained way more from this journey than what I could give. Eric is the one who inspired me to begin into the network automation journey five years ago, and he continues to be an inspiration for me in this regard.*

*I would also like to thank my wife, Cindy, for the support and love she has always given me, no matter what I set out to do. It would have been a lot tougher to review this book without her support and understanding.*

# Table of Contents

# Preface

As Charles Dickens wroteÂ in *A Tale of Two Cities*, *"It was the best of times, it was the worse of times, it was the age of wisdom, it was the age of foolishness."* His seemingly contradictory words perfectly describe the chaos and mood felt during a time of change and transition. We are no doubt experiencing a similar time with the rapid changes in network engineering fields. As software development becomes more integrated into all aspects of networking, the traditional command-line interface and vertically integrated network stack methods are no longer the best ways to manage todayâ€™s networks. For network engineers, the changes weâ€™re seeing are full of excitement and opportunities yet challenging, particularly for those who need to quickly adapt and keep up. This book is written to help ease the transition for networking professionals by providing a practical guide that addresses how to evolve from a traditional platform to one built on software-driven practices.

In this book, we use Python as ourÂ programming language of choice to master network engineering tasks. Python is an easy-to-learn, high-level programming language that can effectively complement network engineersâ€™ creativity and problem-solving skills to streamline daily operations. Python is becoming an integral part of many large-scale networks, and through this book, I hope to share with you the lessons I've learned.

A time of change presents great opportunities for technologicalÂ advancement. The concepts and tools in this book have helped me tremendously in my career, and I hope they can do the same for you.Â

# What this book covers

Chapter 1, *Review of TCP/IP Protocol Suite and Python Language*, reviews the fundamental technologies that make up Internet communication today, from the OSI and client-server models to TCP, UDP, and IP protocol suites. It will also review the basics of the Python language in its types, operators, loops, functions, and packages.

Chapter 2, *Low-Level Network Device Interactions*, uses practical examples to illustrate how to use Python to execute commands on a network device. It will discuss the challenges of having a CLI-only interface in automation. The chapter will use PExpect and Paramiko library examples.

Chapter 3, *API and Intent-Driven Networking,* discusses the newer network devices that support Application Program Interfaces (APIs) and other high-level interaction methods. It also illustrates a tool that allows network engineers to abstract the low-level tasks when scripting in Python while focusing on the design and what you want the network to achieve. A discussion of Cisco NX-API, Juniper PyEZ, and Arista PyEAPI among other technologies is also included.

Chapter 4, *The Python Automation Framework - Ansible Basics*, discusses the basics of Ansible, an open source, Python-based automation framework. Ansible goes one step further from APIs and focuses on network intents and device interaction. In this chapter, we will cover the advantages of using Ansible, its architecture, and practical examples of Ansible with Cisco, Juniper, and Arista devices.

Chapter 5, *The Python Automation Framework - Ansible Advance Topics*, builds on the knowledge obtained from the previous chapter and covers the more advanced Ansible concepts such as conditionals, loops, templates, variables, vaults, and roles. It will also introduce how to write your own Ansible module that fits in your network environment.

Chapter 6, *Network Security with Python*, introduces several Python tools to help you secure your network. It will discuss using Scapy for security testing, using Ansible to quickly implement access lists, and forensic analysis with syslog and UFW using Python.

Chapter 7, *Network Monitoring with Python - Part 1*, covers monitoring the network using various tools. It will use SNMP and PySNMP for queries to obtain device information. From the results, we will use Matplotlib and Pygal to visualize the results. The chapter will end with Cacti examples and how to use Python scripts as input source.

Chapter 8, *Network Monitoring with Python - Part 2*, covers more network-monitoring tools. It will start with using Graphviz to graph network graphs automatically from LLDP information. It will move to introducing push-based network monitoring using NetFlow and other similar technologies. We will use Python to decode flow packets as well as use ntop to visualize flow information. We will also introduce hosted Elasticsearch as a way to complement network monitoring.

Chapter 9, *Building Network Web Services with Python*, shows you how to use the Python web framework, Flask, to create your own API on the network level. The network-level API offers benefits such as abstracting the requester away from network details, consolidating and customizing operations, and better security by limiting the exposure of available operations.

Chapter 10, *OpenFlow Basics*, covers the basics of OpenFlow, a protocol that many credit as the technology that stared the software-defined networking movement. The protocol separates the control and data plane of network devices, which allows network operators to quickly prototype and innovate new features and functions. We will use the Python-based controller Ryu as well as Mininet to simulate an OpenFlow network. We will introduce examples of OpenFlow layer 2 switches and firewalls.

Chapter 11, *Advanced OpenFlow  Topics*, introduces advanced OpenFlow topics by building additional network applications and features using OpenFlow. We will start with building an OpenFlow router with static flows and then enhance the application with the REST API and build BGP speaking capabilities to interact with traditional networks. The chapter will end with using the firewall applications example as a way to virtualize traditional network functions.

Chapter 12, *OpenStack, OpenDaylight, and NFV*, covers other software-defined networking projects: OpenStack, OpenDaylight, and Network Function Virtualization. We will focus on the OpenStack network project, Neutron, in the chapter to discuss the service architecture and how to try out OpenStack with TryStack and DevStack. The chapter will also cover a basic OpenDaylight controller example for a simple hub with Mininet.

Chapter 13, *Hybrid SDN*, uses the knowledge from previous chapters and discusses various considerations and methods for implementing a software-driven network. We will discuss preparing your network for SDN and OpenFlow and considerations for greenfield deployment, controller redundancy, BGP interoperability, monitoring integration, controller secure TLS connection, and physical switch selection for your network.

## What you need for this book

It is strongly recommended that you follow and practice the examples given in this book. To complete the examples, you will need a host machine that runs Python 2.7 and 3.4, with enough administrative permissions to install the tools introduced in the book. The host machine can be a virtual machine and should preferably run a flavor of Linux. In the book, we'll use Ubuntu 16.04, but other Linux distributions should work as well. You might need to tweak your settings accordingly. In addition, either physical or virtual network equipment is needed to test your code.

## Who this book is for

This book is ideal for IT professionals and ops engineers who already manage groups of network devices and would like to expand their knowledge on using Python to overcome networking challenges. Basic knowledge of networking and Python are recommended.

# Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: We can include other contexts through the use of the include directive.

A block of code is set as follows:

```
# This is a comment
   print("hello world")
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
         [
           {
             "hosts": "192.168.199.170",
             "tasks": [
             "name": "check disk usage",
             "shell": "df > df_temp.txt"
            ]
           }
         ]
```

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

```
$ ssh-keygen -t rsa <<<< generates public-private key pair on the host machine
$ cat ~/.ssh/id_rsa.pub <<<< copy the content of the output and paste it to the
~/.ssh/authorized_keys file on the target host
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: Clicking the Next button moves you to the next screen.

*Warnings or important notes appear in a box like this.*

*Tips and tricks appear like this.*

## Reader feedback

## Customer support

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

# Downloading the example code

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

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Mastering-Python-Networking. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

## Downloading the color images of this book

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

# Errata

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

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

## Questions

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

# Review of TCP/IP Protocol Suite and Python Language

This book assumes that you have the basic understandings of networking protocols and the Python language. In my experience, a typical system, network engineer, or developer might not remember the exact TCP state machine on a daily basis (I know I don't), but he/she would be familiar with the basics of the OSI model, the TCP and UDP operations, IP headers, and more such.

This chapter will do a very quick revision on the relevant networking topics. In the same view, we will also do a high-level review on the Python language, just enough so that readers who do not code in Python on a daily basis can have a ground to walk on for the rest of the book.Â

Specifically, we will cover the following topics:

- The internet overview
- The OSI and client-server Model
- TCP, UDP, IP protocolÂ Suites
- Python syntax, types, operators, and loops
- Extending Python with functions, classes, and packages

Worry not if you feel you need further information, as by no means do I think the information presented in this chapter is exhaustive. Do check out the reference section for this chapter to read more on your topic of interest.Â

# The internet overview

What is the internet? This seemingly easy question might receive different answers depending on your background. The internet means different things to different people, the young, the older, the student, the teacher, the business person, a poet, all could have a different answer to the question.

To a network engineer and systems engineer by extension, the internet is a global computer network providing a variety of information. This global computer network system is actually a web of internetwork connecting large and small networks together. Imagine your home network; it will consist of a home switch connecting your smart phone, tablet, computers, and TV together, so they can communicate with each other. Then, when it needs to communicate to the outside world, it passes the information on to the home router that connects your home network to a larger network, often appropriately called the **Internet Service Provider** (**ISP**). Your ISP often consists of edge nodes that aggregate the traffic to their core network. The core network's function is to interconnect these edge networks via a higher speed network. At special edge nodes, your ISP is connected to other ISPs to pass your traffic appropriately to your destination. The return path from your destination to your home computer, tablet, or smart phone may or may not follow the same path through all of these networks back to your screen.

Let's take a look at the components making up this web of networks.

## Servers, hosts, and network components

**Hosts** are end nodes on the networks that communicate to other nodes. In today's world, a host can be the traditional computer, or it can be your smart phone, tablet, or TV. With the rise of **Internet of Things** (**IoT**), the broad definition of host can be expanded to include IP camera, TV set-top boxes, and the ever-increasing type of sensors that we use in agriculture, farming, automobiles, and more. With the explosion of the number of hosts connected to the internet, all of them need to be addressed, routed, and managed, and the demand for proper networking has never been greater.

Most of the time when we are on the internet, we request for services. The service can be a web page, sending or receiving emails, transferring files, and such. These services are provided by **servers**. As the name implies, they provide services to multiple nodes and generally have higher levels of hardware specification because of it. In a way, servers are special super nodes on the network that provide additional capabilities to its peers. We will look at servers later on in the client-server section.

If one thinks of servers and hosts as cities and towns, the **network components** are the roads and highways to connect them together. In fact, the term "information superhighway" comes to mind when describing the network components that transmit the ever increasing bits and bytes across the globe. In the OSI model that we will look at in a bit, these network components are the routers and switches that reside on layer two and three of the model as well as the layer on fiber optics cable, coaxial cable, twisted copper pairs, and some DWDM equipment, to name a few.

Collectively, the hosts, servers, and network components make up the internet as we know today.

## The rise of datacenter

In the last section, we looked at different roles that servers, hosts, and network components play in the internetwork. Because of the higher hardware capacity that the servers demand, they are often put together in a central location, so they can be managed more efficiently. We will refer to these locations as **datacenters**.

## Enterprise datacenters

In a typical enterprise, the company generally has the need for internal tools such as e-mails, document storage, sales tracking, ordering, HR tools, and knowledge sharing intranet. These terms translate into file and mail servers, database servers, and web servers. Unlike user computers, these are generally high-end computers that require higher power, cooling, and network connection. A byproduct of the hardware is also the amount of noise they make. They are generally placed in a central location called the **Main Distribution Frame** (**MDF**) in the enterprise to provide the necessary power feed, power redundancy, cooling, and network speed.

To connect to the MDF, the user's traffic is generally aggregated at a location, which is sometimes called the **Intermediate Distribution Frame** (**IDF**) before they are connected to the MDF. It is not unusual for the IDF-MDF spread to follow the physical layout of the enterprise building or campus. For example, each building floor can consist of an IDF that aggregates to the MDF on another floor. If the enterprise consists of several buildings, further aggregation can be done by combining the building traffic before connecting them to the enterprise datacenter.

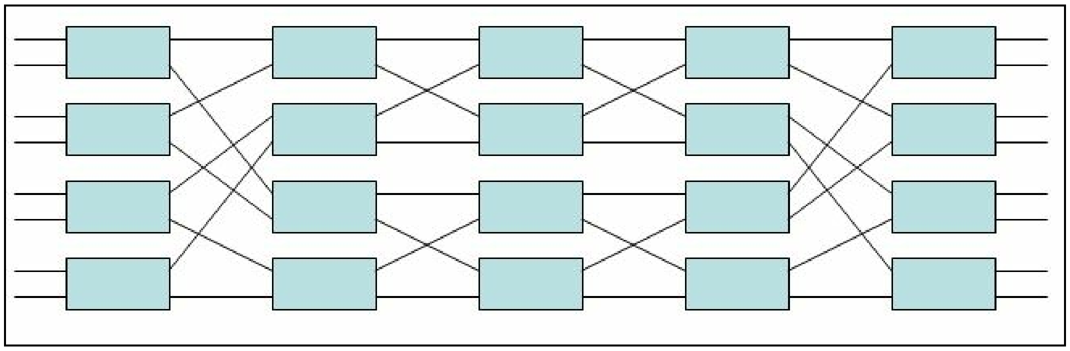
The enterprise datacenters generally follow the three layers of access, distribution, and core. The access layer is analogous to the ports each user connects to; the IDF can be thought of as the distribution layer, while the core layer consists of the connection to the MDF and the enterprise datacenters. This is, of course, a generalization of enterprise networks as some of them will not follow the same model.

## Cloud datacenters

With the rise of cloud computing and software or infrastructure as a service, we can say that the datacenters cloud providers build the cloud datacenters. Because of the number of servers they house, they generally demand a much, much higher capacity of power, cooling, network speed, and feed than any enterprise datacenter. In fact, the cloud datacenters are so big they are typically built close to power sources where they can get the cheapest rate for power, without losing too much during transportation of the power. They can also be creative when it comes to cooling down where the datacenter might be build in a generally cold climate, so they can just open the doors and windows to keep the server running at a safe temperature. Any search engine can give you some of the astounding numbers when it comes to the science of building and managing these cloud datacenters for the likes of Amazon, Microsoft, Google, and Facebook:

Utah Data Center (source: https://en.wikipedia.org/wiki/Utah_Data_Center)

The services that the servers at datacenters need to provide are generally not cost efficient to be housed in any single server. They are spread among a fleet of servers, sometimes across many different racks to provide redundancy and flexibility for service owners. The latency and redundancy requirements put a tremendous amount of pressure on the network. The number of interconnection equates to an explosive growth of network equipment; this translates into the number of times these network equipment need to be racked, provisioned, and managed.



CLOS Network (source: https://en.wikipedia.org/wiki/Clos_network)

In a way, the cloud datacenter is where network automation becomes a necessity. If we follow the traditional way of managing the network devices via a Terminal and command-line interface, the number of engineering hours required would not allow the service to be available in a reasonable amount of time. This is not to mention that human repetition is error prone, inefficient, and a terrible waste of engineering talent.

The cloud datacenter is where the author started his path of network automation with Python a number of years ago, and never looked back since.

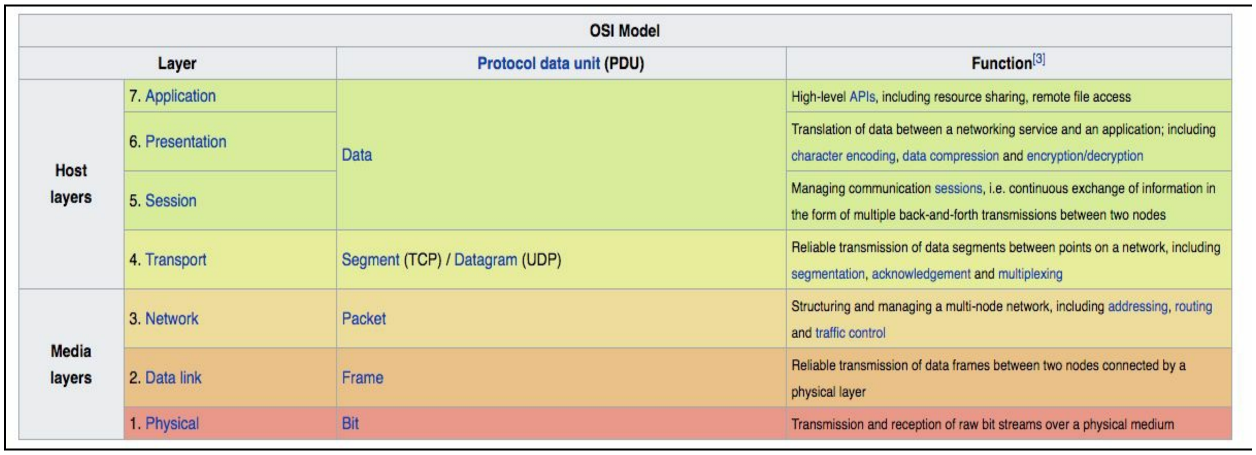
## Edge datacenters

If we have sufficient computing power at the datacenter level, why keep anything anywhere else but at these datacenters? All the connections will be routed back to the server providing the service, and we can call it a day. The answer, of course, depends on the use case. The biggest limitation in routing back the request and session all the way back to the datacenter is the latency introduced in the transport. In other words, network is the bottleneck. As fast as light travels in a vacuum, the latency is still not zero. In the real world, the latency would be much higher when the packet is traversing through multiple networks and sometimes through undersea cable, slow satellite links, 3G or 4G cellular links, or Wi-Fi connections.

The solution? Reduce the number of networks the end user traverse through as much as possible by one. Be as directly connected to the user as possible; and two: place enough resources at the edge location. Imagine if you are building the next generation of video streaming service. In order to increase customer satisfaction with a smooth streaming, you would want to place the video server as close to the customer as possible, either inside or very near to the customer's ISP. Also, the upstream of my video server farm would not just be connected to one or two ISPs, but all the ISPs that I can connect to with as much bandwidth as needed. This gave rise to the peering exchanges edge datacenters of big ISP and content providers. Even when the number of network devices are not as high as the cloud datacenters, they too can benefit from network automation in terms of the increased security and visibility automation brings. We will cover security and visibility in later chapters of this book.

# The OSI model

No network book seems to be complete without first going over the **Open System Interconnection** (**OSI**) model. The model is a conceptional model that componentizes the telecommunication functions into different layers. The model defines seven layers, and each layer sits independently on top of another one, as long as they follow defined structures and characteristics. For example, the network layer, such as IP, can sit on top of different type of data link layer such as the Ethernet or frame relay. The OSI reference model is a good way to normalize different and diverse technologies into a set of common language that people can agree on. This greatly reduces the scope for parties working on the individual layers and allows them to go in depth on the specific tasks without worrying about compatibility:



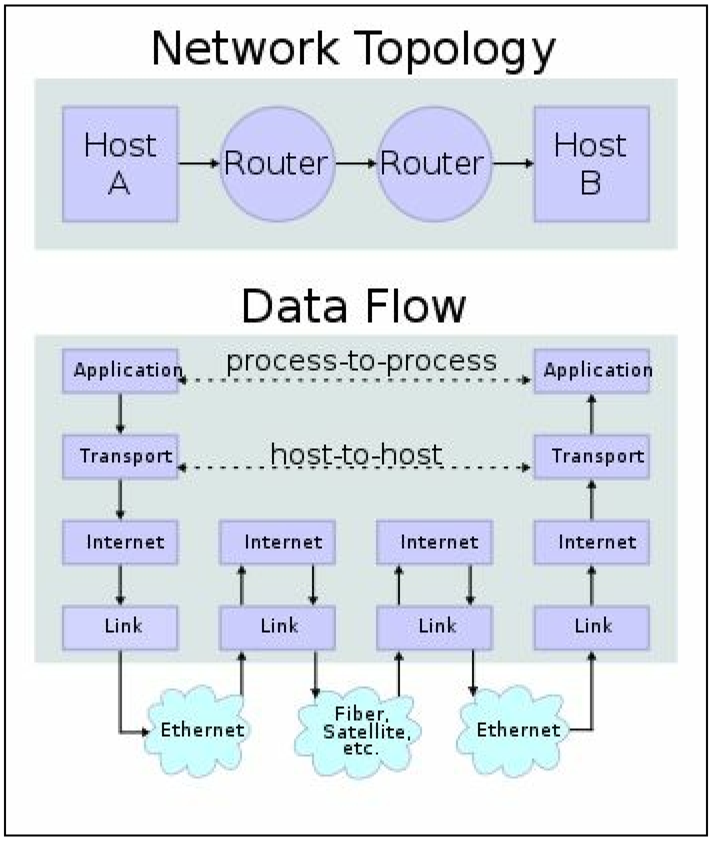
OSI Model

| Layer | | Protocol data unit (PDU) | Function[3] |
|---|---|---|---|
| Host layers | 7. Application | Data | High-level APIs, including resource sharing, remote file access |
| | 6. Presentation | | Translation of data between a networking service and an application; including character encoding, data compression and encryption/decryption |
| | 5. Session | | Managing communication sessions, i.e. continuous exchange of information in the form of multiple back-and-forth transmissions between two nodes |
| | 4. Transport | Segment (TCP) / Datagram (UDP) | Reliable transmission of data segments between points on a network, including segmentation, acknowledgement and multiplexing |
| Media layers | 3. Network | Packet | Structuring and managing a multi-node network, including addressing, routing and traffic control |
| | 2. Data link | Frame | Reliable transmission of data frames between two nodes connected by a physical layer |
| | 1. Physical | Bit | Transmission and reception of raw bit streams over a physical medium |

OSI Model (source: https://en.wikipedia.org/wiki/OSI_model)

The OSI model was initially worked on in the late 1970's and later on published jointly by the **International Organization for Standardization** (**ISO**) and the now called **Telecommunication Standardization Sector** of the **International Telecommunication Union** (**ITU-T**). It is widely accepted and commonly referred to when introducing a new topic in telecommunication.

Around the same time period of the OSI model development, the internet was taking shape. The reference which model they used is often referred to as the TCP/IP model since the **Transmission Control Protocol** (**TCP**) and **Internet Protocol** (**IP**), because originally, this was what the protocol suites contained. It is somewhat similar to the OSI model in the sense that they divide the end-to-end data communication into abstraction layers. What is different is the model combined layers 5 to 7 in the OSI model into the **Application** layer while the **Physical** and **Data link** layers are combined into the **Link** layer:



Internet Protocol Suite (source: https://en.wikipedia.org/wiki/Internet_protocol_suite)

Both the OSI and TCP/IP models are useful in providing a standard for providing an end-to-end data communication. However, for the most part, we will refer to the TCP/IP model more since that is what the internet was built on. We will specify the OSI model when needed, such as when we are discussing the web framework in the upcoming chapters.

## Client server models

The reference models demonstrated a standard way for data communication between two nodes. Of course by now, we all know that not all nodes are created equally. Even in its DARPA-net days, there were workstation nodes, and there were nodes with the purpose of serving content to other nodes. These units typically have higher hardware specification and are managed more closely by the engineers. Since these nodes provide resources and services to others, they are typically referred to as servers. The servers are typically sitting idle, waiting for clients to initiate requests for its resources. This model of distributed resources that are "asked for" by the client is referred to as the Client-server model.

Why is this important? If you think about it for a minute, the importance of networking is really based  on this Client-server model. Without it, there is really not a lot of need for network interconnections. It is the need to transfer bits and bytes from client to server that shines a light on the importance of network engineering. Of course, we are all aware of how the biggest network of them all, the internet, is transforming the lives of all of us and continuing to do so.

How, you asked, can each node determine the time, speed, source, and destination each time they need to talk to each other? This brings us to the network protocols.

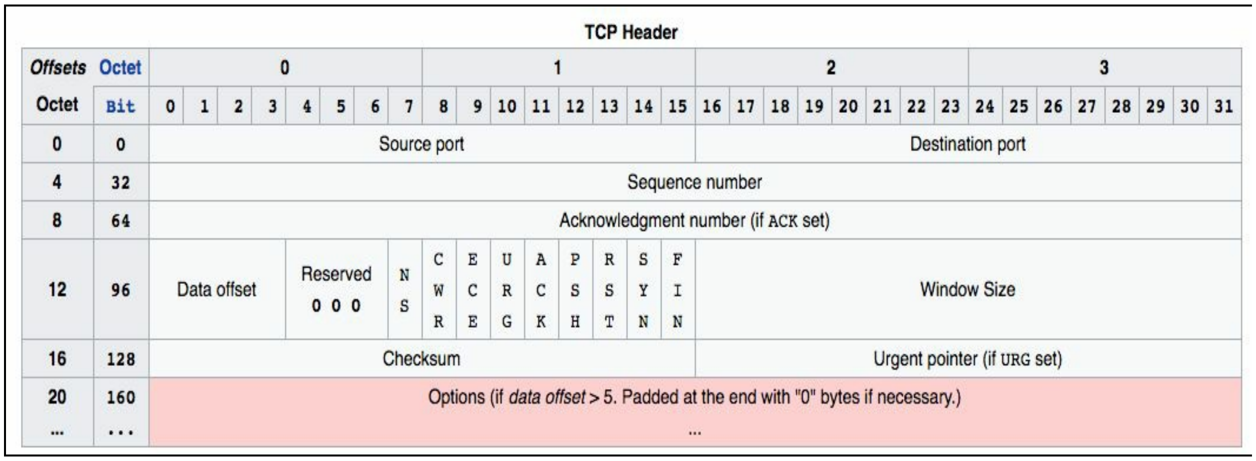
## Network protocol suites

In the early days of computer networking, the protocols were proprietary and closely controlled by the company who designed the connection method. If you are using Novell's IPX/SPX protocol in your hosts, you will not able to communicate with Apple's **AppleTalk** hosts and vice versa. These proprietary protocol suites generally have analogous layers to the OSI reference model and follow the client-server communication method. They generally work great in **Local Area Networks** (**LAN**) that are closed without the need to communicate with the outside world. When the traffic do need to move beyond local LAN, typically an internetwork device, such as a router, is used to translate from one protocol to another such as between AppleTalk to IP. The translation is usually not perfect, but since most of the communication happens within the LAN, it is okay.

However, as the need for internetwork communication rises, the need for standardizing the network protocol suites becomes greater. These proprietary protocols eventually gave way to the standardized protocol suites of TCP, UDP, and IP that greatly enhanced the ability from one network to talk to another. The internet, the greatest network of them all, relies on these protocols to function properly. In the next few sections, we will take a look at each of the protocol suites.

# The Transmission Control Protocol (TCP)

The **Transmission Control Protocol** (**TCP**) is one of the main protocols used on the internet today. If you have opened a web page or have sent an e-mail, you have come across the TCP protocol. The protocol sits at layer 4 of the OSI model, and it is responsible for delivering the data segment between two nodes in a reliable and error-checked manner. The TCP consists of a 128-bit header consists of, among others, source and destination port, sequence number, acknowledgment number, control flags, and checksum:



TCP Header

| *Offsets* | Octet | 0 | | | | | | | | 1 | | | | | | | | 2 | | | | | | | | 3 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Octet | Bit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 0 | 0 | Source port | | | | | | | | | | | | | | | | Destination port | | | | | | | | | | | | | | | |
| 4 | 32 | Sequence number | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 8 | 64 | Acknowledgment number (if ACK set) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 12 | 96 | Data offset | | | | Reserved 0 0 0 | | | NS | CWR | ECE | URG | ACK | PSH | RST | SYN | FIN | Window Size | | | | | | | | | | | | | | | |
| 16 | 128 | Checksum | | | | | | | | | | | | | | | | Urgent pointer (if URG set) | | | | | | | | | | | | | | | |
| 20 | 160 | Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| ... | ... | ... | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

TCP Header (source:https://en.wikipedia.org/wiki/Transmission_Control_Protocol)

## Functions and Characteristics of TCP

TCP uses datagram sockets or ports to establish a host-to-host communication. The standard body called **Internet Assigned Numbers Authority** (**IANA**) designates well-known ports to indicate certain services, such as port `80` for HTTP (web) and port `25` for SMTP (mail). The server in the client-server model typically listens on one of these well-known ports in order to receive communication requests from the client. The TCP connection is managed by the operating system by the socket that represents the local endpoint for connection.

The protocol operation consist of a state machine, where the machine needs to keep track of when it is listening for incoming connection, during communication session, as well as releasing resources once the connection is closed. Each TCP connection goes through a series of states such as `Listen`, `SYN-SENT`, `SYN-RECEIVED`, `ESTABLISHED`, `FIN-WAIT`, `CLOSE-WAIT`, `CLOSING`, `LAST-ACK`, `TIME-WAIT`, and `CLOSED`.

## TCP messages and data transfer

The biggest difference between TCP and **User Datagram Protocol** (**UDP**), which is its close cousin at the same layer, is that it transmits data in an ordered and reliable fashion. The fact that the operation guarantees delivery often referred to TCP as a connection-oriented protocol. It does this by first establishing a three-way handshake to synchronize the sequence number between the transmitter and the receiver, `SYN`, `SYN-ACK`, and `ACK`.

The acknowledgement is used to keep track of subsequent segments in the conversation. Finally at the end of the conversation, one side will send a `FIN` message, the other side will `ACK` the `FIN` message as well as send a `FIN` message of its own. The `FIN` initiator will then `ACK` the `FIN` message that it received.

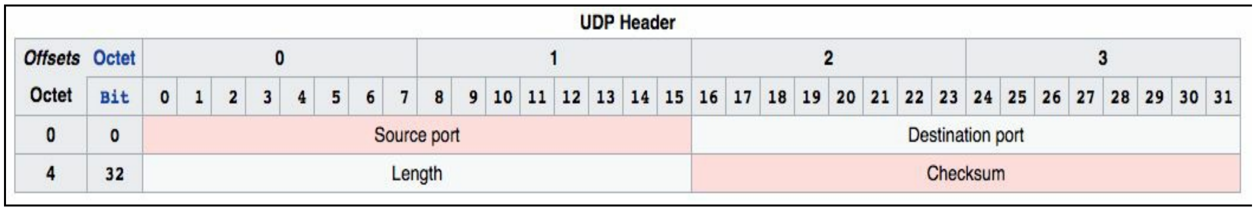
As many of us who have troubleshot a TCP connection can tell you, the operation can get quite complex. One can certainly appreciate that most of the time, the operation just happens silently in the background.

A whole book can be written about the TCP protocol; in fact, many excellent books have been written on the protocol.

*As this section is a quick overview, if interested,* The TCP/IP Guide *is an excellent free source to dig deeper into the subject.*

# User Datagram Protocol (UDP)

**UDP** is also a core member of the internet protocol suite. Like TCP, it is on layer 4 of the OSI model that is responsible for delivering data segments between the application and the IP layer. Unlike TCP, their header is only 32-bit that only consists of source and destination port, length, and checksum. The lightweight header makes it ideal for the application to prefer a faster data delivery without setting up the session between two hosts or needing a reliable delivery of data. Perhaps it is hard to imagine in today's fast internet connections, but the extra header made a big difference in the speed of transmission in the early days of X.21 and frame relay links. Although as important as the speed save, not having to maintain various states such as TCP also saves compute resources on the two endpoints.



UDP Header

| Offsets | Octet | 0 | | | | | | | | 1 | | | | | | | | 2 | | | | | | | | 3 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Octet | Bit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 0 | 0 | Source port | | | | | | | | | | | | | | | | Destination port | | | | | | | | | | | | | | | |
| 4 | 32 | Length | | | | | | | | | | | | | | | | Checksum | | | | | | | | | | | | | | | |

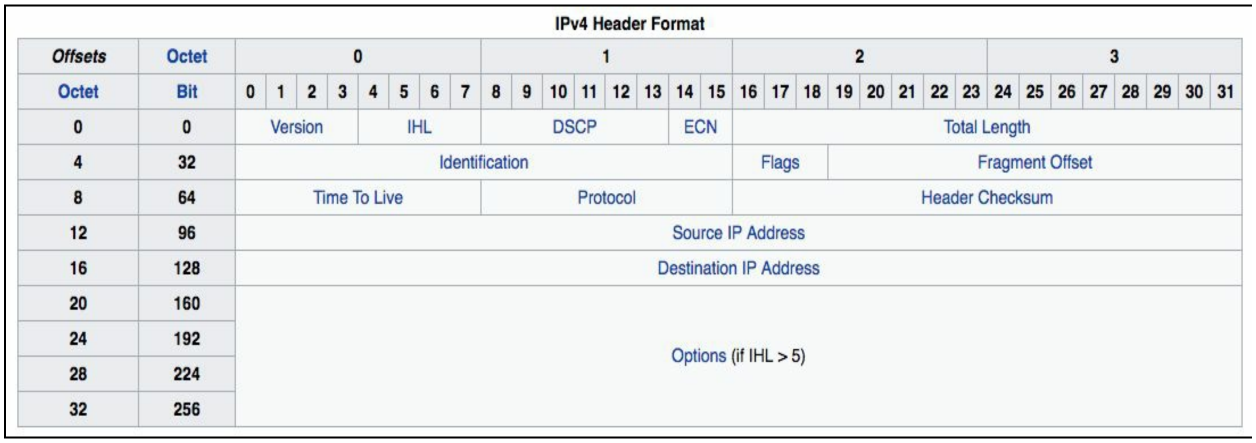
UDP Header (source: https://en.wikipedia.org/wiki/User_Datagram_Protocol)

You might now wonder why UDP was ever used at all in the modern age; given the lack of reliable transmission, wouldn't we want all the connections to be reliable and error free? If one thinks about some of the multimedia video streaming or Skype call, those applications will benefit from a lighter header when the application just wants to deliver the datagram as fast as possible. You can also consider the DNS lookup process. When the address you type in on the browser is translated into a computer understandable address, the user will benefit from the lightweight process since this has to happen "before" even the first bit of information is delivered to you from your favorite website.

Again, this section does not do justice to the topic of UDP, and the reader is encouraged to explore the topic through various resources if he/she is interested in learning more about UDP.

# The Internet Protocol (IP)

As network engineers would tell you, network engineers "live" at the IP layer, which is layer 3 on the OSI model. **IP** has the job of addressing and routing between end nodes, among others. The addressing of IP is probably its most important job. The address space is divided into two parts: the network and the host portion. The subnet mask indicated which portion in the address consist of the network and which is the host. Both IPv4, and later, IPv6 expresses the address in the dotted notation, for example `192.168.0.1`. The subnet mask can either be in a dotted notation (`255.255.255.0`) or use a forward slash to express the number of bits that should be considered in the network bit (/24).

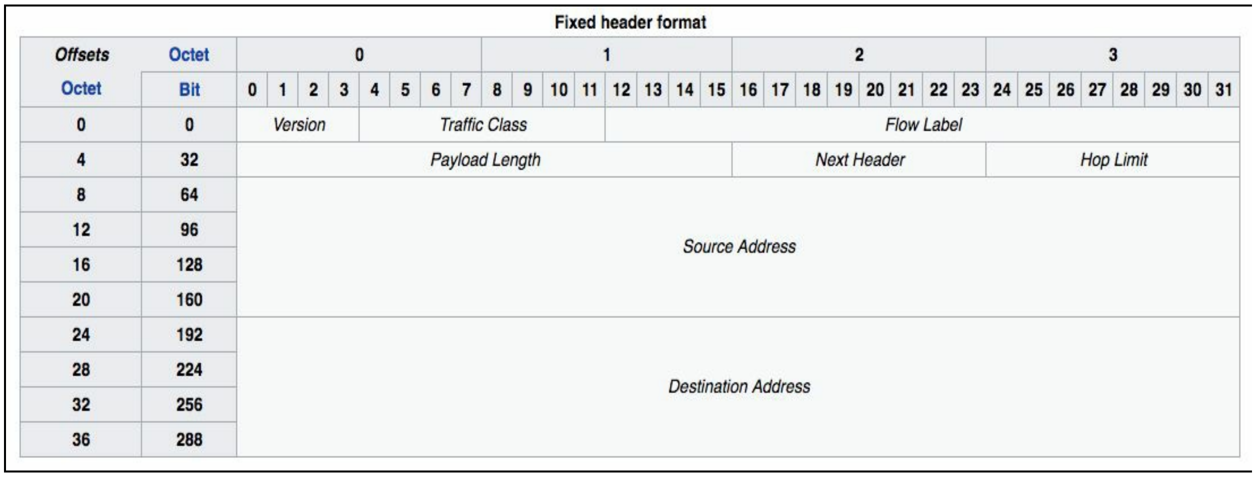


IPv4 Header Format

| Offsets | Octet | 0 | | | | | | | | 1 | | | | | | | | 2 | | | | | | | | 3 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Octet | Bit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 0 | 0 | Version | | | | IHL | | | | DSCP | | | | | | ECN | | Total Length | | | | | | | | | | | | | | | |
| 4 | 32 | Identification | | | | | | | | | | | | | | | | Flags | | | Fragment Offset | | | | | | | | | | | | |
| 8 | 64 | Time To Live | | | | | | | | Protocol | | | | | | | | Header Checksum | | | | | | | | | | | | | | | |
| 12 | 96 | Source IP Address | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 16 | 128 | Destination IP Address | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 20 | 160 | Options (if IHL > 5) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 24 | 192 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 28 | 224 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 32 | 256 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

IPv4 Header (source: https://en.wikipedia.org/wiki/IPv4)

The IPv6 header, next generation of IP header of IPv4, has a fixed portion and various extension headers.



Fixed header format

| Offsets | Octet | 0 | | | | | | | | 1 | | | | | | | | 2 | | | | | | | | 3 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Octet | Bit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 0 | 0 | *Version* | | | | *Traffic Class* | | | | | | | | *Flow Label* | | | | | | | | | | | | | | | | | | | |
| 4 | 32 | *Payload Length* | | | | | | | | | | | | | | | | *Next Header* | | | | | | | | *Hop Limit* | | | | | | | |
| 8 | 64 | *Source Address* | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 12 | 96 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 16 | 128 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 20 | 160 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 24 | 192 | *Destination Address* | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 28 | 224 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 32 | 256 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 36 | 288 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

IPv6 Fixed Header (source: https://en.wikipedia.org/wiki/IPv6_packet)

The **Next Header** field in the fixed header section can indicate an extension header to be followed that carries additional information. The extension headers can include routing and fragment information. As much as the protocol designer would like to move from IPv4 to IPv6, the internet today is still pretty much addressed with IPv4.

## The IP NAT and security

**Network Address Translation** (**NAT**) is typically used for translating a range of private IPv4 addresses to publicly routable IPv4 addresses. But it can also mean a translation between IPv4 to IPv6, such as at a carrier edge when they use IPv6 inside of the network that needs to be translated to IPv4, when the packet leaves the network. Sometimes, NAT6to6 is used as well for security reasons.

Security is a continuous process that integrates all the aspects of networking, including automation and Python. This book aims at using Python to help you manage the network; security will be addressed as part of the chapter such as using SSHv2 over telnet. We will also look at how we can use Python and other tool chains to gain visibility in the network.

## IP routing concepts

In my opinion, IP routing is about having the intermediate devices between the two endpoint transmit the packets between them based the IP header. For every communication on the internet, the packet will traverse through various intermediate devices. As mentioned, the intermediate devices consist of routers, switches, optical gears, and various other gears that do not examine beyond the network and transport layer. In a road trip analogy, you might travel in the United States from the city of San Diego in California to the city of Seattle in Washington. The IP source address is analogous to San Diego and the destination IP address can be thought of as Seattle. On your road trip, you will stop by many different intermediate spots, such as Los Angeles, San Francisco, and Portland; these can be thought of as the routers and switches between the source and destination.

Why was this important? In a way, this book is about managing and optimizing these intermediate devices. In the age of mega datacenters that span sizes of multiple American football fields, the need for efficient, agile, cost effective way to manage the network becomes a major point of competitive advantage for companies. In the future chapters, we will dive into how we can use Python programming to effectively manage the network.

# Python language overview

In a nutshell, this book is about making our lives easier with Python. But what is Python and why is it the language of choice by many DevOps engineers? In the words of the Python Foundation Executive Summary (https://www.python.org/doc/essays/blurb/):

*"Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level, built-in data structure, combined with dynamic typing and dynamic binding, makes it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy-to-learn syntax emphasizes readability and therefore reduces the cost of program maintenance."*

If you are somewhat new to programming, the object-oriented, dynamic semantics probably does not mean much to you. But I think we can all agree on Rapid application development, simple, and easy-to-learn syntax sounds like a good thing. Python as an interpreted language means there is no compilation process required, so the speed of write, test, and edit of the program process is greatly increased. For simple scripts, if your script fails, a few print statement is usually all you need to debug what was going on. Using the interpreter also means that Python is easily ported to different type of operating systems and Python program written on Windows can be used on Linux and Mac.

The object-oriented nature encourages code reuse by breaking into simple reusable form such modules and packages. In fact, all Python files are modules that can be reused or imported in another Python program. This makes it easy to share programs among engineers and encourage code reuse. Python also has a batteries included mantra, which means that for common tasks, you need not download any additional code. In order to achieve this without the code being too bloated, a set of standard libraries is installed when you install the Python interpreter. For common tasks such as regular expression, mathematics functions, and JSON decoding, all you need is the `import` statement, and the interpreter will move those functions into your program. This is what I would consider one of the killer features of the Python language.

Lastly, the fact that Python code can start in a relatively small-sized script with a few lines of code and grow into a fully production system is very handy for network engineers. As many of us know, the network typically grows organically without a master plan. A language that can grow with your network in size is invaluable. You might be surprised to see a language that was deemed as scripting language by many, which was being used for fully production systems (Organizations using Python, https://wiki.python.org/moin/OrganizationsUsingPython).

If you have ever worked in an environment where you have to switch between working on different vendor platforms, such as Cisco IOS and Juniper Junos, you know how painful it is to switch between syntax and usage when trying to achieve the same task. With Python being flexible enough for large and small programs, there is no such context switching, because it is just Python.

For the rest of the chapter, we will take a high-level tour of the Python language for a bit of a refresher. If you are already familiar with the basics, feel free to quickly scan through it or skip the rest of the chapter.

## Python versions

At the time of writing this book in early 2017, Python is going through a transition period of moving from Python version 2 to Python version 3. Unfortunately Python 3 is not backward compatible with Python 2. When I say transition, keep in mind that Python 3 was released back in 2008, over nine years ago with active development with the most recent release of 3.6. The latest Python 2.x release, 2.7, was released over six years ago in mid 2010. Fortunately, both version can coexist on the same machine. Personally, I use Python 2 as my default interpreter when I type in Python at the Command Prompt, and I use Python 3 when I need to use Python 3. More information is given in the next section about invoking Python interpreter, but here is an example of invoking Python 2 and Python 3 on the Ubuntu Linux machine:

```
echou@pythonicNeteng:~$ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>> exit()
```

```
echou@pythonicNeteng:~$ python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for
more information.
>>> exit()
```

With the 2.7 release being end of life on extended support only for security updates, most Python frameworks are now supporting Python 3. Python 3 also has lots of good features such as asynchronous I/O that can be taken advantage of when we need to optimize our code. This book will use Python 3 for its code examples.

If the particular library or the framework does not support Python 3, such as Ansible (they are actively working on porting to Python 3), it will be pointed out, so you can use Python 2 instead.

## Operating system

As mentioned, Python is cross platform. Python programs can be run on Windows, Mac, and Linux. In reality, certain care needs to be taken when you need to ensure cross-platform compatibility, such as taking care of the subtle difference backslashes in Windows filenames. Since this book is for DevOps, systems, and network engineers, Linux is the preferred platform for the intended audience, especially in production. The code in this book will be tested on the Linux Ubuntu 16.06 LTS machine. I will also try my best to make sure the code runs the same on Windows and the Mac platform.

If you are interested in the OS details, they are as follows:

```
echou@pythonicNeteng:~$ uname -a
Linux pythonicNeteng 4.4.0-31-generic #50-Ubuntu SMP
Wed Jul 13 00:07:12 UTC 2016 x86_64 x86_64 x86_64
GNU/Linux
echou@pythonicNeteng:~$
```

## Running a Python program

Python programs are executed by an interpreter, which means the code is fed through this interpreter to be executed by the underlying operating system, and results are displayed . There are several different implementation of the interpreter by the Python development community, such as IronPython and Jython. In this book, we will refer to the most common Python interpreter, CPython, which is in use today.

One way you can use Python is by taking the advantage of the interactive prompt. This is useful when you want to quickly test a piece of Python code or concept without writing a whole program. This is typically done by simply typing in the `Python` keyword:

```
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for
more information.
>>> print("hello world")
hello world
>>>
```

*Notice the parentheses around the print statement. In Python 3, the print statement is a function; therefore, it requires the parentheses. In Python 2, you can omit the parentheses.*

The interactive mode is one of the Python's most useful features. In the interactive shell, you can type any valid statement or sequence of statements and immediately get a result back. I typically use this to explore around a feature or library that I am not familiar with. Talk about instant gratification!

*On Windows, if you do not get a Python shell prompt back, you might not have the program in your system search path. The latest Windows Python installation program provides a check box for adding Python to your system path; make sure that was checked. Or you can add the program in the path manually by going to Environment Settings.*

A more common way to run the Python program, however, is to save your Python file and run via the interpreter after. This will save you from typing the same statements over and over again such as in the interactive shell. Python files are just regular text files that are typically saved with the `.py` extension. In the *Nix world, you can also add the **shebang** (`#!`) line on top to specify the interpreter that is used to run the file. The `#` character can be used to specify comments that will not be executed by the interpreter. The following file, `helloworld.py`, has the following statements:

```
# This is a comment
print("hello world")
```

This can be executed as follows:

```
echou@pythonicNeteng:~/Master_Python_Networking/
Chapter1$ python helloworld.py
hello world
echou@pythonicNeteng:~/Master_Python_Networking/
Chapter1$
```

# Python built-In types

Python has several standard types built into the interpreter:

- `None`: The `Null` object
- Numerics: `int`, `long`, `float`, `complex`, and `bool` (The subclass of `int` with `True` or `False` value)
- Sequences: `str`, list, tuple, and range
- Mappings: `dict`
- Sets: `set` and `frozenset`

## The None type

The `None` type denotes an object with no value. This is returned in functions that do not explicitly return anything. The `None` type is also used in function arguments to error out if the caller does not pass in an actual value.

## Numerics

Python numeric objects are basically numbers. With the exception of Boolean, the numeric types of `int`, `long`, `float`, and `complex` are all signed, meaning they can be positive or negative. Boolean is a subclass of the integer that can be one of two values: `1` for `True`, and `0` for `False`. The rest of the numeric types are differentiated by how precise they can represent the number; for example, int are whole numbers with a limited range while long are whole numbers with unlimited range. Float are numbers using the double-precision representation (64-bit) on the machine.

## Sequences

Sequences are ordered sets of objects with an index of non-negative integers. In this and the next few sections, we will use the interactive interpreter to illustrate the different types. Please feel free to type along on your own computer.

Sometimes it surprises people that `string` is actually a sequence type. But if you look closely, strings are series of characters put together. Strings are enclosed by either single, double, or triple quotes. Note in the following examples, the quotes have to match, and triple quotes allow the string to span different lines:

```
>>> a = "networking is fun"
>>> b = 'DevOps is fun too'
>>> c = """what about coding?
... super fun!"""
>>>
```

The other two commonly used sequence types are lists and tuples. Lists are the sequence of arbitrary objects. Lists can be created by enclosing the objects in square brackets. Just like string, lists are indexed by non-zero integers that start at zero. The values of lists are retrieved by referencing the index number:

```
>>> vendors = ["Cisco", "Arista", "Juniper"]
>>> vendors[0]
'Cisco'
>>> vendors[1]
'Arista'
>>> vendors[2]
'Juniper'
```

Tuples are similar to lists, created by enclosing the values in parentheses. Like lists, the values in the tuple are retrieved by referencing its index number. Unlike list, the values cannot be modified after creation:

```
>>> datacenters = ("SJC1", "LAX1", "SFO1")
>>> datacenters[0]
'SJC1'
>>> datacenters[1]
```

```
'LAX1'
>>> datacenters[2]
'SFO1'
```

Some operations are common to all sequence types, such as returning an element by index as well as slicing:

```
>>> a
'networking is fun'
>>> a[1]
'e'
>>> vendors
['Cisco', 'Arista', 'Juniper']
>>> vendors[1]
'Arista'
>>> datacenters
('SJC1', 'LAX1', 'SFO1')
>>> datacenters[1]
'LAX1'
>>>
>>> a[0:2]
'ne'
>>> vendors[0:2]
['Cisco', 'Arista']
>>> datacenters[0:2]
('SJC1', 'LAX1')
>>>
```

*Remember that index starts at `0`. Therefore, the index of `1` is actually the second element in the sequence.*

There are also common functions that can be applied to sequence types, such as checking the number of elements and minimum and maximum values:

```
>>> len(a)
17
>>> len(vendors)
3
>>> len(datacenters)
3
>>>
>>> b = [1, 2, 3, 4, 5]
>>> min(b)
1
>>> max(b)
5
```

It would come as no surprise that there are various methods that apply only to strings. It is worth noting that these methods do not modify the underlying string data itself and always return a new string. If you want to use the new value, you would need to catch the return value and assign it to a different variable:

```
>>> a
'networking is fun'
>>> a.capitalize()
'Networking is fun'
>>> a.upper()
'NETWORKING IS FUN'
>>> a
'networking is fun'
>>> b = a.upper()
>>> b
'NETWORKING IS FUN'
>>> a.split()
['networking', 'is', 'fun']
>>> a
'networking is fun'
>>> b = a.split()
>>> b
['networking', 'is', 'fun']
>>>
```

Here are some of the common methods for a list. This list is a very useful structure in terms of putting multiple items in and iterating through them. For example, we can make a list of datacenter spine switches and apply the same access list to all of them by iterating through them one by one. Since a list's value can be modified after creation (unlike tuple), we can also expand and contrast the existing list as we move along the program:

```
>>> routers = ['r1', 'r2', 'r3', 'r4', 'r5']
>>> routers.append('r6')
>>> routers
['r1', 'r2', 'r3', 'r4', 'r5', 'r6']
>>> routers.insert(2, 'r100')
>>> routers
['r1', 'r2', 'r100', 'r3', 'r4', 'r5', 'r6']
>>> routers.pop(1)
'r2'
>>> routers
['r1', 'r100', 'r3', 'r4', 'r5', 'r6']
```

## Mapping

Python provides one mapping type called **dictionary**. Dictionary is what I think of as a poor man's database, because it contains objects that can be indexed by keys. This is often referred to as the associated array or hashing table in other languages. If you have used any of the dictionary-like objects in other languages, you will know this is a powerful type, because you can refer to the object by a readable key. This key will make more sense for the poor guy who is trying to maintain the code, probably a few months after you wrote the code at 2 am in the morning. The object can also be another data type, such as a list. You can create a dictionary with curly braces:

```
>>> datacenter1 = {'spines': ['r1', 'r2', 'r3', 'r4']}
>>> datacenter1['leafs'] = ['l1', 'l2', 'l3', 'l4']
>>> datacenter1
{'leafs': ['l1', 'l2', 'l3', 'l4'], 'spines': ['r1',
'r2', 'r3', 'r4']}
>>> datacenter1['spines']
['r1', 'r2', 'r3', 'r4']
>>> datacenter1['leafs']
['l1', 'l2', 'l3', 'l4']
```

## Sets

A **set** is used to contain an unordered collection of objects. Unlike lists and tuples, sets are unordered and cannot be indexed by numbers. But there is one character that makes sets standout and useful: the elements of a set are never duplicated. Imagine if you have a list of IPs that you need to put in an access list of. The only problem in this list of IPs is that they are full of duplicates. Now, think about how you would use a loop to sort it out for the unique items, the set built-in type would allow you to eliminate the duplicate entries with just one line of code. To be honest, I do not use set that much, but when I need it, I am always very thankful this exists. Once the set or sets are created, they can be compared with each other ones using the union, intersection, and differences:

```
>>> a = "hello"
>>> set(a)
{'h', 'l', 'o', 'e'}
>>> b = set([1, 1, 2, 2, 3, 3, 4, 4])
>>> b
{1, 2, 3, 4}
>>> b.add(5)
>>> b
{1, 2, 3, 4, 5}
>>> b.update(['a', 'a', 'b', 'b'])
>>> b
{1, 2, 3, 4, 5, 'b', 'a'}
>>> a = set([1, 2, 3, 4, 5])
>>> b = set([4, 5, 6, 7, 8])
>>> a.intersection(b)
{4, 5}
>>> a.union(b)
{1, 2, 3, 4, 5, 6, 7, 8}
>>> 1 *
{1, 2, 3}
>>>
```

# Python operators

Python has the some numeric operators that you would expect; note that the truncating division, (`//`, also known as **floor division**) truncates the result to an integer and a floating point and return the integer value. The integer value is returned. The modulo (`%`) operator returns the remainder value in the division:

```
>>> 1 + 2
3
>>> 2 - 1
1
>>> 1 * 5
5
>>> 5 / 1
5.0
>>> 5 // 2
2
>>> 5 % 2
1
```

There are also comparison operators:

```
>>> a = 1
>>> b = 2
>>> a == b
False
>>> a > b
False
>>> a < b
True
>>> a <= b
True
```

We will also see two of the common membership operators to see whether an object is in a sequence type:

```
>>> a = 'hello world'
>>> 'h' in a
True
>>> 'z' in a
False
>>> 'h' not in a
False
>>> 'z' not in a
True
```

# Python control flow tools

The `if`, `else`, and `elif` statements control conditional code execution. As one would expect, the format of the conditional statement is as follows:

```
if expression:
  do something
elif expression:
  do something if the expression meets
elif expression:
  do something if the expression meets
...
else:
  statement
```

Here is a simple example:

```
>>> a = 10
>>> if a > 1:
...   print("a is larger than 1")
... elif a < 1:
...   print("a is smaller than 1")
... else:
...   print("a is equal to 1")
...
a is larger than 1
>>>
```

The while loop will continue to execute until the condition is false, so be careful with this one if you don't want to continue to execute:

```
while expression:
  do something
```

```
>>> a = 10
>>> b = 1
>>> while b < a:
...   print(b)
...   b += 1
...
1
2
3
4
5
6
7
8
9
```

The for loop works with any object that supports iteration; this means all the built-in sequence types such as lists, tuples, and strings can be used in a for loop. The letter `i` in the following for loop is an iterating variable, so you can typically pick something that make sense within the context of your code:

```
for i in sequence:
  do something
```

```
>>> a = [100, 200, 300, 400]
>>> for number in a:
...   print(number)
...
100
200
300
400
```

You can also make your own object that supports the iterator protocol and be able to use the for loop for this object:

*Constructing such an object is outside the scope of this chapter, but it is a useful knowledge to have; you can read more about it https://docs.python.org/3/c-api/iter.html.*

## Python functions

Most of the time when you find yourself reusing some pieces of code, you should break up the code into a self-contained chunk as functions. This practice allows for better modularity, is easier to maintain, and allows for code reuse. Python functions are defined using the def keyword with the function name, followed by the function parameters. The body of the function consists of Python statements that are to be executed. At the end of the function, you can choose to return a value to the function caller, or by default, it will return the `None` object if you do not specify a return value:

```
def name(parameter1, parameter2):
  statements
  return value
```

We will see a lot more examples of function in the following chapters, so here is a quick example:

```
>>> def subtract(a, b):
...   c = a - b
...   return c
...
>>> result = subtract(10, 5)
>>> result
5
>>>
```

# Python classes

Python is an **Object-Oriented Programming** (**OOP**) language. The way Python creates objects are with the `class` keyword. A Python object is most commonly a collection of functions (methods), variables, and attributes (properties). Once a class is defined, you can create instances of such a class. The class serves as the blueprint of the subsequent instances.

The topic of object-oriented programming is outside the scope of this chapter, so here is a simple example of a router object definition:

```
>>> class router(object):
...   def __init__(self, name, interface_number,
 vendor):
...     self.name = name
...     self.interface_number = interface_number
...     self.vendor = vendor
...
>>>
```

Once defined, you are able to create as many instances of that class as you'd like:

```
>>> r1 = router("SFO1-R1", 64, "Cisco")
>>> r1.name
'SFO1-R1'
>>> r1.interface_number
64
>>> r1.vendor
'Cisco'
>>>
>>> r2 = router("LAX-R2", 32, "Juniper")
>>> r2.name
'LAX-R2'
>>> r2.interface_number
32
>>> r2.vendor
'Juniper'
>>>
```

Of course, there is a lot more to Python objects and OOP. We will see more examples in the future chapters.

# Python modules and packages

Any Python source file can be used as a module, and any functions and classes you define in that source file can be re-used. To load the code, the file referencing the module needs to use the import keyword. Three things happen when the file is imported:

1. The file creates a new namespace for the objects defined in the source file.
2. The caller executes all the code contained in the module.
3. The file creates a name within the caller that refers to the module being imported. The name matches the name of the module.

Remember the `subtract()` function that you defined using the interactive shell? To reuse the function, we can put it into a file named `subtract.py`:

```
def subtract(a, b):
  c = a - b
  return c
```

In a file within the same directory of `subtract.py`, you can start the Python interpreter and import this function:

```
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>> import subtract
>>> result = subtract.subtract(10, 5)
>>> result
5
```

This works because, by default, Python will first search for the current directory for the available modules. If you are in a different directory, you can manually add a search path location using the `sys` module with `sys.path`. Remember the standard library that we mentioned a while back? You guessed it, those are just Python files being used as modules.

Packages allow a collection of modules to be grouped together. This further organizes the Python modules into a more namespace protection to further reusability. A package is defined by creating a directory with a name you want to use as the namespace, then yo can place the module source file under that directory. In order for Python to know it as a Python-package, just create a `__init__.py` file in this directory. In the same example as the `subtract.py file`, if you were to create a directory called `math_stuff` and create `a __init__.py` file:

```
echou@pythonicNeteng:~/Master_Python_Networking/
Chapter1$ mkdir math_stuff
echou@pythonicNeteng:~/Master_Python_Networking/
Chapter1$ touch math_stuff/__init__.py
echou@pythonicNeteng:~/Master_Python_Networking/
Chapter1$ tree .
.
├── helloworld.py
└── math_stuff
  ├── __init__.py
  └── subtract.py

1 directory, 3 files
echou@pythonicNeteng:~/Master_Python_Networking/
Chapter1$
```

The way you will now refer to the module will need to include the package name:

```
>>> from math_stuff.subtract import subtract
>>> result = subtract(10, 5)
>>> result
5
>>>
```

As you can see, modules and packages are great ways to organize large code files and make sharing Python code a lot easier.

## Summary

In this chapter, we covered the OSI model and reviewed the network protocol suites, such as TCP, UDP, and IP. We then did a quick review of the Python language, including built-in types, operators, control flows, functions, classes, modules, and packages.

In the next chapter, we will start to look at using Python to programmatically interact with the network equipment.

# Low-Level Network Device Interactions

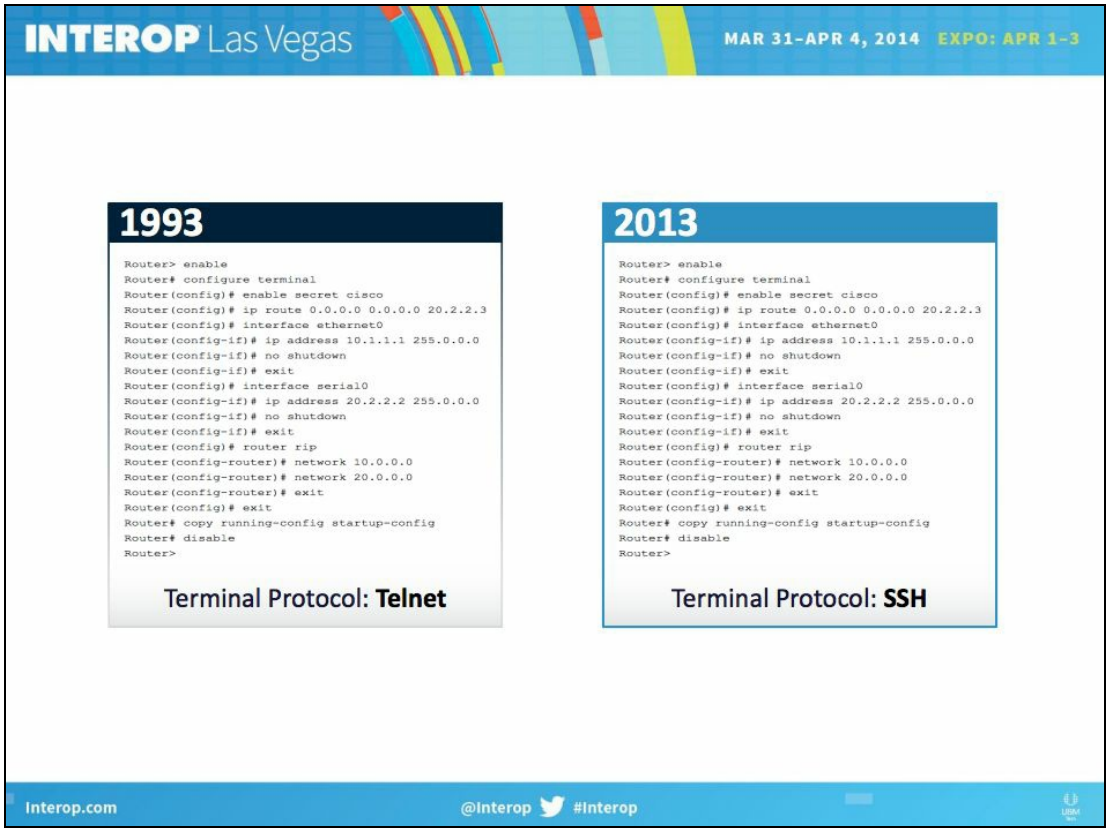
In Chapter 1, *Review of TCP/IP Protocol Suite and Python Language,Â* we looked at the theories and specifications behind network communication protocols, and we took a quick tour of the Python language. In this chapter, we will start to dive deeper into the management of these network devices. In particular, we will examine the different ways in which we can use Python to programmatically communicate with legacyÂ network routers and switches.

What do I mean by legacy network routers and switches? While it is hard to imagine any networking device coming out today without an **Application Program Interface** (**API**) for automating tasks, it is a known fact that many of the network devices deployed today do not have APIs. The only way to manage them is throughÂ **Command Line Interfaces** (**CLI**) using terminal programs, which originally were developed with a human engineer in mind. As the number of network devices increases, it becomes increasingly difficult to manually manage them one by one. Python has two great libraries that can help with these tasks, so this chapter will cover Pexpect and Paramiko. In this chapter, we will take a look at the following topics:Â

- The challenges of CLI
- Constructing a virtual lab
- The Python Pexpect library
- The Python Paramiko library
- The downsides of Pexpect and Paramiko

# The challenges of CLI

At the Interop expo in Las Vegas 2014, *BigSwitch Networks* CEO *Douglas Murray* displayed the following slide to illustrate what has changed in **Data Center Networking** (**DCN**) in 20 years between 1993 to 2013:



INTEROP Las Vegas
MAR 31-APR 4, 2014
EXPO: APR 1-3
Interop.com
@Interop #Interop

1993

```
Router> enable
Router# configure terminal
Router(config)# enable secret cisco
Router(config)# ip route 0.0.0.0 0.0.0.0 20.2.2.3
Router(config)# interface ethernet0
Router(config-if)# ip address 10.1.1.1 255.0.0.0
Router(config-if)# no shutdown
Router(config-if)# exit
Router(config)# interface serial0
Router(config-if)# ip address 20.2.2.2 255.0.0.0
Router(config-if)# no shutdown
Router(config-if)# exit
Router(config)# router rip
Router(config-router)# network 10.0.0.0
Router(config-router)# network 20.0.0.0
Router(config-router)# exit
Router(config)# exit
Router# copy running-config startup-config
Router# disable
Router>
```

Terminal Protocol: **Telnet**

2013

```
Router> enable
Router# configure terminal
Router(config)# enable secret cisco
Router(config)# ip route 0.0.0.0 0.0.0.0 20.2.2.3
Router(config)# interface ethernet0
Router(config-if)# ip address 10.1.1.1 255.0.0.0
Router(config-if)# no shutdown
Router(config-if)# exit
Router(config)# interface serial0
Router(config-if)# ip address 20.2.2.2 255.0.0.0
Router(config-if)# no shutdown
Router(config-if)# exit
Router(config)# router rip
Router(config-router)# network 10.0.0.0
Router(config-router)# network 20.0.0.0
Router(config-router)# exit
Router(config)# exit
Router# copy running-config startup-config
Router# disable
Router>
```

Terminal Protocol: **SSH**

Datacenter Networking Changes (source: https://www.bigswitch.com/sites/default/files/presentations/murraydouglasstartuphotseatpanel.pdf)

While he might be negatively biased toward the incumbent vendors when displaying this slide, the point is well taken. In his opinion, the only thing in managing routers and switches that has changed in 20 years was the protocol changing from Telnet to the more secured SSH. It is right around the same time that we start to see the consensus around the industry for the clear need to move away from manual, human-driven CLI toward an automatic, computer-centric automation API. Make no mistake, we still need to directly communicate with the device when making network designs, bringing up initial proof of concepts, and deploying the topology for the first time. However, once we have moved beyond the initial deployment, the need becomes to consistently make the same changes error-free, over and over again without being distracted or feeling tired. Sounds like an ideal job for computers and our favorite language, Python.

Referring back to the slide, the challenge, is the interaction between the router and the administrator. The router expects the administrator to enter a series of manual inputs that requires human interpretation. For example, you have to type in `enable` to get into a privileged mode, and upon receiving the returned prompt with `#` sign, you then need to type in `configure terminal` in order to go into the configuration mode. The same process can further be expanded into interface configuration mode and routing protocol configuration mode. This is in a sharp contrast to an computer-driven mindset. When computer wants to accomplish a single task, say put an IP on an interface, it wants to structurally give all the information to the router at once, and then it expects a single `yes` or `no` answer from the router to indicate success or failure of the task.

The solution, as implemented by both Pexpect and Paramiko, is to treat the interactive process as a child process and watch over the interaction between the process and the destination device. Based on the returned value, the parent process will decide the subsequent action, if any.

## Constructing a virtual lab

Before we dive into the packages, let's examine the options of putting together a lab for the benefit of learning. As the old saying goes, "*Practice Makes Perfect*": we need an isolated sandbox to  safely make mistakes, try out new ways of doing things, and repeat some of the steps to reinforce concepts that were not clear in the first try. It is easy enough to install Python and the necessary packages for the management host, but what about those routers and switches that we want to simulate?

To put together a network lab, we basically have two options, each with its advantages and disadvantages:

- **Physical device**: This option consists of physical devices that you can see and touch. If you are lucky enough, you might be able to put together a lab that is an exact replication of your production environment.
    - **Advantages**: It is an easy transition from lab to production, easier to understand by managers and fellow engineers who can look at touch the devices.
    - **Disadvantages**: It is relatively expensive, requires human capital to rack and stack, and is not very flexible once constructed.
- **Virtual devices**: These are emulation or simulation of the actual network devices. They are either provided by the vendors or by the open source community.
    - **Advantages**: They are easier to set up, are relatively cheap, and can make changes quickly.
    - **Disadvantages**: They are usually a scaled-down version of their physical counterpart, and sometimes there are feature gaps too.

Of course, the decision of virtual and physical lab is a personal decision derived from a trade-off between the cost, ease of implementation, and the risk of having a gap between lab and production.

In my opinion, as more and more vendors decides to produce virtual appliances, the virtual lab is the way to proceed in a learning environment. The feature gap is relatively small and specifically documented when the virtual instance is provided by the vendor. The cost is relatively small compared to buying physical devices and the time-to-built is quicker because they are usually just software programs. For this book, I will use a combination of physical and virtual devices for demonstration with a preference toward virtual devices. For the purpose of the book, the differences should be transparent. If there are any known differences between the virtual and physical devices pertaining to our objectives, I will make sure to list them out.

On the virtual lab front, besides images from various vendors, I am also using a program from Cisco called **Virtual Internet Routing Lab** (**VIRL**).

*I want to point out that the use of this program is entirely optional to the reader. But it is strongly recommended that the reader have some lab equipment to follow along with the examples in this book.*

## Cisco Virtual Internet Routing Lab (VIRL)

I remember when I first started to study for my **Cisco Certified Internetwork Expert** (**CCIE**) lab exam, I purchased some used Cisco equipment from eBay to study. Even at a discount, each router and switch are hundreds of US dollars, so to save money, I even purchased some really outdated Cisco routers from the 1980s (search for Cisco AGS routers in your favorite search engine for a good chuckle) that significantly lacked in feature and horsepower even for lab standards. As much as it makes an interesting conversation with family members when I turned them on (they were really loud), putting the physical devices together was not fun. They were heavy and clunky, a pain to connect all the cables, and to introduce link failure, I would literally unplug a cable.

Fast forward a few years, **Dynamip** was created and I fell in love with how easy it was to create network scenarios. All you need is the iOS images from Cisco, a few carefully constructed topology file, and you can easily construct a virtual network that you can test your knowledge on. I had a whole folder of network topologies, presaved configurations, and different version of images, as called for by the scenario. The addition of a GNS3 frontend gives the whole setup a beautiful GUI facelift. Now, you can just click and drop your links and devices; you can even just print out the network topology for your manager right out of the GNS3 design panel.

In 2015, the Cisco community decided to fill this need by releasing the Cisco VIRL. If you have hardware machines that meet the hardware requirements and you are willing to pay for the required annual license, this is my preferred method of developing and try out many of the Python code both for this book and my own production use.

*As of January 1, 2017, only the personal edition 20-Node license is available for purchase for USD $199.99 per year.*

Even at a monetary cost, in my opinion, the VIRL platform offers a few advantages over other alternatives:

- **An ease of use**: All the images for IOSv, IOS-XRv, CSR100v, NX-OSv, ASAv are included in the single download.
- **Official (kind of)**: Although the support is community driven, it is an internally widely used tool at Cisco. It is also the popular kid on the bock that increases bug fix, documentation, and tribal knowledge.
- **The cloud migration path**: The project offers logical migration path when your emulation grows out of the hardware power you have, such as Cisco dCloud (https://dcloud.cisco.com/), VIRL on Packet (http://virl.cisco.com/cloud/), and Cisco DevNet (https://developer.cisco.com/).
- **The link and control-plane simulation**: The tool can simulate latency and jitter and packet-loss on a per-link basis for real-world link characteristics. There is also control-plane traffic generator for the external route injection.
- **Others**: The tool also offers some nice features such as VM Maestro topology design and simulation control, AutoNetkit for automatic config generation, and the user workspace management if the server is shared.

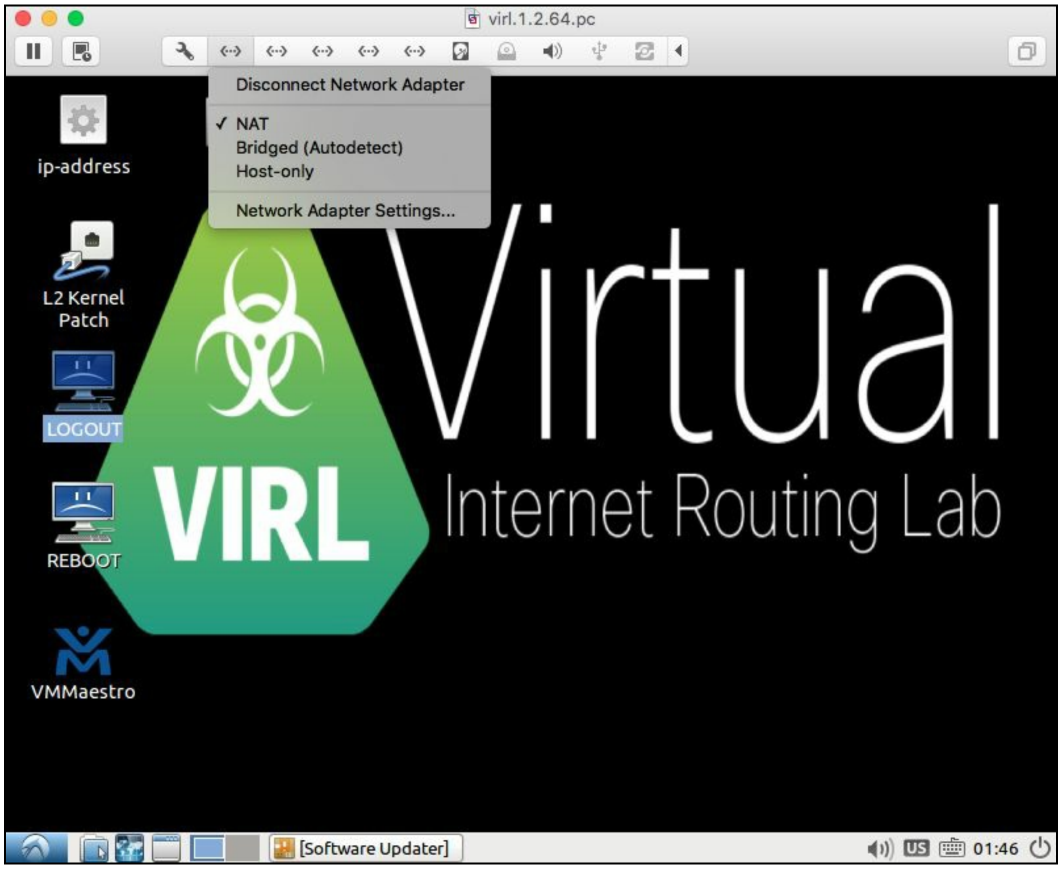
We will not use all of the features in this book. But since this is a relatively new tool that is worth your consideration, if you do decide this is the tool you would like to use as well, I want to offer some of the setups I used.

*Again, I want to stress the importance of having a lab, but it does not need to be Cisco VIRL, at least for this book.*

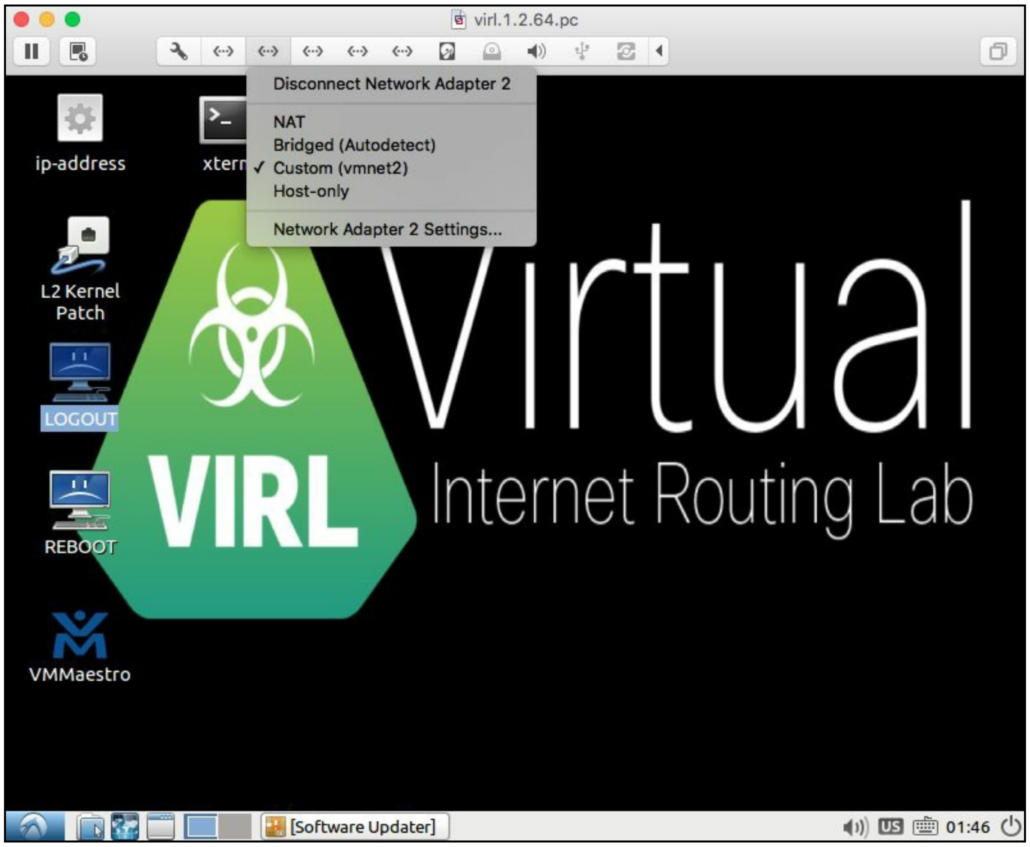
# VIRL tips

The VIRL website offers lots of great instruction and documentation. I also find the community generally offers quick and accurate help. I will not repeat the information already offered in those two places; however, here are some of the setups I use for the code in this book:

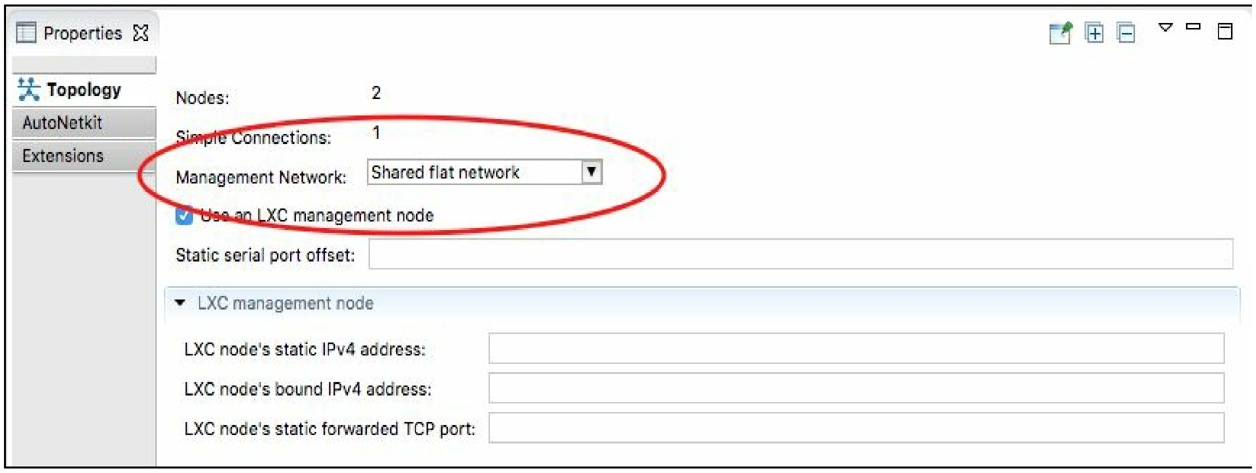
- VIRL uses two virtual Ethernet interfaces for connections. The first interface is set up as NAT for the host machine's Internet connection, and the second is used for local management interface connectivity (VMWare Fusion VMnet2 in the following example). I use a virtual machine with a similar network setup in order to run my Python code, with first Ethernet used for Internet and the second Ethernet connected to Vmnet2:



VMnet2 is a custom network created to connect the Ubuntu host with the VIRL virtual machine:

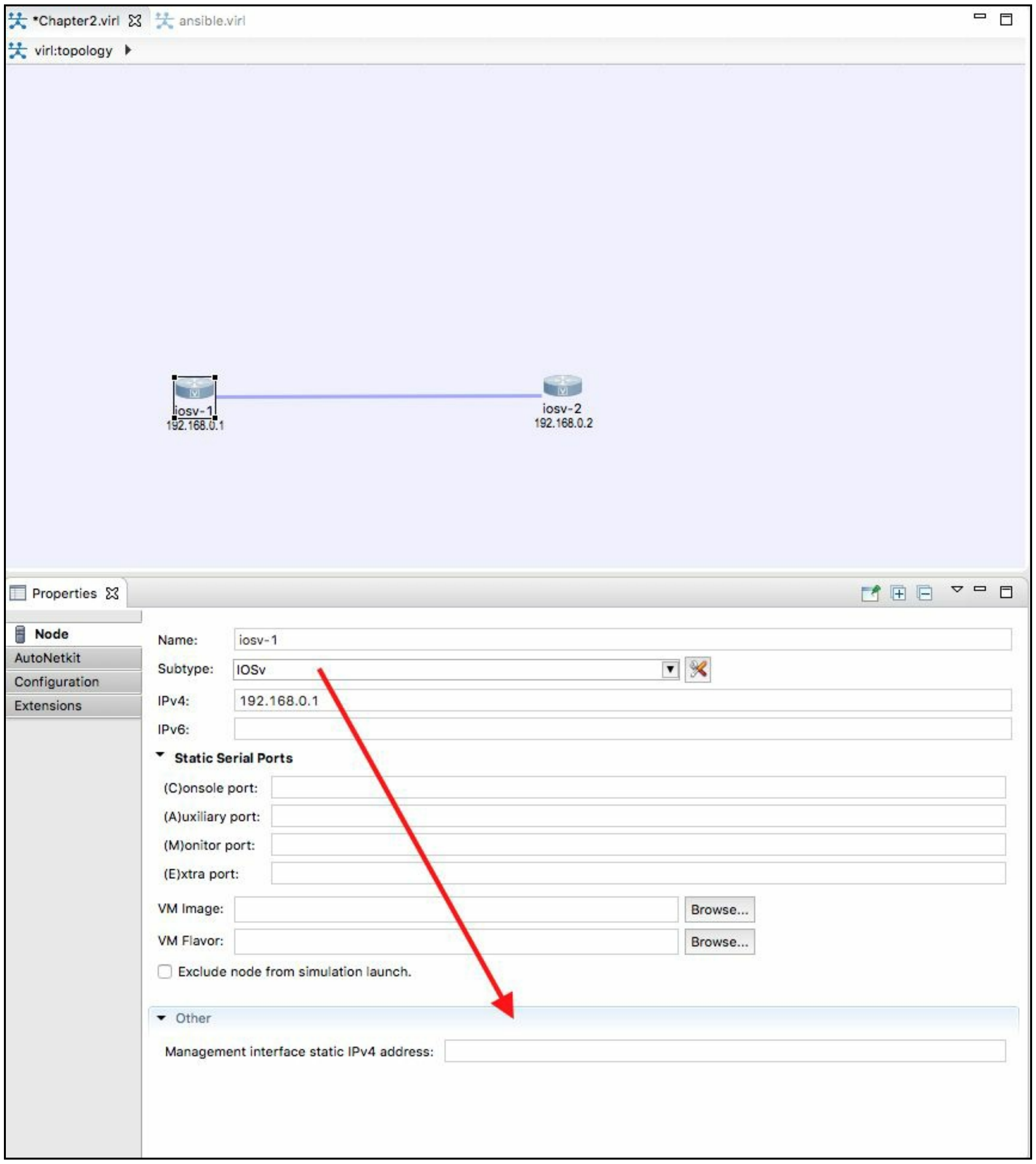


- In the Topology Design option, I set the Management Network option to Shared Flat Network in order to use VMnet2 as the management network on the virtual routers:



Nodes: 2
Simple Connections: 1
Management Network: Shared flat network
✓ Use an LXC management node
Static serial port offset:
LXC node's static IPv4 address:
LXC node's bound IPv4 address:
LXC node's static forwarded TCP port:

- Under the node configuration, you have the option to statically configure the management IP. I always statically set the IP address instead of what is dynamically assigned by the software, so I can easily get to the device without checking:



Name: iosv-1
Subtype: IOSv
IPv4: 192.168.0.1
IPv6:

**Static Serial Ports**

(C)onsole port:
(A)uxiliary port:
(M)onitor port:
(E)xtra port:

VM Image:
VM Flavor:

☐ Exclude node from simulation launch.

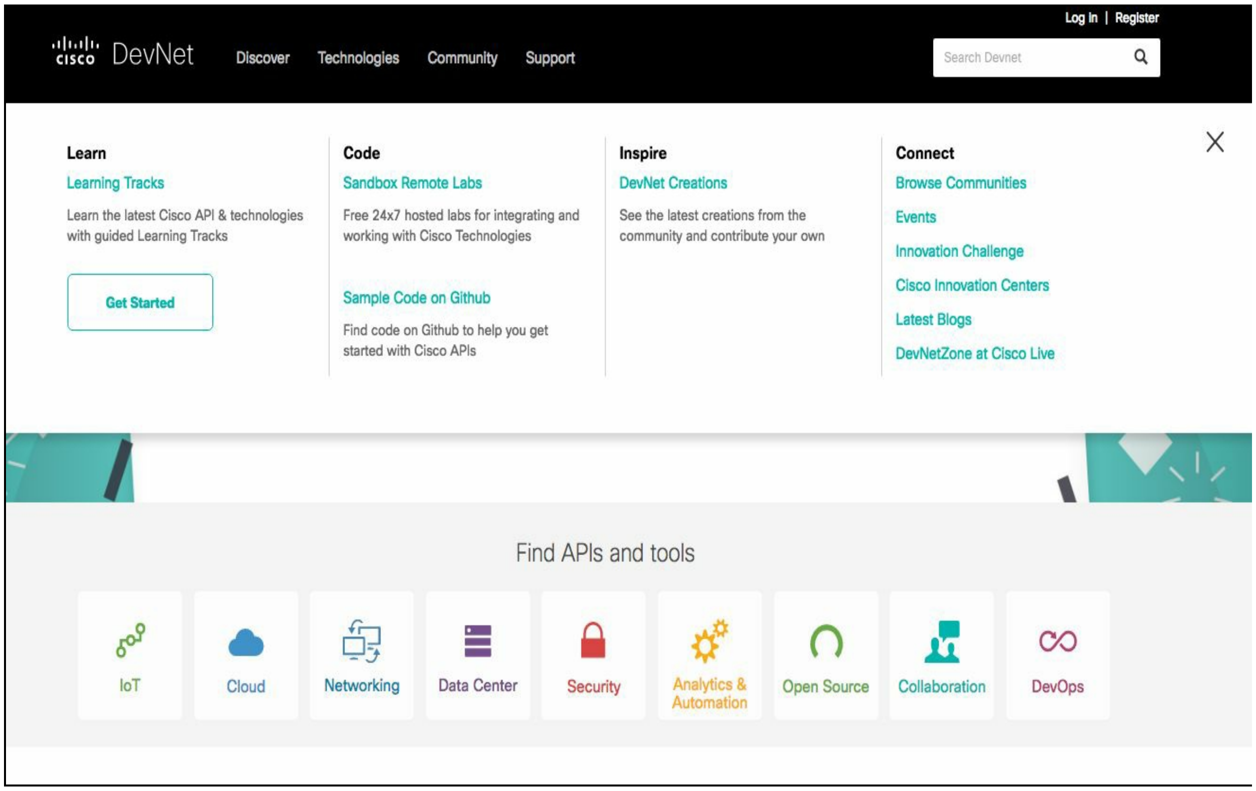
Other

Management interface static IPv4 address:

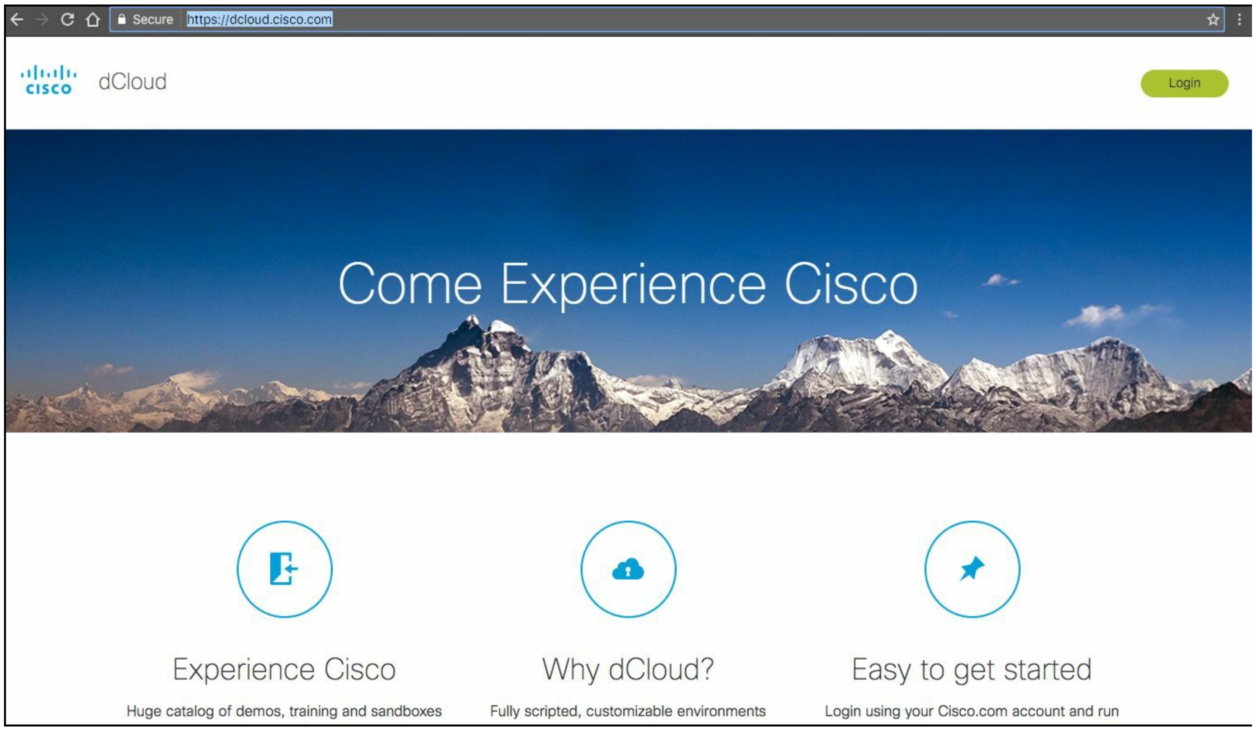
## Cisco DevNet and dCloud

Cisco provides two other excellent, at the time of writing free, methods of practice API interaction with various Cisco gears. Both of the tools require Cisco connection online login. They are both really good, especially for the price point (they are free!). It is hard for me to imagine that these online tools will remain free for long, it is my belief that at some point, these tools need to be charging money for usage or be rolled into a bigger initiative that required some fee. however we can take advantage of them while they are available at no extra charge.

The first tool is DevNet (https://developer.cisco.com/), which includes guided learning tracks, complete documentation, and sandbox remote labs, among other benefits. Some of the labs are always on, while some you need to reserve and availability depends on usage. In my experience with DevNet, some of the documentations and links were outdated, but they can be easily retrieved for the most updated version. In a rapidly changing field such as software development, this is somewhat expected. DevNet is certainly a tool that you should take full advantage of regardless of whether you have a locally run VIRL host or not:



Another online lab option for Cisco is Cisco dCloud. You can think of dCloud as running VIRL in other people's server without having to manage or pay for those resources. It seems that Cisco is treating dCloud as both a standalone product as well as an extension to VIRL. For example, the case when you are unable to run more than a few IOX-XR or NX-OS instances locally, you can use dCloud to extend your local lab. It is a relatively new tool, but it is definitely worth a look:



Experience Cisco

Huge catalog of demos, training and sandboxes

Why dCloud?

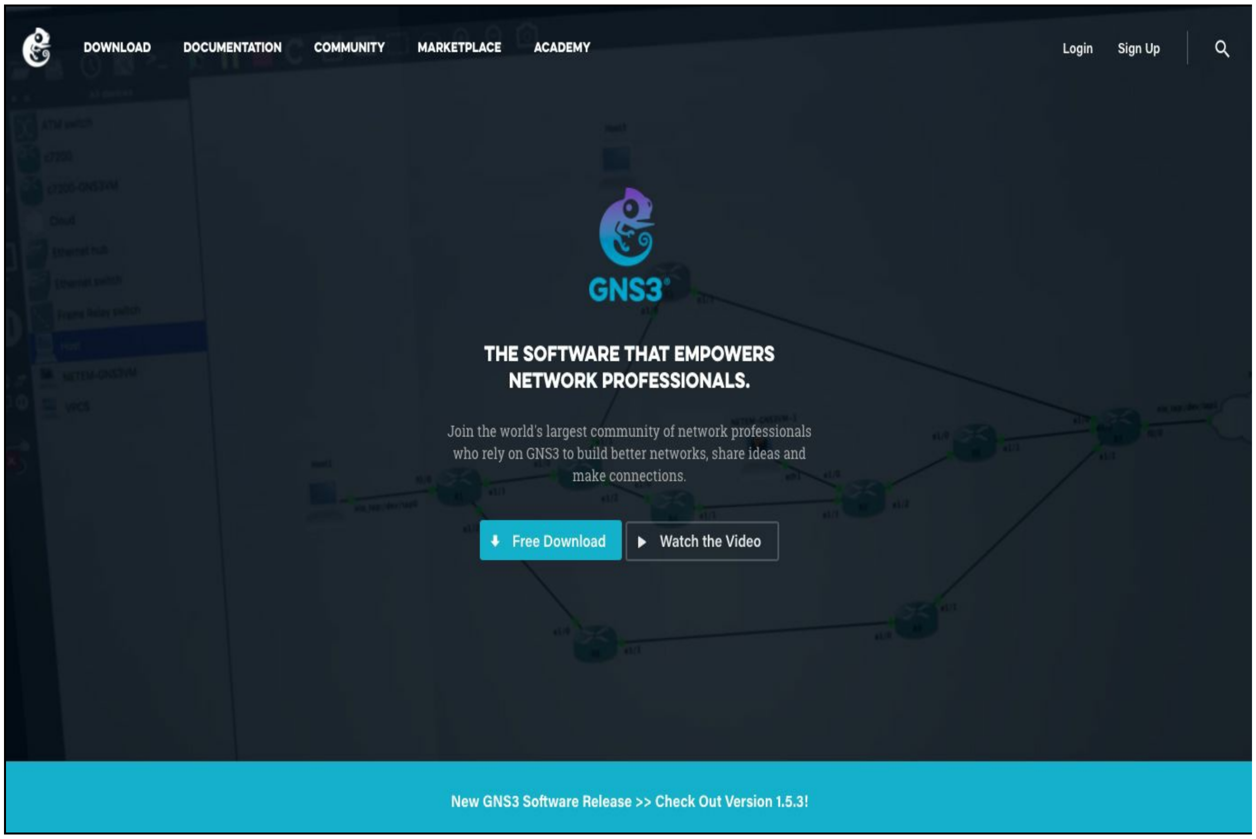
Fully scripted, customizable environments

Easy to get started

Login using your Cisco.com account and run

## GNS3

There are a few other virtual labs that I use for this book and other purposes and GNS3 is one of them:


DOWNLOAD
DOCUMENTATION
COMMUNITY
MARKETPLACE
ACADEMY
Login
Sign Up
GNS3
THE SOFTWARE THAT EMPOWERS NETWORK PROFESSIONALS.
Join the world's largest community of network professionals who rely on GNS3 to build better networks, share ideas and make connections.
Free Download
Watch the Video
New GNS3 Software Release >> Check Out Version 1.5.3!

As discussed, GNS3 is what a lot of us used to study for test and practice for labs. It has really grown up from the early days into a viable commercial product. Cisco-made tools, VIRL, DevNet, and dCloud only contain Cisco technologies. They provide ways to communicate to the interfaces outside of the lab; however, they are not as easy as just having other vendor's virtualized appliances living directly into the simulation environment. GNS3 is vendor neutral and can include other vendor's virtualized platform directly either by making an clone of the image (such as Arista vEOS), or by directly launch other images via other hypervisors (such as Juniper Olive emulation). GNS3 does not have the the breadth and depth of the Cisco VIRL project, but since they can run different variation Cisco technologies,  I use it many times when I need to incorporate other vendor technologies into the lab along with some Cisco images.

There are also other vendor virtualized platforms such as Arista vEOS (https://eos.arista.com/tag/veos/), Juniper vMX (http://www.juniper.net/us/en/products-services/routing/mx-series/vmx/), and vSRX (http://www.juniper.net/us/en/products-services/security/srx-series/vsrx/) that you can use as standalone virtual appliance. They are great complementary tools for testing platform specific features, such as the differences between the API versions. Many of them are offered as paid products on cloud provider market places and are offered the identical feature as their physical counterpart.

# Python Pexpect Library

*Pexpect is a pure Python module for spawning child applications, controlling them, and responding to expected patterns in their output. Pexpect works like Don Libes' Expect. Pexpct allows your script to spawn a child application and control it as if a human were typing commands. - Pexpect Read the Docs,* *https://pexpect.readthedocs.io/en/stable/index.html*

Like the original Expect module by *Don Libe*, Pexpect launches or spawns another process and watches over it in order to control the interaction. Unlike the original Expect, it is entirely written in Python that does not quire TCL or C extensions to be compiled. This allows us to use the familiar Python syntax and its rich standard library in our code.

## Installation

The installation of the pip and `pexpect` packages is straightforward:

```
sudo apt-get install python-pip
sudo apt-get install python3-pip
sudo pip3 install pexpect
sudo pip install pexpect
```

*I am using pip3 for installing Python 3 packages while using PIP for installing packages to my default Python 2 interpreter.*

Do a quick to test to make sure the package is usable:

```
>>> import pexpect
>>> dir(pexpect)
['EOF', 'ExceptionPexpect', 'Expecter', 'PY3',
  'TIMEOUT', '__all__', '__builtins__', '__cached__',
  '__doc__', '__file__', '__loader__', '__name__',
  '__package__', '__path__', '__revision__',
  '__spec__', '__version__', 'exceptions', 'expect',
  'is_executable_file', 'pty_spawn', 'run', 'runu',
  'searcher_re', 'searcher_string', 'spawn',
  'spawnbase', 'spawnu', 'split_command_line', 'sys',
  'utils', 'which']
>>>
```

# The Pexpect overview

Let's take a look at how you would interact with the router if you were to Telnet into the device:

```
echou@ubuntu:~$ telnet 172.16.1.20
Trying 172.16.1.20...
Connected to 172.16.1.20.
Escape character is '^]'.
<skip>
User Access Verification

Username: cisco
Password:
```

I used VIRL AutoNetkit to automatically generate initial configuration of the routers, which generated the default username of `cisc`, and the password of `cisco`. Notice the user is already in the privileged mode:

```
iosv-1#sh run | i cisco
enable password cisco
username cisco privilege 15 secret 5 $1$Wiwq$7xt2oE0P9ThdxFS02trFw.
 password cisco
 password cisco
iosv-1#
```

Also note that the auto config generated `vty` access for both telnet and SSH:

```
line vty 0 4
 exec-timeout 720 0
 password cisco
 login local
 transport input telnet ssh
```

Let's see a Pexpect example using the Python interactive shell:

```
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pexpect
>>> child = pexpect.spawn('telnet 172.16.1.20')
>>> child.expect('Username')
0
>>> child.sendline('cisco')
6
>>> child.expect('Password')
0
>>> child.sendline('cisco')
```

```
6
>>> child.expect('iosv-1#')
0
>>> child.sendline('show version | i V')
19
>>> child.expect('iosv-1#')
0
>>> child.before
b'show version | i VrnCisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M),
Version 15.6(2)T, RELEASE SOFTWARE (fc2)rnProcessor board ID
9MM4BI7B0DSWK40KV1IIRrn'
>>> child.sendline('exit')
5
>>> exit()
```

*Starting from Pexpect version 4.0, you can run Pexpect on a Windows platform. But as noted on the documentation, running Pexpect on Windows should be considered experimental for now.*

One thing to note in the previous interactive example is that as you can see, Pexpect spawns off a child process and watches over it in an interactive fashion. There are two important methods shown in the example, `expect()` and `sendline()`. There is also a method called `send()` but `sendline()` includes a linefeed, which is similar to pressing the enter key at the end of your line in your previous telnet session. From the router's perspective, it is just as if someone typed in the text from a Terminal. In other words, we are tricking the router into thinking they are interfacing with a human being when they are actually communicating with a computer.

The `before` and `after` properties will be set to the text printed by the child application. The `before` properties will be set to the text printed by child application up to the expected pattern. The `after` string will contain the text that was matched by the expected pattern. In our case, the before text will be set to the output between the two expected matches (`iosv-1#`) including the `show version` command. The after text is the router hostname prompt:

```
>>> child.sendline('show version | i V')
19
>>> child.expect('iosv-1#')
0
>>> child.before
b'show version | i VrnCisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M),
Version 15.6(2)T, RELEASE SOFTWARE (fc2)rnProcessor board ID
9MM4BI7B0DSWK40KV1IIRrn'
>>> child.after
b'iosv-1#'
```

What would happen if you expect the wrong term? For example, if you type in the lowercase `username` instead of `Username` after you spawn the child application, then the Pexpect process will be looking for a string of `username` from the child process. In this case, the Pexpect process will just hang because the word `username` would never be returned by the router. The session will eventually timeout, or you can manually exit out via *Ctrl* + *C*.

The `expect()` method waits for the child application to return a given string, so in the previous example, if you would like to accommodate for both lower and upper case `u`, you can use this term:

```
>>> child.expect('[Uu]sername')
```

The square bracket serves as an `or` operation that tells the child application to expect a lower or upper case `u` followed by 'sername' as the string. What we are telling the process is that we will accept either `'Username'` or `'username'` as the expected string.

*For more information on Python regular expression, go to https://docs.python.org/3.5/library/re.html.*

The `expect()` method can also contain a list of options instead of just a single string; these options can also be regular expression themselves. Going back to the previous example, you can use the following list of options to accommodate the two different possible string:

```
>>> child.expect(['Username', 'username'])
```

Generally speaking, use the regular expression for a single expect string when you can fit the different hostname in a regular expression, whereas use the possible options if you need to catch completely different responses from the router, such as a password rejection. For example, if you use several different passwords for your login, you want to catch`% Login invalid` as well as the device prompt.

One important difference between Pexpect RE and Python RE is that the Pexpect matching is non-greedy, which means they will match as little as possible when using special characters. Because Pexpect performs regular expression on a stream, you cannot look ahead, as the child process generating the stream may not be finished. This means the special character called `$` for the end of the line match is useless, `.+` will always return no characters, and `.*` pattern will match as little as possible. In general, just keep this in mind and be as specific as you can be on the expect match strings.

Let's consider the following scenario:

```
>>> child.sendline('show run | i hostname')
22
>>> child.expect('iosv-1')
0
>>> child.before
b'show run | i hostnamernhostname '
>>>
```

Hmm.. something is not quit right here. Compare to the terminal output before; the output you expect would be `hostname iosv-1`:

```
iosv-1#show run | i hostname
hostname iosv-1
iosv-1#
```

A closer look at the expected string will reveal the mistake. In this case, we were missing the hash (`#`) sign behind the hostname called `iosv-1`. Therefore, the child application treated the second part of the return string as the expected string:

```
>>> child.sendline('show run | i hostname')
22
>>> child.expect('iosv-1#')
0
>>> child.before
b'show run | i hostnamernhostname iosv-1rn'
>>>
```

Pretty quickly, you can see a pattern emerging from the usage of Pexpect. The user maps out the sequence of interaction between the Pexpect process and the child application. With some Python variables and loops, we can start to construct a useful program that will help us gather information and make changes to network devices.

## Our first Expect program

Our first program extends what we have done in the last section with some additional code:

```
     #!/usr/bin/python3

     import pexpect

     devices = {'iosv-1': {'prompt': 'iosv-1#', 'ip': '172.16.1.20'}, 'iosv-2':
{'prompt': 'iosv-2#', 'ip': '172.16.1.21'}}
     username = 'cisco'
     password = 'cisco'

     for device in devices.keys():
         device_prompt = devices[device]['prompt']
         child = pexpect.spawn('telnet ' + devices[device]['ip'])
         child.expect('Username:')
         child.sendline(username)
         child.expect('Password:')
         child.sendline(password)
         child.expect(device_prompt)
         child.sendline('show version | i V')
         child.expect(device_prompt)
         print(child.before)
         child.sendline('exit')
```

We use a nested dictionary in line 5:

```
       devices = {'iosv-1': {'prompt': 'iosv-1#', 'ip':
      '172.16.1.20'}, 'iosv-2': {'prompt': 'iosv-2#',
      'ip': '172.16.1.21'}}
```

The nested dictionary allows us to refer to the same device (such as iosv-1) with the appropriate IP address and prompt symbol. We can then use those values for the `expect()` method later on in the loop.

The output prints out the `'show version | i V'` output on the screen for each of the devices:

```
    $ python3 chapter2_1.py
      b'show version | i VrnCisco IOS Software, IOSv
      Software (VIOS-ADVENTERPRISEK9-M), Version 15.6(2)T,
    RELEASE SOFTWARE (fc2)rnProcessor board ID
    9MM4BI7B0DSWK40KV1IIRrn'
    b'show version | i VrnCisco IOS Software, IOSv
    Software (VIOS-ADVENTERPRISEK9-M), Version 15.6(2)T,
     RELEASE SOFTWARE (fc2)rn'
```

## More Pexpect features

In this section, we will look at more Pexpect features that might come in handy when the situation arises.

If you have a slow link to your remote device, the default `expect()` method timeout is 30 seconds, which you can be increased via the `timeout` argument:

```
>>> child.expect('Username', timeout=5)
```

You can choose to pass the command back to the user using the `interact()` method. This is useful when you just want to automate certain parts of the initial task:

```
>>> child.sendline('show version | i V')
19
>>> child.expect('iosv-1#')
0
>>> child.before
b'show version | i VrnCisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M),
Version 15.6(2)T, RELEASE SOFTWARE (fc2)rnProcessor board ID
9MM4BI7B0DSWK40KV1IIRrn'
>>> child.interact()
iosv-1#show run | i hostname
hostname iosv-1
iosv-1#exit
Connection closed by foreign host.
>>>
```

You can get a lot of information about the `child.spawn` object by printing it out in a string format:

```
>>> str(child)
"<pexpect.pty_spawn.spawn object at 0x7fb01e29dba8>ncommand: /usr/bin/telnetnargs:
['/usr/bin/telnet', '172.16.1.20']nsearcher: Nonenbuffer (last 100 chars):
b''nbefore (last 100 chars): b'NTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE
(fc2)\r\nProcessor board ID 9MM4BI7B0DSWK40KV1IIR\r\n'nafter: b'iosv-1#'nmatch:
<_sre.SRE_Match object; span=(164, 171), match=b'iosv-1#'>nmatch_index:
0nexitstatus: 1nflag_eof: Falsenpid: 2807nchild_fd: 5nclosed: Falsentimeout:
30ndelimiter: <class 'pexpect.exceptions.EOF'>nlogfile: Nonenlogfile_read:
Nonenlogfile_send: Nonenmaxread: 2000nignorecase: Falsensearchwindowsize:
Nonendelaybeforesend: 0.05ndelayafterclose: 0.1ndelayafterterminate: 0.1"
>>>
```

The most useful debug tool for Pexpect is to log the output in a file:

```
>>> child = pexpect.spawn('telnet 172.16.1.20')
>>> child.logfile = open('debug', 'wb')
```

*Use `child.logfile = open('debug', 'w')` for Python 2. Python 3 uses byte string by default. For more information on Pexpect features, check https://pexpect.readthedocs.io/en/stable/api/index.html.*

## Pexpect and SSH

If you try to use the previous telnet example and just plug into a ssh session, you might find yourself pretty frustrated. You  always have to include the username in the session, `ssh` new key question, and more such. There are many ways to make ssh sessions work, but luckily, Pexpect has a subclass called `pxssh`, which specializes setting up SSH connections. The class adds methods for login, logout, and various tricky things to handle many situation in the `ssh` login process. The procedures are mostly the same with the exception of `login()` and `logout()`:

```
>>> from pexpect import pxssh
>>> child = pxssh.pxssh()
>>> child.login('172.16.1.20', 'cisco', 'cisco', auto_prompt_reset=False)
True
>>> child.sendline('show version | i V')
19
>>> child.expect('iosv-1#')
0
>>> child.before
b'show version | i VrnCisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M),
Version 15.6(2)T, RELEASE SOFTWARE (fc2)rnProcessor board ID
9MM4BI7B0DSWK40KV1IIRrn'
>>> child.logout()
>>>
```

Notice the `'auto_prompt_reset=False'` argument in the `login()` method. By default, `pxssh` uses the shell prompt to synchronize the output. But since it uses the PS1 option for most of bash or Csh, they will error out on Cisco or other network devices.

# Putting things together for Pexpect

As the final step, let's put everything you learned so far for Pexpect into a script. Putting the code in a script makes it easier to use it in a production environment as well as share the code with your colleagues.

*You can download the script from the book GitHub repository, h ttps://github.com/PacktPublishing/Mastering-Python-Networking as well as look at the output generated from the script as a result of the commands.*

```
  #!/usr/bin/python3

  import getpass
  from pexpect import pxssh

  devices = {'iosv-1': {'prompt': 'iosv-1#', 'ip': '172.16.1.20'},
  'iosv-2': {'prompt': 'iosv-2#', 'ip': '172.16.1.21'}}
  commands = ['term length 0', 'show version', 'show run']

  username = input('Username: ')
  password = getpass.getpass('Password: ')

  # Starts the loop for devices
  for device in devices.keys():
      outputFileName = device + '_output.txt'
      device_prompt = devices[device]['prompt']
      child = pxssh.pxssh()
      child.login(devices[device]['ip'], username.strip(), password.strip(),
auto_promp t_reset=False)
      # Starts the loop for commands and write to output
      with open(outputFileName, 'wb') as f:
          for command in commands:
              child.sendline(command)
              child.expect(device_prompt)
              f.write(child.before)

      child.logout()
```

The script further expands on our first Pexpect program with the following additional features:

- It use SSH instead of telnet
- It supports multiple commands instead of just one by making the commands into a list (line 8) and loop through the commands (starting line 20)

- It prompts the user for username and password instead of hardcoding them in the script
- It writes the output into a file to be further analyzed

TIP *For Python 2, use* `raw_input()` *instead of* `input()` *for the username prompt. Also use* `w` *for file mode instead of* `wb`*.*

## The Python Paramiko library

Paramiko is a Python implementation of the SSHv2 protocol. Just like the `pxssh` subclass of Pexpect, Paramiko simplifies the SSHv2 interaction with the remote device. Unlike `pxssh`, Paramiko is only focused on SSHv2 and provides both client and server operations.

Paramiko is the low-level SSH client behind the high-level automation framework Ansible for its network modules. We will cover Ansible in the later chapters, so first, let's take a look at the Paramiko library.

# Installating Paramiko

Installating Paramiko is pretty straight forward with PIP. However, there is a hard dependency on the Cryptography library. The library provides the low-level, C-based encryption algorithms for the SSH protocol.

*The installation instruction for Windows, Mac, and other flavors of Linux can be found* *https://cryptography.io/en/latest/installation/*

We will show the installation for our Ubuntu 16.04 virtual machine in the following output. The following output shows the installation steps as well as Paramiko successfully imported in the Python interactive prompt.

**Python 2**:

```
sudo apt-get install build-essential libssl-dev libffi-dev python-dev
sudo pip install cryptography
sudo pip install paramiko
$ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import paramiko
>>> exit()
```

**Python 3**:

```
sudo apt-get install build-essential libssl-dev libffi-dev python3-dev
sudo pip3 install cryptography
sudo pip3 install paramiko
$ python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import paramiko
>>>
```

# The Paramiko overview

Let's look at a quick example using the Python 3 interactive shell:

```
>>> import paramiko, time
>>> connection = paramiko.SSHClient()
>>> connection.set_missing_host_key_policy(paramiko.AutoAddPolicy())
>>> connection.connect('172.16.1.20', username='cisco', password='cisco',
look_for_keys=False, allow_agent=False)
>>> new_connection = connection.invoke_shell()
>>> output = new_connection.recv(5000)
>>> print(output)
b"rn***************************************************************************rn*
IOSv is strictly limited to use for evaluation, demonstration and IOS *rn*
education. IOSv is provided as-is and is not supported by Cisco's *rn* Technical
Advisory Center. Any use or disclosure, in whole or in part, *rn* of the IOSv
Software or Documentation to any third party for any *rn* purposes is expressly
prohibited except as otherwise authorized by *rn* Cisco in writing.
*rn***************************************************************************rniosv
1#"
>>> new_connection.send("show version | i Vn")
19
>>> time.sleep(3)
>>> output = new_connection.recv(5000)
>>> print(output)
b'show version | i VrnCisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M),
Version 15.6(2)T, RELEASE SOFTWARE (fc2)rnProcessor board ID
9MM4BI7B0DSWK40KV1IIRrniosv-1#'
>>> new_connection.close()
>>>
```

*The `time.sleep()` function inserts time delay to ensure that all the outputs were captured. This is particularly useful on a slower network connection or a busy device. This command is not required but recommended depending on your situation.*

Even if you are seeing the Paramiko operation for the first time, the beauty of Python and its clear syntax means that you can make a pretty good educated guess at what the program is trying to do:

```
>>> import paramiko
>>> connection = paramiko.SSHClient()
>>> connection.set_missing_host_key_policy(paramiko.AutoAddPolicy())
>>> connection.connect('172.16.1.20', username='cisco', password='cisco',
look_for_keys=False, allow_agent=False)
```

The first four lines create an instance of the `SSHClient` class from Paramiko.

The next line sets the policy that the client should use when the SSH server's hostname, in this case `iosv-1`, is not present in either the system host keys or the application's keys. In this case, we will just automatically add the key to the application's HostKeys object. At this point, if you log onto the router, you will see the additional login session from Paramiko:

```
iosv-1#who
 Line User Host(s) Idle Location
*578 vty 0 cisco idle 00:00:00 172.16.1.1
 579 vty 1 cisco idle 00:01:30 172.16.1.173
Interface User Mode Idle Peer Address
iosv-1#
```

The next few lines invokes a new interactive shell from the connection and a repeatable pattern of sending a command and retrieves the output. Finally we close the connection.

Why do we need to invoke an interactive shell instead of using another method called `exec_command()`? Unfortunately, `exec_command()` on Cisco IOS only allows a single command. Consider the following example with `exec_command()` for the connection:

```
>>> connection.connect('172.16.1.20', username='cisco', password='cisco',
look_for_keys=False, allow_agent=False)
>>> stdin, stdout, stderr = connection.exec_command('show version | i V')
>>> stdout.read()
b'Cisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M), Version 15.6(2)T,
RELEASE SOFTWARE (fc2)rnProcessor board ID 9MM4BI7B0DSWK40KV1IIRrn'
>>>
```

Everything works great, however, if you look at the number of sessions on the Cisco device, you will notice that the connection is dropped by the Cisco device without you closing the connection:

```
iosv-1#who
 Line User Host(s) Idle Location
*578 vty 0 cisco idle 00:00:00 172.16.1.1
Interface User Mode Idle Peer Address
iosv-1#
```

Furthermore, `exec_command()` will return an error of SSH session not being active:

```
>>> stdin, stdout, stderr = connection.exec_command('show version | i V')
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 File "/usr/local/lib/python3.5/dist-packages/paramiko/client.py", line 435, in
exec_command
 chan = self._transport.open_session(timeout=timeout)
 File "/usr/local/lib/python3.5/dist-packages/paramiko/transport.py", line 711, in
open_session
 timeout=timeout)
 File "/usr/local/lib/python3.5/dist-packages/paramiko/transport.py", line 795, in
open_channel
 raise SSHException('SSH session not active')
paramiko.ssh_exception.SSHException: SSH session not active
>>>
```

*The Netmiko library by Kirk Byers is an open source Python library that simplifies SSH management to network devices. To read about it, check out the article,* https://pynet.twb-tech.com/blog/automation/netmiko.html *and the source code,* https://github.com/ktbyers/netmiko.

What would happen if you do not clear out the received buffer? The output will just keep on filling up the buffer and overwrite it:

```
>>> new_connection.send("show version | i Vn")
19
>>> new_connection.send("show version | i Vn")
19
>>> new_connection.send("show version | i Vn")
19
>>> new_connection.recv(5000)
b'show version | i VrnCisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M),
Version 15.6(2)T, RELEASE SOFTWARE (fc2)rnProcessor board ID
9MM4BI7B0DSWK40KV1IIRrniosv-1#show version | i VrnCisco IOS Software, IOSv Software
(VIOS-ADVENTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE (fc2)rnProcessor board
ID 9MM4BI7B0DSWK40KV1IIRrniosv-1#show version | i VrnCisco IOS Software, IOSv
Software (VIOS-ADVENTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE
(fc2)rnProcessor board ID 9MM4BI7B0DSWK40KV1IIRrniosv-1#'
>>>
```

For the consistency of deterministic output, we will retrieve the output from buffer each time we execute a command.

## Our first Paramiko program

Our first program will use the same general structure of the Pexpect program as far as looping over a list of devices and commands, with the exception of using Paramiko instead of Pexpect. This will give us a good compare and contrast of the differences between the two programs.

You can download the the code from the book Github repository, https://github.com/PacktPublishing/Mastering-Python-Networking. I will list out the notable differences here:

```
devices = {'iosv-1': {'ip': '172.16.1.20'}, 'iosv-2': {'ip': '172.16.1.21'}}
```

We no longer need to match the device prompt using Paramiko, therefore, the device dictionary can be simplified:

```
  commands = ['show versionn', 'show runn']
```

There is no send line equivalent in Paramiko, so we are manually including the newline break in each of the commands:

```
def clear_buffer(connection):
    if connection.recv_ready():
        return connection.recv(max_buffer)
```

We are including a new method to clear the buffer for sending commands such as `terminal length 0` or `enable` because we do not need to the output for those commands. We simply want to clear the buffer and get to the execution prompt. This function will later on be used in the loop such as in line 25:

```
      output = clear_buffer(new_connection)
```

The rest of the program should be pretty self explanatory, as we have seen the usage in this chapter. The last thing I would like to point out is that since this is an interactive program, we placed some buffer and wait for the command to be finished on the remote device before retrieving the output:

```
time.sleep(2)
```

After we clear the buffer, during the time in-between the execution of commands, we will wait two seconds. This will ensure us to give the device adequate time in case it is busy.

## More Paramiko features

We will work with Paramiko later when we discuss Ansible, as Paramiko is the underlying transport for many of the network modules. In this section, we will take a look at some of the other features for Paramiko.

## Paramiko for Servers

Paramiko can be used to manage servers through SSHv2 as well. Let's look at an example of how we can use Paramiko to manage servers. We will use key-based authentication for the SSHv2 session.

*In this example, I used another virtual machine on the same hypervisor as the destination server. You can also use a server on the VIRL simulator or an instance in one of the public cloud providers, such as Amazon AWS EC2.*

We will generate a public-private key pair for our Paramiko host:

```
ssh-keygen -t rsa
```

This command, by default, will generate  a public key named `id_rsa.pub`, as the public key under the user directory called `~/.ssh` along with a private key named `id_rsa`. Treat the private key as your password that you do not want to share, but treat the public key as a business card that identifies who you are. Together, the message will be encrypted by your private key locally and decrypted by remote host using the public key. Therefore, we should copy the public key to the remote host. In production, we can do this via out-of-band using an USB drive; in our lab, we can simply just copy the public key file to the remote host's `~/.ssh/authorized_keys` file. Open up a Terminal window for the remote server, so you can paste in the public key.

Copy the content of `~/.ssh/id_rsa` on your management host with Pramiko:

```
<Management Host with Pramiko>$ cat ~/.ssh/id_rsa.pub
ssh-rsa <your public key> echou@pythonicNeteng
```

Then, paste it to the remote host under the user directory; in this case I am using `echou` for both the sides:

```
<Remote Host>$ vim ~/.ssh/authorized_keys
ssh-rsa <your public key> echou@pythonicNeteng
```

You are now ready to use Paramiko to manage the remote host:

```
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import paramiko
>>> key = paramiko.RSAKey.from_private_key_file('/home/echou/.ssh/id_rsa')
>>> client = paramiko.SSHClient()
>>> client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
>>> client.connect('192.168.199.182', username='echou', pkey=key)
>>> stdin, stdout, stderr = client.exec_command('ls -l')
>>> stdout.read()
b'total 44ndrwxr-xr-x 2 echou echou 4096 Jan 7 10:14 Desktopndrwxr-xr-x 2 echou
echou 4096 Jan 7 10:14 Documentsndrwxr-xr-x 2 echou echou 4096 Jan 7 10:14
Downloadsn-rw-r--r-- 1 echou echou 8980 Jan 7 10:03 examples.desktopndrwxr-xr-x 2
echou echou 4096 Jan 7 10:14 Musicndrwxr-xr-x 2 echou echou 4096 Jan 7 10:14
Picturesndrwxr-xr-x 2 echou echou 4096 Jan 7 10:14 Publicndrwxr-xr-x 2 echou echou
4096 Jan 7 10:14 Templatesndrwxr-xr-x 2 echou echou 4096 Jan 7 10:14 Videosn'
>>> stdin, stdout, stderr = client.exec_command('pwd')
>>> stdout.read()
b'/home/echoun'
>>> client.close()
>>>
```

Notice that in the server example, we do not need to create an interactive session to execute multiple commands. You can now turn off password-based authentication in your remote host's SSHv2 configuration for a more secured key-based authentication with automation enabled.

## Putting things together for Paramiko

We are almost at the end of the chapter. In this last section, let's make the Paramiko program more reusable. There is one downside for our existing script; we need to open up the script every time we want to add or delete a host; or whenever we need to change the commands we have to execute on the remote host. Besides having a higher chance for mistakes, how about when you need to let your co-workers or NOC use the script? They might not feel comfortable working in Python, Paramiko, or Linux.

By making both the hosts and commands files that we read in as a parameter for the script, we can eliminate some of these concerns. The users (and a future you) can simply modify these text files when you need to make host or command changes.

The file is named `chapter2_4.py`.

We break the commands into a `commands.txt` file. Up to this point, we have been using show commands; in this example, we will make configuration changes for the logging buffer size:

```
$ cat commands.txt
config t
logging buffered 30000
end
copy run start
```

The devices information is written into a `devices.json` file. We choose JSON format for the devices information because JSON data types can be easily translated into Python dictionary data types:

```
$ cat devices.json
{
     "iosv-1": {"ip": "172.16.1.20"},
     "iosv-2": {"ip": "172.16.1.21"}
    }
```

In the script, we made the following changes:

```
with open('devices.json', 'r') as f:
    devices = json.load(f)

with open('commands.txt', 'r') as f:
    commands = [line for line in f.readlines()]
```

Here is an abbreviated output from the script execution:

```
$ python3 chapter2_4.py
Username: cisco
Password:
b'terminal length 0rniosv-2#config trnEnter configuration commands, one per line.
End with CNTL/Z.rniosv-2(config)#'
b'logging buffered 30000rniosv-2(config)#'
...
```

Do a quick check to make sure the change has taken place in both running-config and startup-config:

```
iosv-1#sh run | i logging
logging buffered 30000
iosv-1#sh start | i logging
logging buffered 30000
```

```
iosv-2#sh run | i logging
logging buffered 30000
iosv-2#sh start | i logging
logging buffered 30000
```

## Looking ahead

We have taken a pretty huge leap forward as far as automating our network using Python in this chapter. However, the method we have used is somewhat a work-around for the automation. This is mainly due to the fact that the systems were originally built for human.

## Downsides of Pexpect and Paramiko compared to other tools

The number one downside for our method so far is that they do not return structured data. They return the data that we find ideal to fit on a terminal to be interpreted by a human but not by a computer program. The human eye can easily interpret a space while the computer only sees a return character. This, of course, depends on the network devices to be made that is automation friendly.

We will take a look at a better way in the next chapter.

## Idempotent network device interaction

The term **idempotency** has different meanings, depending on its context. But in this chapter's context, the term means when the client makes the same call to the device, the result should always be the same. I believe we can all agree that this is necessary. Imagine a time when each time you execute the script you get a different result back. This is a scary thought indeed and would render our automation effort useless.

Since Pexpect and Paramiko are blasting out a series of commands interactively, the chance of having a non-idempotent interaction is higher. Going back to the fact that the return results needed to be screen scraped for useful elements, the risk is much higher that something might have changed between the time we wrote the script to the time when the script is executed for the 100th time. For example, if the vendor makes a screen output change between releases, it might be okay for human engineers, but it will break your script.

If we need to rely on the script for production, we need the script to be idempotent as much as possible.

## Bad automation speeds bad things up

Bad automation allows you to poke yourselves in the eye a lot faster; it is as simple as that. Computers are much faster at executing tasks than us human engineers. If we had the same set of operating procedures executing by a human versus script, the script will finish faster than humans, sometimes without the benefit of having a solid feedback look between procedures. The Internet is full of horror stories when someone presses the *Enter* key and immediately regrets it.

We need to make sure that the chance of a bad automation script can go wrong is as small as possible.

## Summary

In this chapter, we covered the low-level ways to communicate directly to the network devices. Without a way to programmatically communicate and make changes to network devices, there is no automation. We looked at two libraries in Python that allow us to manage the devices that were meant to be managed by CLI. Although useful, it is easy to see how the process can be somewhat fragile. This is mostly due to the fact that the network gears in question were meant to be managed by human beings and not computers.

In the next chapter, we will look at network devices supporting API and intent-driven networking.

# API and Intent-Driven Networking

In the previousÂ chapter, we looked at ways to interact with the device interactively using Pexpect and Paramiko. Both of these tools use a persistent session that simulates a user typing in commands as if they are sitting in front of the Terminal. This works fine up to a point. It is easy enough to send commands over for execution on the device and capture the output back. However, when the output becomes more than a few lines of characters, it becomes difficult for a computer program to interpret the output. In order for our simple computer program to automate some of whatÂ we do, we need to be able to interpret the returned results and make follow-up actions based on the returned result. When we cannot accurately and predictably interpret the results back, we cannot execute the next command with confidence.Â

Luckily, this problem was solved by the Internet community. Imagine the difference between a computer and a human being reading a web page. The human sees words, pictures, and spaces interpreted by the browser; the computer sees raw HTML code, Unicode characters, and binary files. What happens when a website needs to become a web service for another computer? Doesn't this problem sound familiar to the one that we presented before? The answer is the application program interface, or API for short. An API is a concept; according to Wikipedia:

*In computer programming , anÂ A**pplication Programming Interface (API)** is a set of subroutineÂ  definitions, protocols, and tools for building application software . In general terms, it's a set of clearly defined methods of communication between various software components. A good API makes it easier to develop a computer programÂ  by providing all the building blocks, which are then put together by the programmer.*

In our use case, the set of clearly defined methods of communication would be between our Python program and the destination device.

In this chapter, we will look at the following topics:

- Treating infrastructure as code and data modeling
- Cisco NX-API and application-centric infrastructure
- Juniper NETCONF and PyEZ
- Arista eAPI and pyeapi

## Infrastructure as the Python code

In a perfect world, network engineers and people who design and manage networks should focus on what they want the network to achieve instead of the device-level interactions. In my first job as an intern for a local ISP, wide-eyed and excited, I received my first assignment to install a router on a customer's site to turn up their fractional frame relay link (remember those?). *How would I do that?* I asked. I was handed a standard operating procedure for turning up frame relay links. I went to the customer site, blindly typed in the commands and looked at the green lights flash, and then I happily packed my bag and patted myself on the back for a job well done. As exciting as that first assignment was, I did not fully understand what I was doing. I was simply following instructions without thinking about the implication of the commands I was typing in. How would I troubleshoot something if the light was red instead of green? I think I would have called back to the office.

Of course, network engineering is not about typing in commands onto a device, but it is about building a way that allows services to be delivered from one point to another with as little friction as possible. The commands we have to use and the output that we have to interpret are merely a means to an end. I would like to hereby argue that we should focus as much on the intent of the network as possible for an intent-driven networking and abstract ourselves from the device-level interaction on an as-needed basis.

In using an API, it is my opinion that it gets us closer to a state of intent-driven networking. In short, because we abstract the layer of a specific command executed on destination device, we focus on our intent instead of the specific command given to the device. For example, if our intend is to deny an IP from entering our network, we might use access-list and access-group on a Cisco and filter-list on a Juniper. However, in using API, our program can start asking the executor for their intent while masking what kind of physical device it is they are talking to.

## Screen scraping versus API structured output

Imagine a common scenario where we need to log into the device and make sure all the interfaces on the devices are in an up/up state (both status and protocol are showing as `up`). For the human network engineers getting into a Cisco NX-OS device, it is simple enough to issue the `show IP interface brief` command to easily tell from the output which interface is up:

```
nx-osv-2# show ip int brief
IP Interface Status for VRF "default"(1)
Interface IP Address Interface Status
Lo0 192.168.0.2 protocol-up/link-up/admin-up
Eth2/1 10.0.0.6 protocol-up/link-up/admin-up
nx-osv-2#
```

The line break, white spaces, and the first line of the column title are easily distinguished from the human eye. In fact, they are there to help us line up, say, the IP addresses of each interface from line 1 to line 2 and 3. If we were to put ourselves into the computer's position, all these spaces and line breaks are only taking us away from the really important output, which is: which interfaces are in the up/up state? To illustrate this point, we can look at the Paramiko output again:

```
>>> new_connection.send('sh ip int briefn')
16
>>> output = new_connection.recv(5000)
>>> print(output)
b'sh ip int briefrrnIP Interface Status for VRF
"default"(1)rnInterface IP Address Interface
StatusrnLo0 192.168.0.2 protocol-up/link-up/admin-up
rnEth2/1 10.0.0.6 protocol-up/link-up/admin-up rnrnx-
osv-2# '
>>>
```

If we were to parse out that data, then of course, there are many ways to do it, but here is what I would do in a pseudo code fashion:

1. Split each line via the line break.
2. I may or may not need the first line that contains the executed command;

for now, I don't think I need it.

3. Take out everything on the second line up until the VRF, and save it in a variable as we want to know which VRF the output is showing.
4. For the rest of the lines, because we do not know how many interfaces there are, we will do a regular expression to search if the line starts with possible interfaces, such as `lo` for loopback and `Eth`.

5. We will then split this line into three sections via space, each consisting of the name of the interface, IP address, and then the interface status.
6. The interface status will then be split further using the forward slash (`/`) to give us the protocol, link, and the admin status.

Whew, that is a lot of work just for something that a human being can tell in a glance! You might be able to optimize the code and the number of lines, but in general, this is what we need to do when we need to screen scrap something that is somewhat unstructured. There are many downsides to this method, but the few bigger problems that I see are here:

- **Scalability**: We spent so much time on painstaking details for each output that it is hard to imagine we can do this for the hundreds of commands that we typically run.
- **Predicability**: There is really no guarantee that the output stays the same. If the output is changed ever so slightly, it might just enter with our hard earned battle of information gathering.
- **Vendor and software lock-in**: Perhaps the biggest problem is that once we spent all this time parsing the output for this particular vendor and software version, in this case Cisco NX-OS, we need to repeat this process for the next vendor that we pick. I don't know about you, but if I were to evaluate a new vendor, the new vendor is at a severe on-boarding disadvantage if I had to rewrite all the screen scrap code again.

Let's compare that with an output from an NX-API call for the same show IP interface brief command. We will go over the specifics of getting this output from the device later in this chapter, but what is important here is to compare the following output to the previous screen scraping steps:

```
{
 "ins_api":{
 "outputs":{
 "output":{
 "body":{
 "TABLE_intf":[
   {
   "ROW_intf":{
   "admin-state":"up",
   "intf-name":"Lo0",
   "iod":84,
   "ip-disabled":"FALSE",
   "link-state":"up",
   "prefix":"192.168.0.2",
   "proto-state":"up"
   }
   },
 {
 "ROW_intf":{
 "admin-state":"up",
 "intf-name":"Eth2/1",
 "iod":36,
 "ip-disabled":"FALSE",
 "link-state":"up",
 "prefix":"10.0.0.6",
 "proto-state":"up"
 }
 }
 ],
  "TABLE_vrf":[
  {
 "ROW_vrf":{
 "vrf-name-out":"default"
 }
 },
 {
 "ROW_vrf":{
 "vrf-name-out":"default"
 }
 }
 ]
 },
 "code":"200",
 "input":"show ip int brief",
 "msg":"Success"
 }
 },
 "sid":"eoc",
 "type":"cli_show",
 "version":"1.2"
 }
}
```

NX-API can return output in XML or JSON, and this is obviously the JSON output that we are looking at. Right away, you can see the answers are structured and can be mapped directly to the Python dictionary data structure. There is no parsing required, so simply pick the key you want and retrieve the value associated with this key. There is also an added benefit of a code to indicate command success or failure, with a message telling the sender reasons behind the success or failure. You no longer need to keep track of the command issued, because it is already returned to you in the `input` field. There is also other meta data such as the NX-API version.

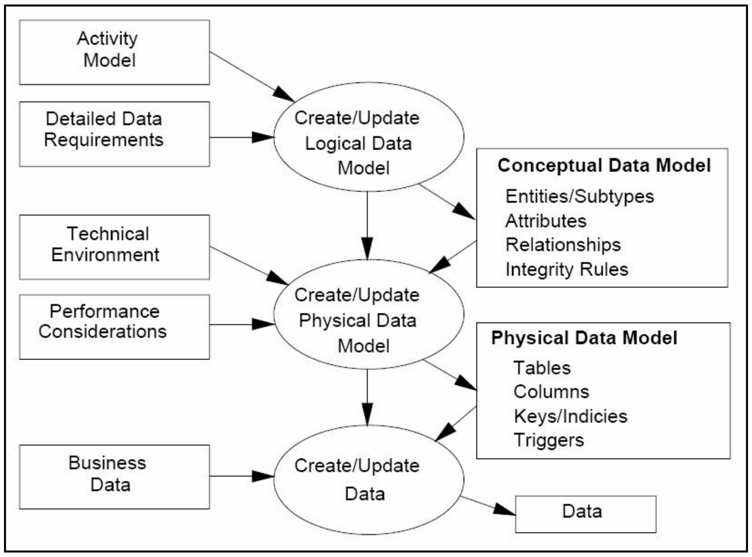
This type of exchange makes life easier for both vendors and operators. On the vendor side, they can easily transfer configuration and state information, as well as add and expose extra fields when the need rises. On the operator side, they can easily ingest the information and build their infrastructure around it. It is generally agreed on that automation is much needed and a good thing. The questions are usually centered around which format and structure the automation should take place. As you can see later in this chapter, there are many competing technologies under the umbrella of API, as on the transport side alone, we have REST API, NETCONF, and RESTCONF, among others. Ultimately, the overall market will decide about this, but in the meantime, we should all take a step back and decide which technology best suits our need.

# Data modeling for infrastructure as code

According to Wikipedia,

*"A data model is an abstract model that organizes elements of data and standardizes how they relate to one another and to properties of the real-world entities. For instance, a data model may specify that the data element representing a car be composed of a number of other elements which, in turn, represent the color and size of the car and define its owner."*

The data modeling process can be illustrated in the following graph:



Data Modeling Process (source: https://en.wikipedia.org/wiki/Data_model)

When applied to networking, we can apply this concept as an abstract model that describes our network, be it datacenter, campus, or global Wide Area Network. If we take a closer look at a physical datacenter, a layer 2 Ethernet switch can be thought of as a device containing a table of Mac addresses mapped to each ports. Our switch data model describes how the Mac address should be kept in a table, which includes the keys, additional characteristics (think of VLAN and private VLAN), and more. Similarly, we can move beyond devices and map the datacenter in a model. We can start with the number of devices in each of the access, distribution, core layer, how they are connected, and how they should behave in a production environment. For example, if we have a Fat-Tree network, how many links should each of the spine routers have, how many routes they should contain, and how many next-hops should each of the prefixes have. These characteristics can be mapped out in a format that can be referenced against the ideal state that we should always check against.

One of the relatively new network data modeling language that is gaining traction is YANG. **Yet Another Next Generation** (**YANG**) (despite common belief, some of the IETF work group do have a sense of humor). It was first published in RFC 6020 in 2010, and has since gained traction among vendors and operators. At the time of writing this book, the support for YANG varied greatly from vendors to platforms. The adaptation rate in production is therefore relatively low. However, it is a technology worth keeping an eye out for.

## The Cisco API and ACI

Cisco aystems, as the 800 pound gorilla in the networking space, have not missed on the trend of network automation. The problem has always been the confusion surrounding Cisco's various product lines and the level of technology support. With product lines spans from routers, switches, firewall, servers (unified computing), wireless, the collaboration software and hardware, and analytic software, to name a few, it is hard to know where to start.

Since this book focuses on Python and networking, we will scope the section to the main networking products. In particular, we will cover the following:

- Nexus product automation with NX-API
- Cisco NETCONF and YANG examples
- The Cisco application-centric infrastructure for the datacenter
- The Cisco application-centric infrastructure for the enterprise

For the NX-API and NETCONF examples here, we can either use the Cisco DevNet always-on lab devices or locally run Cisco VIRL. Since ACI is a separate product and is licensed on top of the physical switches, for the following ACI examples, I would recommend using the DevNet labs to get an understanding of the tools, unless, of course, you are one of the lucky ones who has a private ACI lab that you can use.

# Cisco NX-API

Nexus is Cisco's product line of datacenter switches. NX-API (http://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus9000/sw/6-x/programmability/guide/b_Cisco_Nexus_9000_Series_NX-OS_Programmability_Guide/b_Cisco_Nexus_9000_Series_NX-OS_Programmability_Guide_chapter_011.html) allows the engineer to interact with the switch outside of the device via a variety of transports including SSH, HTTP, and HTTPS.

## Lab Software Installation and Device Preparation

Here are the Ubuntu packages that we will install, you may already have some of the packages such as Python development, pip, and Git:

```
$ sudo apt-get install -y python3-dev libxml2-dev libxslt1-dev libffi-dev libssl-
dev zlib1g-dev python3-pip git python3-requests
```

*If you are using Python 2, use the following packages instead:*`sudo apt-get install -y python-dev libxml2-dev libxslt1-dev libffi-dev libssl-dev zlib1g-dev python-pip git python-requests`

The ncclient (https://github.com/ncclient/ncclient) library is a Python library for NETCONF clients, so we will install this from the GitHub repository to install the latest version:

```
$ git clone https://github.com/ncclient/ncclient
$ cd ncclient/
$ sudo python3 setup.py install
$ sudo python setup.py install
```

NX-API on Nexus devices is off by default, so we will need to turn it on. We can either use the user that is already created or create a new user for the NETCONF procedures:

```
feature nxapi
username cisco password 5 $1$Nk7ZkwH0$fyiRmMMfIheqE3BqvcL0C1 role network-opera
tor
username cisco role network-admin
username cisco passphrase lifetime 99999 warntime 14 gracetime 3
```

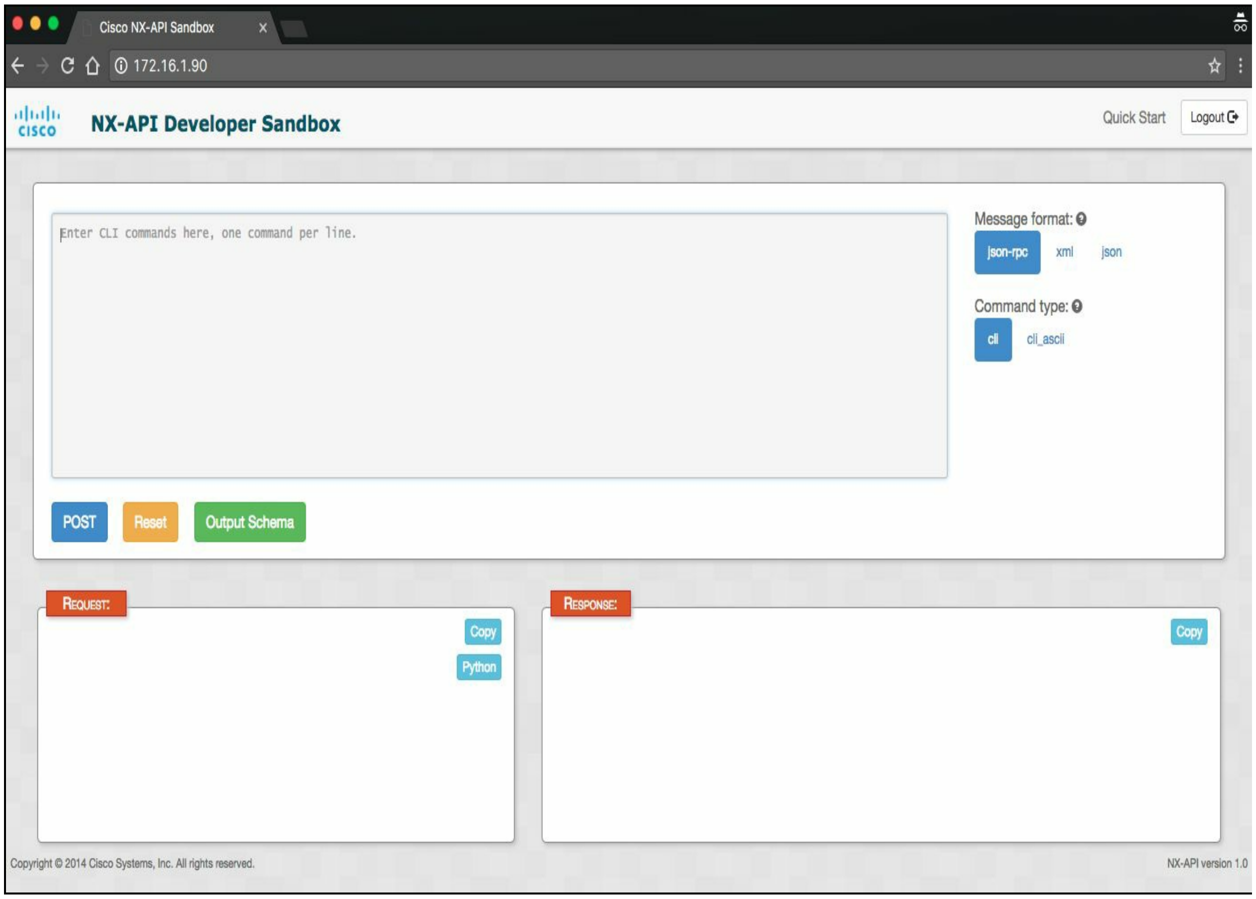
For our lab, we will turn on both HTTP and the sandbox configuration, as they should be turned off in production:

```
nx-osv-2(config)# nxapi http port 80
nx-osv-2(config)# nxapi sandbox
```

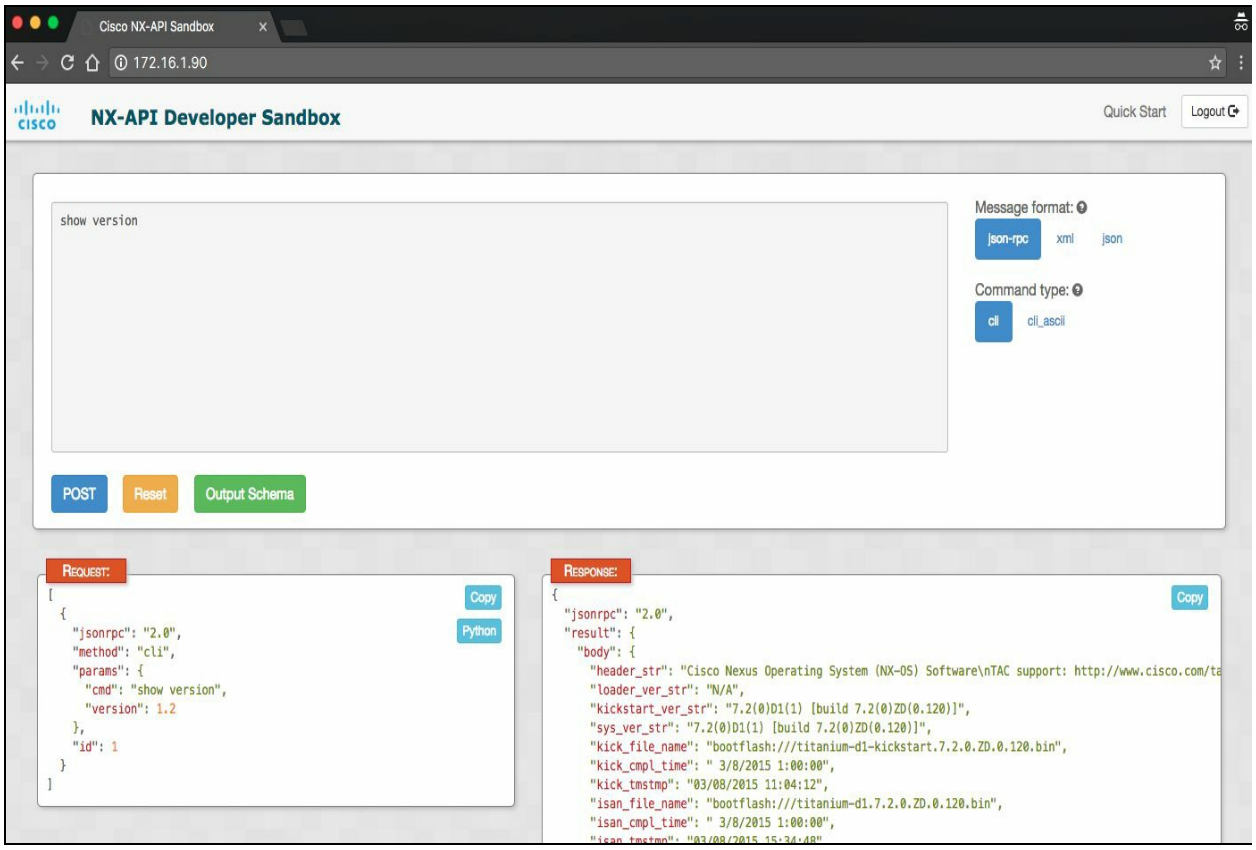
We are now ready to look at our first NX-API example.

## NX-API examples

Since we have turned on sandbox, we can launch a web browser and take a look at the various message formats, requests, and responses based on the CLI command that we are already familiar with:



In the following example, I have selected JSON-RPC and CLI command type for the command called `show version`:



```
show version
```

```
[
  {
    "jsonrpc": "2.0",
    "method": "cli",
    "params": {
      "cmd": "show version",
      "version": 1.2
    },
    "id": 1
  }
]
```

```
{
  "jsonrpc": "2.0",
  "result": {
    "body": {
      "header_str": "Cisco Nexus Operating System (NX-OS) Software\nTAC support: http://www.cisco.com/ta
      "loader_ver_str": "N/A",
      "kickstart_ver_str": "7.2(0)D1(1) [build 7.2(0)ZD(0.120)]",
      "sys_ver_str": "7.2(0)D1(1) [build 7.2(0)ZD(0.120)]",
      "kick_file_name": "bootflash:///titanium-d1-kickstart.7.2.0.ZD.0.120.bin",
      "kick_cmpl_time": " 3/8/2015 1:00:00",
      "kick_tmstmp": "03/08/2015 11:04:12",
      "isan_file_name": "bootflash:///titanium-d1.7.2.0.ZD.0.120.bin",
      "isan_cmpl_time": " 3/8/2015 1:00:00",
      "isan_tmstmp": "03/08/2015 15:34:48"
```

The sandbox comes in handy if you are unsure about the supportability of message format, or if you have questions about the field key for which the value you want to retrieve in your code.

In our first example, we are just going to connect to the Nexus device and print out the capabilities exchanged when the connection was first made:

```
#!/usr/bin/env python3
from ncclient import manager
conn = manager.connect(
        host='172.16.1.90',
        port=22,
        username='cisco',
        password='cisco',
        hostkey_verify=False,
        device_params={'name': 'nexus'},
        look_for_keys=False)
for value in conn.server_capabilities:
    print(value)
conn.close_session()
```

The connection parameters of host, port, username, and password are pretty self explanatory. The device parameter specifies the kind of device the client is connecting to. We will also see a differentiation in the Juniper NETCONF sections. The `hostkey_verify` bypass the `known_host` requirement for SSH, otherwise the host needs to be listed in the `~/.ssh/known_hosts` file. The`look_for_keys` option disables public-private key authentication but uses a username and password for authentication.

The output will show that the XML and NETCONF supported feature by this version of NX-OS:

```
$ python3 cisco_nxapi_1.py
urn:ietf:params:xml:ns:netconf:base:1.0
urn:ietf:params:netconf:base:1.0
```

Using ncclient and NETCONF over SSH is great because it gets us closer to the native implementation and syntax. We will use the library more later on. For NX-API, I personally feel that it is easier to deal with HTTPS and JSON-RPC. In the earlier screenshot of NX-API Developer Sandbox, if you noticed in the Request box, there is a box labeled Python. If you click on it, you will be able to get an automatically converted Python script based on the request library.

*Requests is a very popular, self-proclaimed HTTP for the human library used by companies like Amazon, Google, NSA, and more. You can find more information about it on the official site (http://docs.python-requests.org/en/master/).*

For the `show version` example, the following Python script is automatically generated for you. I am pasting in the output without any modification:

```
"""
 NX-API-BOT
"""
import requests
import json

"""
Modify these please
"""
url='http://YOURIP/ins'
switchuser='USERID'
switchpassword='PASSWORD'

myheaders={'content-type':'application/json-rpc'}
payload=[
    {
      "jsonrpc": "2.0",
      "method": "cli",
      "params": {
        "cmd": "show version",
        "version": 1.2
      },
      "id": 1
    }
  ]
  response = requests.post(url,data=json.dumps(payload),
  headers=myheaders,auth=(switchuser,switchpassword)).json()
```

In `cisco_nxapi_2.py` file, you will see that I have only modified the URL, username, and password of the preceding file, and I have parsed the output to only include the software version. Here is the output:

```
$ python3 cisco_nxapi_2.py
7.2(0)D1(1) [build 7.2(0)ZD(0.120)]
```

The best part about using this method is that the same syntax works with both configuration command as well as show commands. This is illustrated in the `cisco_nxapi_3.py` file. For multi-line configuration, you can use the id field to specify the order of operations. In `cisco_nxapi_4.py`, the following payload was listed for changing the description of the interface Ethernet 2/12 in the interface configuration mode:

```
    {
      "jsonrpc": "2.0",
      "method": "cli",
      "params": {
        "cmd": "interface ethernet 2/12",
        "version": 1.2
      },
      "id": 1
    },
    {
      "jsonrpc": "2.0",
      "method": "cli",
      "params": {
        "cmd": "description foo-bar",
        "version": 1.2
      },
      "id": 2
    },
    {
      "jsonrpc": "2.0",
      "method": "cli",
      "params": {
        "cmd": "end",
        "version": 1.2
      },
      "id": 3
    },
    {
      "jsonrpc": "2.0",
      "method": "cli",
      "params": {
        "cmd": "copy run start",
        "version": 1.2
      },
      "id": 4
    }
  ]
```

In the next section, we will look at the examples for Cisco NETCONF and the YANG model.

## Cisco and YANG model

Earlier in the chapter, we looked at the possibility of expressing the network using data modeling language YANG. Let's look into it a little bit.

First off, we should know that YANG only defines the type of data sent over NETCONF protocol, and NETCONF exists as a standalone protocol as we saw in the NX-API section. YANG, being relatively new, has a spotty supportability across vendors and product lines. For example, if we run the same capability exchange script that we have used before to a Cisco 1000v running IOS-XE, this is what we will see:

```
urn:cisco:params:xml:ns:yang:cisco-virtual-service?module=cisco-
virtual-service&revision=2015-04-09
http://tail-f.com/ns/mibs/SNMP-NOTIFICATION-MIB/200210140000Z?
module=SNMP-NOTIFICATION-MIB&revision=2002-10-14
urn:ietf:params:xml:ns:yang:iana-crypt-hash?module=iana-crypt-
hash&revision=2014-04-04&features=crypt-hash-sha-512,crypt-hash-
sha-256,crypt-hash-md5
urn:ietf:params:xml:ns:yang:smiv2:TUNNEL-MIB?module=TUNNEL-
MIB&revision=2005-05-16
urn:ietf:params:xml:ns:yang:smiv2:CISCO-IP-URPF-MIB?module=CISCO-
IP-URPF-MIB&revision=2011-12-29
urn:ietf:params:xml:ns:yang:smiv2:ENTITY-STATE-MIB?module=ENTITY-
STATE-MIB&revision=2005-11-22
urn:ietf:params:xml:ns:yang:smiv2:IANAifType-MIB?module=IANAifType-
MIB&revision=2006-03-31
<omitted>
```

Compare this to the output that we saw; clearly IOS-XE understands the YANG model more than NX-OS. Industry wide network data modeling for networking is clearly something that is beneficial to network automation. However, given the uneven support across vendors and products, it is not something that is mature enough to be used across your production network, in my opinion. For this book, I have included a script called `cisco_yang_1.py` that shows how to parse out the NETCONF XML output with YANG filters called `urn:ietf:params:xml:ns:yang:ietf-interfaces` as a starting point to see the existing tag overlay.

*You can check the latest vendor support on the YANG GitHub project page: (https://github.com/YangModels/yang/tree/master/vendor).*

## The Cisco ACI

The Cisco **Application Centric Infrastructure** (**ACI**) is meant to provide a centralized approach to all of the network components. In the datacenter context, it means the centralized controller is aware of and manages the spine, leaf, top of rack switches as well as all the network service functions. This can be done through GUI, CLI, or API. Some might argue that the ACI is Cisco's answer to the broader software-defined networking.

One of the somewhat confusing point for ACI is the difference between ACI and ACI-EM. In short, ACI focuses on datacenter operations while ACI-EM focuses on enterprise modules. Both offer a centralized view and control of the network components, but each has it own focus and share of tools. For example, it is rare to see any major datacenter deploy customer facing wireless infrastructure, but wireless network is a crucial part of enterprises today. Another example would be the different approaches to network security. While security is important in any network, in the datacenter environment, lots of security policies are pushed to the edge node on the server for scalability; in enterprises security, policy is somewhat shared between the network devices and servers.

Unlike NETCONF RPC, ACI API follows the REST model to use the HTTP verb (`GET`, `POST`, `PUT`, `DELETE`) to specify the operation intended.

*We can look at the `cisco_apic_em_1.py` file, which is a modified version of the Cisco sample code on `lab2-1-get-network-device-list.py` (https://github.com/CiscoDevNet/apicem-1.3-LL-sample-codes/blob/master/basic-labs/lab2-1-get-network-device-list.py).*

The abbreviated section without comments and spaces are listed here.

The first function called `getTicket()` uses HTTPS `POST` on the controller with the path called `/api/vi/ticket` with a username and password embedded in the header. Then, parse the returned response for a ticket with some limited valid time:

```
  def getTicket():
      url = "https://" + controller + "/api/v1/ticket"
      payload = {"username":"usernae","password":"password"}
      header = {"content-type": "application/json"}
      response= requests.post(url,data=json.dumps(payload), headers=header,
verify=False)
      r_json=response.json()
      ticket = r_json["response"]["serviceTicket"]
      return ticket
```

The second function then calls another path called `/api/v1/network-devices` with the newly acquired ticket embedded in the header, then parse the results:

```
  url = "https://" + controller + "/api/v1/network-device"
  header = {"content-type": "application/json", "X-Auth-Token":ticket}
```

The output displays both the raw JSON response output as well as a parsed table. A partial output when executed against a DevNet lab controller is shown here:

```
    Network Devices =
    {
     "version": "1.0",
     "response": [
     {
     "reachabilityStatus": "Unreachable",
     "id": "8dbd8068-1091-4cde-8cf5-d1b58dc5c9c7",
     "platformId": "WS-C2960C-8PC-L",
    <omitted>
     "lineCardId": null,
     "family": "Wireless Controller",
     "interfaceCount": "12",
     "upTime": "497 days, 2:27:52.95"
     }
    ]
    }
    8dbd8068-1091-4cde-8cf5-d1b58dc5c9c7 Cisco Catalyst 2960-C Series
     Switches
    cd6d9b24-839b-4d58-adfe-3fdf781e1782 Cisco 3500I Series Unified
    Access Points
    <omitted>
    55450140-de19-47b5-ae80-bfd741b23fd9 Cisco 4400 Series Integrated
    Services Routers
    ae19cd21-1b26-4f58-8ccd-d265deabb6c3 Cisco 5500 Series Wireless LAN
    Controllers
```

As one can see, we only query a single controller device, but we are able to get a high-level view of all the network devices that the controller is aware of. The downside is, of course, the ACI controller only supports Cisco devices at this time.

## The Python API for Juniper networks

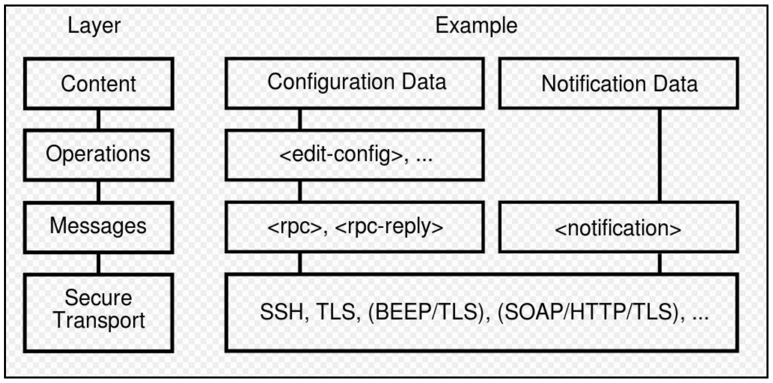
Juniper networks have always been a fan favorite among the service provider crowd. If we take a step back and look at the service provider vertical, it would make sense that automating network equipments is on the top of their mind. Before the dawn of cloud scale datacenters, service providers were the ones with the most network equipment. A typical enterprise might have a few redundant Internet connection at the corporate headquarter, and have a few remote sites homing the network connectivities back to the headquarters in a hub-and-spoke fashion in order to access headquarter resources, such as mail and databases. But to a service provider, they are the ones who need to build, provision, manage, and troubleshoot these connections and the underlying networks. They make their money by selling the bandwidth along with value-added managed services. It would make sense for the service providers to invest in automation to use the least amount of engineering hour to keep the network humming along, and automation is the key.

In my opinion, the difference between a service provider network needs as opposed to their cloud datacenter counterpart is that traditionally, service providers aggregate more into a single device with added services. A good example would be the **Multiprotocol Label Switching** (**MPLS**) that almost all major service providers provide but rarely adapted in the enterprise or datacenter networks. Juniper, as they have been very successful at, has identified this need and excel at fulfilling the service provider requirements of automating. Let's take a look at some of Juniper's automation APIs.

# Juniper and NETCONF

The **Network Configuration Protocol** (**NETCONF**) is an IETF standard, which was first published in 2006 as RFC 4741 and later revised in RFC 6241. Both the RFC's Juniper contributed heavily to the standards; in fact, in RFC, 4741 Juniper was the sole author. It makes sense that Juniper devices fully support NETCONF, and it serves as the underlying layer for most of its automation tools and frameworks. Some of the main characteristics of NETCONF include:

1. It uses **Extensible Markup Language** (**XML**) for data encoding.
2. It uses **Remote Procedure Calls** (**RPC**), therefore in the case of HTTP(s) as the transport, the URL endpoint is identical while the operation intended is specified in the body of the request.
3. It is conceptually based on layers; from top to bottom, they include the content, operations, messages, and transport:



NetConf Model (source: https://en.wikipedia.org/wiki/NETCONF)

Juniper networks provide an extensive NETCONF XML management protocol developer guide (https://www.juniper.net/techpubs/en_US/junos13.2/information-products/pathway-pages/netconf-guide/netconf.html#overview) in its technical library. Let's take a look at its usage.

## Device Preparation

In order to start using NETCONF, let's create a separate user as well as turning on the service:

```
set system login user netconf uid 2001
set system login user netconf class super-user
set system login user netconf authentication encrypted-password
 "$1$0EkA.XVf$cm80A0GC2dgSWJIYWv7Pt1"
set system services ssh
set system services telnet
set system services netconf ssh port 830
```

*For the Juniper device lab, I am using an older, unsupported platform called* ***Juniper Olive****. It is solely used for lab purposes. You can use your favorite search engine to find out some interesting facts and history about Juniper Olive.*

On the Juniper device, you can always take a look at the configuration either in a flat file or in the XML format. The flat file comes in handy when you need to specify a one-liner command to make configuration changes:

```
netconf@foo> show configuration | display set
set version 12.1R1.9
set system host-name foo
set system domain-name bar
<omitted>
```

The XML format comes in handy at times when you need to see the XML structure of the configuration:

```
netconf@foo> show configuration | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.1R1/junos">
 <configuration junos:commit-seconds="1485561328" junos:commit-
localtime="2017-01-27 23:55:28 UTC" junos:commit-user="netconf">
 <version>12.1R1.9</version>
 <system>
 <host-name>foo</host-name>
<domain-name>bar</domain-name>
```

*We have installed the necessary Linux libraries and the ncclient Python library in the Cisco section. If you have not done so, refer back to the section and install the necessary packages.*

We are now ready to look at our first Juniper NETCONF example.

## Juniper NETCONF examples

We will use a pretty straightforward example to execute `show version`. We will name this file `junos_netconf_1.py`:

```
#!/usr/bin/env python3

from ncclient import manager

conn = manager.connect(
    host='192.168.24.252',
    port='830',
    username='netconf',
    password='juniper!',
    timeout=10,
    device_params={'name':'junos'},
    hostkey_verify=False)

result = conn.command('show version', format='text')
print(result)
conn.close_session()
```

All the fields in the script should be pretty self explanatory with the exception of `device_params`. Starting from ncclient 0.4.1, the device handler was added to specify different vendors or platforms, for example, the name can be juniper, CSR, Nexus, or Huawei. We also added `hostkey_verify=False` because we are using a self-signed certificate from the Juniper device.

The returned output is an `rpc-reply` encoded in XML with and `output` element:

```
    <rpc-reply message-id="urn:uuid:7d9280eb-1384-45fe-be48-
    b7cd14ccf2b7">
     <output>
    Hostname: foo
    Model: olive
    JUNOS Base OS boot [12.1R1.9]
    JUNOS Base OS Software Suite [12.1R1.9]
    <omitted>
    JUNOS Runtime Software Suite [12.1R1.9]
    JUNOS Routing Software Suite [12.1R1.9]
    </output>
    </rpc-reply>
```

We can parse the XML output to just include the output text:

```
    print(result.xpath('output')[0].text)
```

In `junos_netconf_2.py`, we will make configuration changes to the device. We will start with some new imports for constructing new XML elements and the connection manager object:

```
  #!/usr/bin/env python3

  from ncclient import manager
  from ncclient.xml_ import new_ele, sub_ele

  conn = manager.connect(host='192.168.24.252', port='830',
username='netconf' , password='juniper!', timeout=10,
device_params={'name':'junos'}, hostkey_v erify=False)
```

We will lock the configuration and make configuration changes:

```
  # lock configuration and make configuration changes
  conn.lock()

  # build configuration
  config = new_ele('system')
  sub_ele(config, 'host-name').text = 'master'
  sub_ele(config, 'domain-name').text = 'python'
```

You can see from the XML display that the node structure with `'system'` is the parent of `'host-name'` and `'domain-name'`:

```
 <system>
    <host-name>foo</host-name>
    <domain-name>bar</domain-name>
...
</system>
```

We can then push the configuration and commit the configuration. These are normal best practice steps for Juniper configuration changes:

```
  # send, validate, and commit config
  conn.load_configuration(config=config)
  conn.validate()
  commit_config = conn.commit()
  print(commit_config.tostring)

  # unlock config
  conn.unlock()

  # close session
  conn.close_session()
```

Overall, the NETCONF steps map pretty well to what you would have done in the CLI steps. Please take a look at the `junos_netconf_3.py` script for a combination of the previous examples and the functional example.

## Juniper PyEZ for developers

**PyEZ** is a high-level Python implementation that integrates better with your existing Python code. By utilizing Python API, you can perform common operation and configuration tasks without the extensive knowledge of the Junos CLI.

*Juniper maintains a comprehensive Junos PyEZ developer guide at https://www.juniper.net/techpubs/en_US/junos-pyez1.0/information-products/pathway-pages/junos-pyez-developer-guide.html#configuration on their technical library. If you are interested in using PyEZ, I would highly recommend at least a glance through the various topics on the guide.*

# Installation and preparation

The installation instruction for each of the operating system can be found on the Installing Junos PyEZ (https://www.juniper.net/techpubs/en_US/junos-pyez1.0/topics/task/installation/junos-pyez-server-installing.html) page. Next, we will show the installation instructions for Ubuntu 16.04.

There are some dependent packages, many of which should already be on the device:

```
$ sudo apt-get install -y python3-pip python3-dev libxml2-dev libxslt1-dev libssl-
dev libffi-dev
```

PyEZ packages can be installed via pip. Here, I have installed for both Python 3 and Python 2:

```
$ sudo pip3 install junos-eznc
$ sudo pip install junos-eznc
```

On the device, NETCONF needs to be configured as the underlying XML API for PyEZ:

```
set system services netconf ssh port 830
```

For user authentication, we can either use password authentication or ssh key pair. Creating the local user is straightforward:

```
set system login user netconf uid 2001
set system login user netconf class super-user
set system login user netconf authentication encrypted-password
"$1$0EkA.XVf$cm80A0GC2dgSWJIYWv7Pt1"
```

For the ssh key authentication, first generate the key pair:

```
$ ssh-keygen -t rsa
```

By default, the public key will be called `id_rsa.pub under ~/.ssh/` while the private key is named `id_rsa` under the same directory. Treat the private key like a password that you would never share. The public key can be freely distributed; however, in this case, we will move it to the `/tmp` directory and enable the Python 3 HTTP server module to create a reachable URL:

```
$ mv ~/.ssh/id_rsa.pub /tmp
$ cd /tmp
$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 ...
```

*For Python 2, use `python -m SimpleHTTPServer` instead.*

At this point, we can create the user and associate the public key:

```
netconf@foo# set system login user echou class super-user authentication load-key-
file http://192.168.24.164:8000/id_rsa.pub
/var/home/netconf/...transferring.file........100% of 394 B 2482 kBps
```

Now, if we try ssh with the private key from the management station, the user will be automatically authenticated:

```
$ ssh -i ~/.ssh/id_rsa 192.168.24.252
--- JUNOS 12.1R1.9 built 2012-03-24 12:52:33 UTC
echou@foo>
```

Let's make sure the authentication methods work with PyEZ. Let's try the username and password combination:

```
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from jnpr.junos import Device
>>> dev = Device(host='192.168.24.252', user='netconf', password='juniper!')
>>> dev.open()
Device(192.168.24.252)
>>> dev.facts
{'serialnumber': '', 'personality': 'UNKNOWN', 'model': 'olive', 'ifd_style':
'CLASSIC', '2RE': False, 'HOME': '/var/home/juniper', 'version_info':
junos.version_info(major=(12, 1), type=R, minor=1, build=9), 'switch_style':
'NONE', 'fqdn': 'foo.bar', 'hostname': 'foo', 'version': '12.1R1.9', 'domain':
'bar', 'vc_capable': False}
>>> dev.close()
```

We can also try to use the ssh key authentication:

```
>>> from jnpr.junos import Device
>>> dev1 = Device(host='192.168.24.252', user='echou',
ssh_private_key_file='/home/echou/.ssh/id_rsa')
>>> dev1.open()
Device(192.168.24.252)
```

```
>>> dev1.facts
{'HOME': '/var/home/echou', 'model': 'olive', 'hostname': 'foo', 'switch_style':
'NONE', 'personality': 'UNKNOWN', '2RE': False, 'domain': 'bar', 'vc_capable':
False, 'version': '12.1R1.9', 'serialnumber': '', 'fqdn': 'foo.bar', 'ifd_style':
'CLASSIC', 'version_info': junos.version_info(major=(12, 1), type=R, minor=1,
build=9)}
>>> dev1.close()
```

Great! We are now ready to look at some examples for PyEZ.

# PyEZ examples

In the previous interactive prompt, we already saw that when the device connects, the object already automatically retrieves a few facts about the device. In our first example, `junos_pyez_1.py`, we are connecting to the device and executing a RPC call for `show interface em1`:

```
    #!/usr/bin/env python3
    from jnpr.junos import Device
    import xml.etree.ElementTree as ET
    import pprint

    dev = Device(host='192.168.24.252', user='juniper', passwd='juniper!')

    try:
        dev.open()
    except Exception as err:
        print(err)
        sys.exit(1)

    result =
dev.rpc.get_interface_information(interface_name='em1', terse=True)
    pprint.pprint(ET.tostring(result))

    dev.close()
```

The device class has a proper of the `rpc` property that includes all operational commands. This is pretty awesome because there is no slippage between what we can do in CLI vs. API. The catch is that we need to find out the `xml rpc` element tag. In our first example, how do we know `show interface em1` equates to `get_interface_information`? We have three ways of finding out about the information:

1. We can reference the *Junos XML API Operational Developer Reference*.
2. We can also use the CLI and display the XML RPC equivalent and replace the dash (-) between the words with underscore (_).
3. We can also do this programmatically using the PyEZ library.

I typically use the second option to get the output directly:

```
netconf@foo> show interfaces em1 | display xml rpc
```

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.1R1/junos">
 <rpc>
 <get-interface-information>
 <interface-name>em1</interface-name>
 </get-interface-information>
 </rpc>
 <cli>
 <banner></banner>
 </cli>
</rpc-reply>
```

However, a third option is also feasible:

```
>>> dev1.display_xml_rpc('show interfaces em1', format='text')
'<get-interface-information>n <interface-name>em1</interface-
name>n</get-interface-information>n'
```

Of course, we will need to make configuration changes as well. In the `junos_pyez_2.py` configuration example, we will import an additional `Config()` method from PyEZ:

```
  #!/usr/bin/env python3
  from jnpr.junos import Device
  from jnpr.junos.utils.config import Config
```

We will utilize the same block for connecting to a device:

```
  dev = Device(host='192.168.24.252', user='juniper',
passwd='juniper!')

  try:
      dev.open()
  except Exception as err:
      print(err)
      sys.exit(1)
```

The `new Config()` method will load the XML data and make the configuration changes:

```
  config_change = """
  <system>
    <host-name>master</host-name>
    <domain-name>python</domain-name>
  </system>
  """

  cu = Config(dev)
  cu.lock()
  cu.load(config_change)
  cu.commit()
  cu.unlock()

  dev.close()
```

The PyEZ examples are simple by design, but hopefully, they demonstrate the ways you can leverage PyEZ for your Junos automation needs.

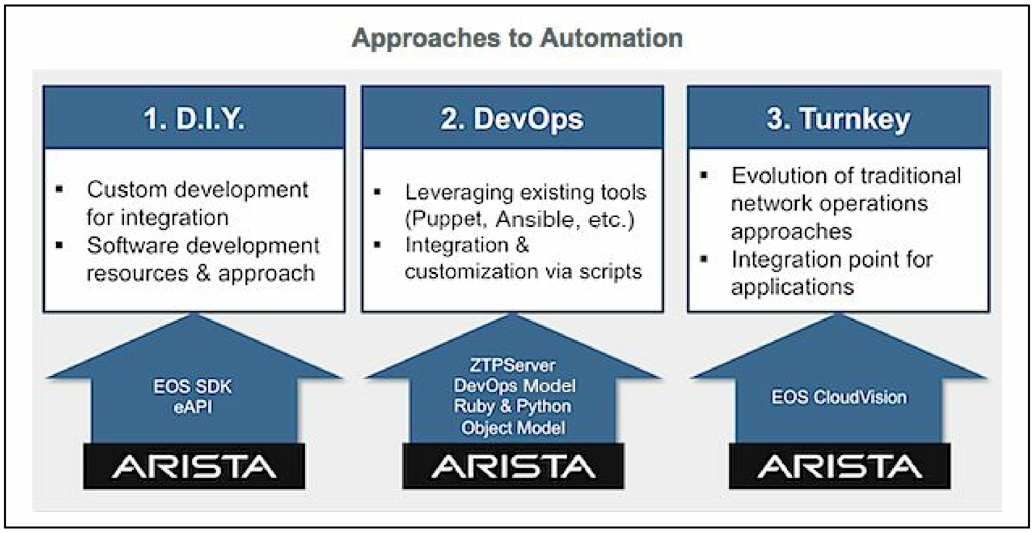
# The Arista Python API

**Arista Networks** have always been focused on large-scale datacenter networks. In its corporate profile page (https://www.arista.com/en/company/company-overview), it is stated as follows:

*"Arista Networks was founded to pioneer and deliver software-driven cloud networking solutions for large data center storage and computing environments."*

Notice that the statement specifically called our **large datacenter**, which we already know is exploded with servers, databases, and yes, network equipment too. Therefore, automation has always been on their mind. In fact, they have a Linux underpin behind their operating system, then EOS has many added benefits such as having Python already built-in along with Linux APIs.

Arista's automation story consist of three approaches:


Approaches to Automation
1. D.I.Y.
Custom development for integration
Software development resources & approach
2. DevOps
Leveraging existing tools (Puppet, Ansible, etc.)
Integration & customization via scripts
3. Turnkey
Evolution of traditional network operations approaches
Integration point for applications
EOS SDK
eAPI
ZTPServer
DevOps Model
Ruby & Python Object Model
EOS CloudVision
ARISTA
ARISTA
ARISTA

EOS Automation (source: https://www.arista.com/en/products/eos/automation)

Like its network counterparts, you can interact with Arista devices directly via eAPI, or you can choose to leverage their Python library for the interaction. We will see examples of both. We will look at Arista's integration for Ansible in later chapters.

## The Arista eAPI management

Arista's eAPI was first introduced in EOS 4.12 a few years ago. It transports a list of show or config commands over HTTP or HTTPS and responses back in JSON. An important distinction is that it is a **Remote Procedure Call** (**RPC**) and JSON-RPC, instead of the representation of state transfer and REST, even though the transport layer is HTTP(S). For our intents and purposes, the difference is that we make the request to the same URL endpoint using the same HTTP method (`POST`). Instead of using HTTP verb (`GET`, `POST`, `PUT`, `DELETE`) to express our action, we simply state our intended action in the body of the request. In the case of eAPI, we will specify a `method` key with a `runCmds` value for our intention.

For the following examples, I am using a physical Arista switch running EOS 4.16.

## The eAPI preparation

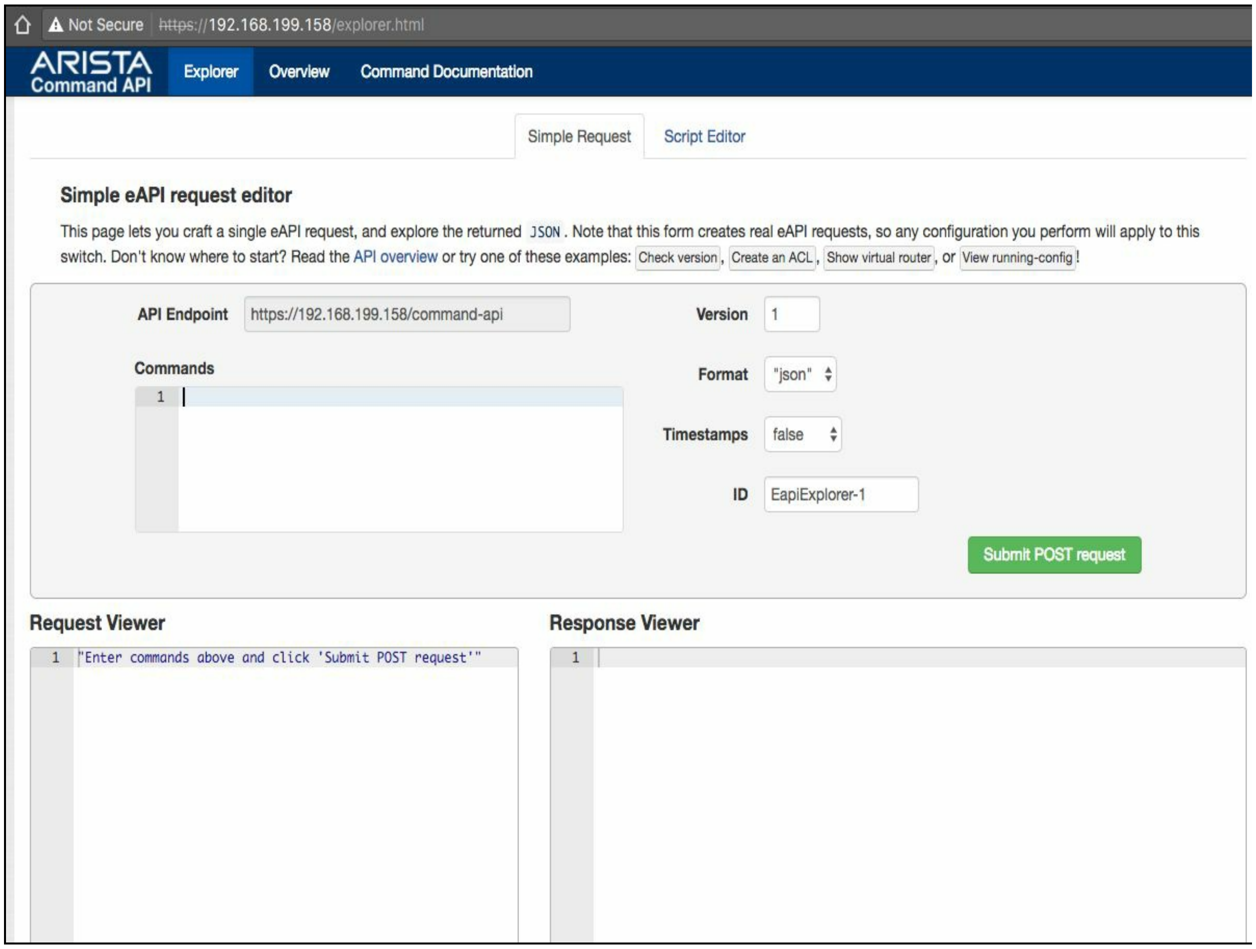
The eAPI agent on the Arista device is disabled by default, so we will need to enable it on the device before we can use it:

```
arista1(config)#management api http-commands
arista1(config-mgmt-api-http-cmds)#no shut
arista1(config-mgmt-api-http-cmds)#protocol https port 443
arista1(config-mgmt-api-http-cmds)#no protocol http
arista1(config-mgmt-api-http-cmds)#vrf management
```

As you can see, we have turned off HTTP server and use HTTPS as the transport instead. Starting from a few EOS version ago, the management interfaces is, by default, residing in a VRF called **management.** In my topology, I am accessing the device via the management interface; therefore, I have specified the VRF specifically. You can check that API management is turned on by the show command:

```
arista1#sh management api http-commands
Enabled: Yes
HTTPS server: running, set to use port 443
HTTP server: shutdown, set to use port 80
Local HTTP server: shutdown, no authentication, set to use port 8080
Unix Socket server: shutdown, no authentication
VRF: management
Hits: 64
Last hit: 33 seconds ago
Bytes in: 8250
Bytes out: 29862
Requests: 23
Commands: 42
Duration: 7.086 seconds
SSL Profile: none
QoS DSCP: 0
 User Requests Bytes in Bytes out Last hit
----------- -------------- -------------- --------------- --------------
 admin 23 8250 29862 33 seconds ago

URLs
-----------------------------------------
Management1 : https://192.168.199.158:443

arista1#
```

After enabling the agent, you will be able to access the exploration page for eAPI by going to the device's IP address. If you have changed the default port for access, just append it at the end. The authentication is tied in to the method of authentication on the switch; in this case the username and password were configured locally. By default, a self-signed certificate will be used.



Simple eAPI request editor

This page lets you craft a single eAPI request, and explore the returned JSON. Note that this form creates real eAPI requests, so any configuration you perform will apply to this switch. Don't know where to start? Read the API overview or try one of these examples: Check version, Create an ACL, Show virtual router, or View running-config!

API Endpoint https://192.168.199.158/command-api

Commands

Version 1

Format "json"

Timestamps false

ID EapiExplorer-1

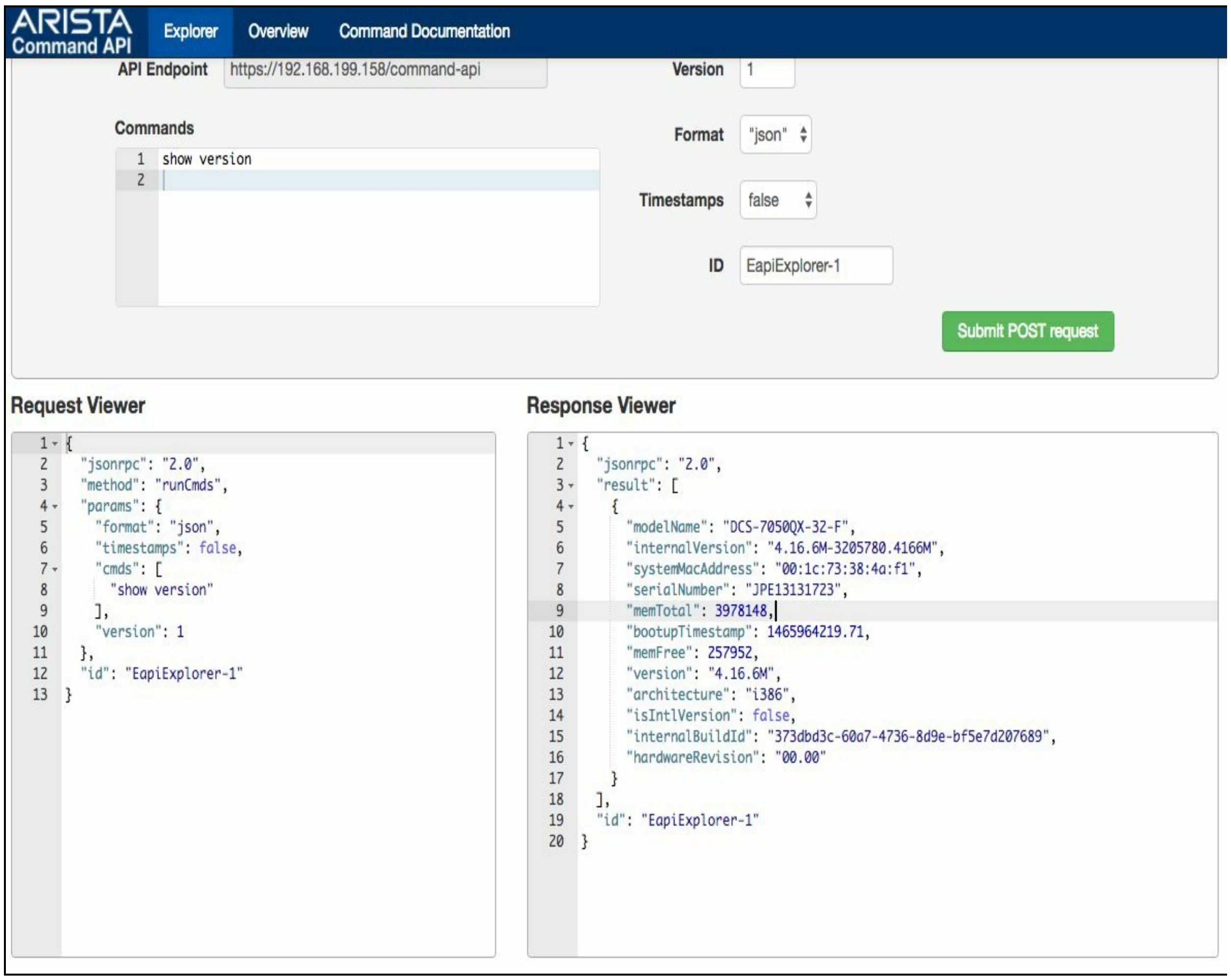
Request Viewer

```
"Enter commands above and click 'Submit POST request'"
```

Response Viewer

Arista EOS Explorer

You will be taken to an explorer page where you can type in the CLI command and get a nice output for the body of your request. For example, if I wanted to see how to make a request body for `show version`, this is the output I will see from the explorer:



API Endpoint https://192.168.199.158/command-api

Version 1

Commands

```
show version
```

Format "json"

Timestamps false

ID EapiExplorer-1

Request Viewer

```
{
  "jsonrpc": "2.0",
  "method": "runCmds",
  "params": {
    "format": "json",
    "timestamps": false,
    "cmds": [
      "show version"
    ],
    "version": 1
  },
  "id": "EapiExplorer-1"
}
```

Response Viewer

```
{
  "jsonrpc": "2.0",
  "result": [
    {
      "modelName": "DCS-7050QX-32-F",
      "internalVersion": "4.16.6M-3205780.4166M",
      "systemMacAddress": "00:1c:73:38:4a:f1",
      "serialNumber": "JPE13131723",
      "memTotal": 3978148,
      "bootupTimestamp": 1465964219.71,
      "memFree": 257952,
      "version": "4.16.6M",
      "architecture": "i386",
      "isIntlVersion": false,
      "internalBuildId": "373dbd3c-60a7-4736-8d9e-bf5e7d207689",
      "hardwareRevision": "00.00"
    }
  ],
  "id": "EapiExplorer-1"
}
```

Arista EOS Explorer Viewer

The overview link will take you to the sample usage and background information while the command documentation will serve as reference points for the show commands. Each of the command references will contain the returned value field name, type, and a brief description. The online reference scripts from Arista uses jsonrpclib (https://github.com/joshmarshall/jsonrpclib/), which is what we will use as well. However, as of this book, it has a dependency of Python 2.6+ and not yet ported to Python 3; therefore, we will use Python 2.7 for the examples.

*By the time you read this book, there might be an updated status. Please read the GitHub pull request (https://github.com/joshmarshall/jsonrpclib/issues/38) and the GitHub README (https://github.com/joshmarshall/jsonrpclib/) for the latest status.*

Installation is straightforward using `easy_install` or `pip`:

```
$ sudo easy_install jsonrpclib
$ sudo pip install jsonrpclib
```

# eAPI examples

We can then write a simple program called `eapi_1.py` to look at the response text:

```
#!/usr/bin/python2

from __future__ import print_function
from jsonrpclib import Server
import ssl

ssl._create_default_https_context = ssl._create_unverified_context

switch = Server("https://admin:arista@192.168.199.158/command-api")

response = switch.runCmds( 1, [ "show version" ] )
print('Serial Number: ' + response[0]['serialNumber'])
```

TIP

*Note since this is Python 2, in the script I used the `from __future__ import print_function` to make future migration easier. The `ssl` related lines are for Python version > 2.7.9, for more information please see* https://www.python.org/dev/peps/pep-0476/.

This is the response I received from the previous `runCms()` method:

```
 [{u'memTotal': 3978148, u'internalVersion': u'4.16.6M-
 3205780.4166M', u'serialNumber': u'<omitted>', u'systemMacAddress':
 u'<omitted>', u'bootupTimestamp': 1465964219.71, u'memFree':
 277832, u'version': u'4.16.6M', u'modelName': u'DCS-7050QX-32-F',
 u'isIntlVersion': False, u'internalBuildId': u'373dbd3c-60a7-4736-
8d9e-bf5e7d207689', u'hardwareRevision': u'00.00', u'architecture':
u'i386'}]
```

As you can see, this is a list containing one dictionary item. If we need to grab the serial number, we can simply reference the item number and the key in line 12:

```
print('Serial Number: ' + response[0]['serialNumber'])
```

The output will contain only the serial number:

```
$ python eapi_1.py
Serial Number: <omitted>
```

To be more familiar with the command reference, I recommend that you click on the Command Documentation link on the eAPI page, and compare your output with the output of version in the documentation.

As noted earlier, unlike REST, JSON-RPC client uses the same URL endpoint for calling the server resources. You can see from the previous example that the `runCmds()` method can consume a list of commands. For the execution of configuration commands, you can follow the same framework, and place the list of commands inside the second argument.

Here is an example of configuration commands called `eapi_2.py`:

```
  #!/usr/bin/python2

  from __future__ import print_function
  from jsonrpclib import Server
  import ssl, pprint

  ssl._create_default_https_context = ssl._create_unverified_context

  # Run Arista commands thru eAPI
  def runAristaCommands(switch_object, list_of_commands):
      response = switch_object.runCmds(1, list_of_commands)
      return response


  switch = Server("https://admin:arista@192.168.199.158/command-
api")

  commands = ["enable", "configure", "interface ethernet 1/3",
"switchport acc ess vlan 100", "end", "write memory"]

 response = runAristaCommands(switch, commands)
 pprint.pprint(response)
```

Here is the output of the command execution:

```
$ python2 eapi_2.py
[{}, {}, {}, {}, {}, {u'messages': [u'Copy completed successfully.']}]
```

Now, do a quick check on the switch to verify command execution:

```
arista1#sh run int eth 1/3
interface Ethernet1/3
    switchport access vlan 100
arista1#
```

Overall, eAPI is fairly straightforward and simple to use. Most programming languages have libraries similar to `jsonrpclib` that abstracts away JSON-RPC internals. With a few commands you can start integrating Arista EOS automation into your network.

## The Arista Pyeapi library

The Python client for the eAPI and Pyeapi (http://pyeapi.readthedocs.io/en/master/index.html) libraries is a native Python library wrapper around eAPI. It provides a set of bindings to configure Arista EOS nodes. Why do we need pyeapi when we already have eAPI? Since we have used Python to demonstrate eAPI, keep in mind that the only requirement of eAPI is a JSON-RPC capable client. Thus, it is compatible with most programming languages. Personally, I think of eAPI as a common denominator among all different languages. When I first started out in the field, Perl was the dominant language for scripting and network automation. There are still many enterprises that rely on Perl scripts as their primary automation tool. Or in a situation where the company already has invested a ton of resources and code base in another language, eAPI with JSON-RPC would be the way to move forward with.

However, for those of us who prefer to code in Python, a native Python library means a more natural feeling in writing our code. It certainly makes extending a Python program to support the EOS node easier. It also makes keeping up with the latest change in Python easier. For example, we can use Python 3 with pyeapi!

*At the time of writing, Python 3 (3.4+) support is officially work-in-progress as stated (http://pyeapi.readthedocs.io/en/master/requirements.html) on the documentation. Please check the documentation.*

# The Pyeapi installation

Installation is straightforward with pip:

```
$ sudo pip install pyeapi
$ sudo pip3 install pyeapi
```

*Note that pip will also install the netaddr library as it is part of the stated requirement (http://pyeapi.readthedocs.io/en/master/requirements.html) for pyeapi.*

By default, pyapi client will look for an INI style hidden (with a period in front) file called `eapi.conf` in your home directory. You can override this behavior by specifying the `eapi.conf` file path, but it is generally a good idea to separate your connection credential and lock it down from the script itself. You can check out the Arista pyapi documentation (http://pyeapi.readthedocs.io/en/master/configfile.html#configfile) for the fields contained in the file; here is the file I am using in the lab:

```
cat ~/.eapi.conf
[connection:Arista1]
host: 192.168.199.158
username: admin
password: arista
transport: https
```

The first line called `[connection:Arista1]` contains the name that we will use in our pyeapi connection; the rest of the fields should be pretty self-explanatory. You can lock down the file to be read-only for the user that is going call this file:

```
$ chmod 400 ~/.eapi.conf
$ ls -l ~/.eapi.conf
-r-------- 1 echou echou 94 Jan 27 18:15 /home/echou/.eapi.conf
```

## Pyeapi examples

Now, we are ready to take a look around the usage. Let's start by connecting to the EOS node by creating an object:

```
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyeapi
>>> arista1 = pyeapi.connect_to('Arista1')
```

We can execute show commands to the node and receive the output back:

```
>>> import pprint
>>> pprint.pprint(arista1.enable('show hostname'))
[{'command': 'show hostname',
 'encoding': 'json',
 'result': {'fqdn': 'arista1', 'hostname': 'arista1'}}]
```

The configuration field can be either a single command or a list of commands using the `config()` method:

```
>>> arista1.config('hostname arista1-new')
[{}]
>>> pprint.pprint(arista1.enable('show hostname'))
[{'command': 'show hostname',
 'encoding': 'json',
 'result': {'fqdn': 'arista1-new', 'hostname': 'arista1-new'}}]
>>> arista1.config(['interface ethernet 1/3', 'description my_link'])
[{}, {}]
```

Note that command abbreviation (`show run` versus `show running-config`) and some extensions would not work:

```
>>> pprint.pprint(arista1.enable('show run'))
Traceback (most recent call last):
...
 File "/usr/local/lib/python3.5/dist-packages/pyeapi/eapilib.py", line 396, in send
 raise CommandError(code, msg, command_error=err, output=out)
pyeapi.eapilib.CommandError: Error [1002]: CLI command 2 of 2 'show run' failed:
invalid command [incomplete token (at token 1: 'run')]
>>>
>>> pprint.pprint(arista1.enable('show running-config interface ethernet 1/3'))
Traceback (most recent call last):
...
pyeapi.eapilib.CommandError: Error [1002]: CLI command 2 of 2 'show running-config
interface ethernet 1/3' failed: invalid command [incomplete token (at token 2:
'interface')]
```

However you can always catch the results back and get the desired value:

```
>>> result = arista1.enable('show running-config')
>>> pprint.pprint(result[0]['result']['cmds']['interface Ethernet1/3'])
{'cmds': {'description my_link': None, 'switchport access vlan 100': None},
'comments': []}
```

So far, we have been doing what we have been doing with eAPI for show and configuration commands. Pyeapi offers various API to make life easier. In the following example, we will connect to the node, call the VLAN API, and start to operate on the VLAN parameters of the device. Let's take a look:

```
>>> import pyeapi
>>> node = pyeapi.connect_to('Arista1')
>>> vlans = node.api('vlans')
>>> type(vlans)
<class 'pyeapi.api.vlans.Vlans'>
>>> dir(vlans)
[...'command_builder', 'config', 'configure', 'configure_interface',
'configure_vlan', 'create', 'default', 'delete', 'error', 'get', 'get_block',
'getall', 'items', 'keys', 'node', 'remove_trunk_group', 'set_name', 'set_state',
'set_trunk_groups', 'values']
>>> vlans.getall()
{'1': {'vlan_id': '1', 'trunk_groups': [], 'state': 'active', 'name': 'default'}}
>>> vlans.get(1)
{'vlan_id': 1, 'trunk_groups': [], 'state': 'active', 'name': 'default'}
>>> vlans.create(10)
True
>>> vlans.getall()
{'1': {'vlan_id': '1', 'trunk_groups': [], 'state': 'active', 'name': 'default'},
'10': {'vlan_id': '10', 'trunk_groups': [], 'state': 'active', 'name': 'VLAN0010'}}
>>> vlans.set_name(10, 'my_vlan_10')
True
```

Let's verify VLAN 10 that we created via the VLAN API on the device:

```
arista1#sh vlan
VLAN Name Status Ports
----- -------------------------------- --------- -------------------------------
1 default active
10 my_vlan_10 active
```

As you can see, the Python native API on the EOS object is really where the pyeapi excel beyond eAPI as it abstracts the lower level attributes into the device object.

*For a full list of ever increasing pyeapi APIs, check the official documentation (http://pyeapi.readthedocs.io/en/master/api_modules/_list_of_modules.html).*

To round up this chapter, let's assume we repeat the previous steps enough that we would like to write another Python class to save us some work called `pyeapi_1.py`:

```
        #!/usr/bin/env python3

        import pyeapi

        class my_switch():

            def __init__(self, config_file_location, device):
                 # loads the config file
                 pyeapi.client.load_config(config_file_location)
                 self.node = pyeapi.connect_to(device)
                 self.hostname = self.node.enable('show hostname')[0]
      ['result']['host name']
                self.running_config = self.node.enable('show running-
      config')

             def create_vlan(self, vlan_number, vlan_name):
                 vlans = self.node.api('vlans')
                 vlans.create(vlan_number)
                 vlans.set_name(vlan_number, vlan_name)
```

As you can see, we automatically connect to the node and set the hostname and `running_config` upon connection. We also create a method to the class that creates VLAN using the VLAN API. Let's try it out:

```
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyeapi_1
>>> s1 = pyeapi_1.my_switch('/tmp/.eapi.conf', 'Arista1')
>>> s1.hostname
'arista1'
>>> s1.running_config
[{'encoding': 'json', 'result': {'cmds': {'interface Ethernet27': {'cmds': {},
'comments': []}, 'ip routing': None, 'interface face Ethernet29': {'cmds': {},
'comments': []}, 'interface Ethernet26': {'cmds': {}, 'comments': []}, 'interface
Ethernet24/4': h.':
<omitted>
'interface Ethernet3/1': {'cmds': {}, 'comments': []}}, 'comments': [], 'header':
['! device: arista1 (DCS-7050QX-32, EOS-4.16.6M)n!n']}, 'command': 'show running-
config'}]
>>> s1.create_vlan(11, 'my_vlan_11')
>>> s1.node.api('vlans').getall()
{'11': {'name': 'my_vlan_11', 'vlan_id': '11', 'trunk_groups': [], 'state':
'active'}, '10': {'name': 'my_vlan_10', 'vlan_id': '10', 'trunk_groups': [],
'state': 'active'}, '1': {'name': 'default', 'vlan_id': '1', 'trunk_groups': [],
'state': 'active'}}
>>>
```

## Vendor neutral libraries

There are several efforts of vendor neutral libraries such as Netmiko (https://github.com/ktbyers/netmiko) and napalm (https://github.com/napalm-automation/napalm). Because these libraries do not come natively from the device vendor, they are sometimes a step slower to support the latest platform or features. They are also community supported, which means they are not necessarily the ideal fit if you need to rely on somebody else to fix bugs without any measure of service level commitment.

## Summary

In this chapter, we looked at various ways to communicate and manage network devices from Cisco, Juniper, and Arista. We looked at both direct communication with the likes of NETCONF and REST, as well as using vendor provided libraries such as PyEZ and Pyeapi. These are different layers of abstractions meant to provide a way to programmatically manage your network devices without human intervention.

In the next chapter, we will take a look at a higher level of vendor-neutral abstraction tool called **Ansible**. Ansible is an open source, general purpose automation tool written in Python. It can be used to automate servers, network devices, load balancers, and many more. Of course, for our purpose, we will focus on using this automation framework for network devices.

# The Python Automation Framework - Ansible Basics

The previous two chapters incrementally introduced different ways to interact with the network devices. In Chapter 2, *Low Level Network Device Interactions*, we discussed Pexpect and Paramiko that manage an interactive session automatically. In Chapter 3,Â *API and Intent-Driven Networking*, we saw that networking is not just about individual devices, as it is a way for us to connect individual parts together, but our overall goal typically requires a higher business logic. For example, imagine this hierarchy of thought process from forming our business objective to actually perform an action on a networking device (with the top being close to the business logic): we looked at various APIs that provide a structured way of feedback from device as well as some well-defined command structure. Both of the methods are fairly low level, meaning they are focusedÂ on the individual device that we are performing our action on.

You wake up one morning and think to yourself that you have valuable information on your network, so you should put some secure measure around the network devices to make them more secure!

You then proceed to break the objective down into two actionable items to begin with:Â

- Upgrading the devices to the latest version of software, which requires:Â
  1. Uploading the image to the device.Â
  2. Telling the device to boot from the new image.Â
  3. We will proceed to reboot the device.Â
  4. Verify that the device is running with the new software image.
- Configuring the appropriate access control list on the networking devices, which includes the following:
  1. Constructing the access list on the device.Â

2. Configuring the access list on the interface, which in most cases is under the interface configuration section to be applied to theÂ interfaces.Â

Being an automation-minded Python engineer, you proceed to write the scripts using Pexpect, Paramiko, or API. But, as you can see, any network task represents a purpose with the actual execution at the tail end of the operation. For a long time, the engineer will translate the business logic into network commands, then we document them into standard operating procedures, so we can refer back to the process either for ourselves or our colleagues. Â

In this chapter and the next, we will start to discuss an open source automation tool called **Ansible.** It is a toolÂ that can simplify the process of going from business logic to network commands. ItÂ can configure systems, deploy software, and orchestrate a combination of tasks. Ansible is written in Python and has emerged as one of the leading automation tools supported by network equipment vendors.Â

In this chapter, we will take a look at the following topics:Â

- A quick Ansible example
- The advantages of Ansible
- The Ansible architecture
- Ansible Cisco modules and examples
- Ansible Juniper modules and examples
- Ansible Arista modules and examples

At the time of writing this book, Ansible release 2.2 was compatible with Python 2.6 and 2.7 with a technical review for Python 3 support. However, just like Python, many of the useful features of Ansible come from the community-driven extension modules. After the core module is tested stable with Python 3 (hopefully in the near future), it will take some time to bring all the extension modules up from Python 2 to Python 3. For the rest of the book, we will use Python 2.7 with Ansible.

*For the latest information on Ansible Python 3 support, check* *outÂ http://docs.ansible.com/ansible/python_3_support.html.Â*

As one can tell from the previous chapters, I am a believer in learning by examples. Just like the underlying Python code for Ansible, the syntax for Ansible constructs are easy enough to understand even if you have not worked with Ansible before. If you have some experience with YAML or Jinja2, you will quickly draw the correlation between the syntax and the intended procedure. Let's take a look at an example first.Â

## A quick Ansible example

As with other automation tools, Ansible started out first by managing servers before expanding to manage the networking equipment. For the most part, the modules and what Ansible refers to as the playbook are the same between server modules and network modules, but there are still some differences. Therefore, it is helpful to see the server example first and draw comparisons later on when we start to look at network modules.

## The control node installation

First, let us clarify the terminology we will use in the context of Ansible. We will refer to the virtual machine with Ansible installed as the control machine, and the machine is being managed as the target machines or managed nodes. Ansible can be installed on most of the Unix systems with the only dependency of Python 2.6 or 2.7; the current Windows is not supported as the control machine. Windows host can still be managed by Ansible, as they are just not supported as the control machine.

*As Windows 10 starts to adapt Ubuntu Linux, Ansible might soon be ready to run on Windows as well.*

On the managed node requirements, you may notice that Python 2.4 or later is a requirement. This is true for managing target nodes with operating systems such as Linux, but obviously, not all network equipment support Python. We will see how this requirement is bypassed for networking modules.

*For Windows machine, the underlying modules are written in PowerShell, Microsoft's automation, and the configuration management framework.*

We will be installing Ansible on our Ubuntu virtual machine, and for instructions on installation on other operating systems, check out the installation document (http://docs.ansible.com/ansible/intro_installation.html). Following you will see the steps to install the software packages.

```
$ sudo apt-get install software-properties-common
$ sudo apt-add-repository ppa:ansible/ansible
$ sudo apt-get update
$ sudo apt-get install ansible
```

*You might be tempted to just use `pip install ansible` for installation, but this will give you some problem down the line.*

*Follow the installation instruction on the Ansible documentation page for your operating system.*

You can now do a quick verification as follows:

```
$ ansible --version
ansible 2.2.1.0
 config file = /etc/ansible/ansible.cfg
 configured module search path = Default w/o overrides
```

Now, we are ready to see an example.

# Your first Ansible playbook

Our first playbook will include the following steps:

1. Make sure the control node can use key-based authorization.
2. Create an inventory File.
3. Create a playbook.
4. Execute and test it.

## The Public key authorization

The first thing to do is to copy your SSH public key from your control machine to the target machine. A full public key Infrastructure tutorial is outside the scope of this book, but here is a quick walk through on the control node:

```
$ ssh-keygen -t rsa <<<< generates public-private key pair on the host machine
$ cat ~/.ssh/id_rsa.pub <<<< copy the content of the output and paste it to the
~/.ssh/authorized_keys file on the target host
```

*You can read more about PKI on https://en.wikipedia.org/wiki/Public_key_infrastructure.*

Because we are using key-based authentication, we can turn off password-based authentication on the remote node and be more secure. You will now be able to `ssh` from control node to remote node without using the private key and without being prompted for a password.

*Can you automate the initial public key copying? It is possible but is highly depended on your use case, regulation, and environment. It is comparable to the initial console setup for network gears to establish initial IP reachability. Do you automate this? Why or why not?*

# The inventory file

We do not need Ansible if we have no remote target to manage, correct? Everything starts from the fact that we need to manage some task on a remote host. The way we specify the potential remote target is with a host file. We can either have this file specifying the remote host either in `/etc/ansible/hosts` or use the `-i` option to specify the location of the file. Personally, I prefer to have this file in the same directory where my playbook is.

*Technically, this file can be named anything as long as it is in a valid format. However, you can potentially save yourself and your colleagues some headaches in the future by following this convention.*

The inventory file is a simple, plain text INI-style file that states your target. By default, this can either be a DNS FQDN or an IP address:

```
$ cat hosts
192.168.199.170
```

We can now use the command-line option to test Ansible and the host file:

```
$ ansible -i hosts 192.168.199.170 -m ping
192.168.199.170 | SUCCESS => {
 "changed": false,
 "ping": "pong"
}
```

*By default, Ansible assumes that the same user executing the playbook exists on the remote host. For example, I am executing the playbook as `echou` locally; the same user also exist on my remote host. If you want to execute as a different user, you can use the `-u` option when executing, that is `-u REMOTE_USER`.*

The previous line reads that we will use the host file as the inventory file, and execute the `ping` module on the host called `192.168.199.170`. Ping (http://docs.ansible.com/ansible/ping_module.html) is a trivial test module that connects to the remote host, verifies a usable Python installation, and return the output `pong` upon success.

*You may take a look at all the ever expanding module list (http://docs.ansible.com/ansible/list_of_all_modules.html) if you have any questions about the usage of existing modules that were shipped with Ansible.*

If you get a host key error, it is typically because the host key is not in the `known_hosts` file, and is typically under `~/.ssh/known_hosts`. You can either ssh to the host and answer `yes` at adding the host, or you can disable this checking on `/etc/ansible/ansible.cfg` or `~/.ansible.cfg` with the following code:

```
[defaults]
host_key_checking = False
```

Now that we have a valid inventory file, we can make our first playbook.

## Our first playbook

Playbooks are Ansible's blueprint to describe what you would like to be done to the hosts using modules. It is where we will be spending the majority of our time as operators when working with Ansible. If you are building a tree house, the playbook will be your manual, the modules will be your tools, while the inventory will be the components that you will be working on using the tools.

The playbook is designed to be human readable in the YAML format. We will look at the common syntax and usage in the Ansible architecture section. For now, our focus is to run an example to get the look and feel of Ansible.

*Originally, YAML was said to mean, Yet Another Markup Language, but now,*yaml.org (http://yaml.org/) *has repurposed the acronym to be YAML Ain't Markup Language for its data-driven nature.*

Let's look at this simple 6-line playbook, `df_playbook.yml`:

```
---
- hosts: 192.168.199.170

  tasks:
    - name: check disk usage
      shell: df > df_temp.txt
```

In any given playbook, you can contain multiple plays. In this case, we have one play (line 2 - 6). In this play, there can be multiple tasks, and we also just have one task (line 4 - 6) with the name specifying the purpose of the task and the `shell` module taking an argument of `df`. The `shell` module takes the command in the argument and executes it on the remote host. In this case, we execute the `df` command to check the disk usage and copy the output to a file named `df_temp.txt`.

We can execute the playbook via the following code:

```
$ ansible-playbook -i hosts df_playbook.yml
PLAY [192.168.199.170] *********************************************************

TASK [setup] *******************************************************************
ok: [192.168.199.170]

TASK [check disk usage] ********************************************************
changed: [192.168.199.170]

PLAY RECAP *********************************************************************
192.168.199.170 : ok=2 changed=1 unreachable=0 failed=0
```

If you log into the managed host (`192.168.199.170` for me), you will see that the `df_temp.txt` file will be there with the `df` command output. Neat, huh?

Noticed that there were actually two tasks executed even though we only specified one task in the playbook; the setup module is automatically added by default. It is executed by Ansible to gather information about the remote host that can be used later on in the playbook. For example, one of the facts that the setup module gathers is the operating system. The playbook can later on specify to use `apt` for Debian-based hosts and `yum` for Red Hat-based hosts to install new packages.

*If you are curious about the output of a setup module, you can find out what information Ansible gathers via `$ ansible -i hosts <host> -m setup`.*

Underneath the hood, there are actually a few things that has happened for tasks such as copy over the Python script, copy the script output to a temporary file, then capture the output and delete the temporary file. For now, we can probably safely ignore these underlying details until we need them.

It is important that we fully understand the simple process that we have just gone through, because we will be referring back to these elements later in the chapter. I purposely chose a server example to be presented here, because they will make more sense as we dive into the networking modules when we need to deviate from them (remember we mentioned Python interpreter is most likely not on the network gear?).

Congratulations on executing your first Ansible playbook! We will look more into the Ansible architecture, but for now, let's take a look at why Ansible is a good fit for network management. Remember that Ansible is written in Python so once we are more familiar with the framework, we can use Python to extend it.

## The advantages of Ansible

There are many infrastructure automation framework besides Ansible, namely Chef, Puppet, and SaltStack. Each framework offers its own unique features and models, and there is no right tool that fits all the organizations. In this section, I would like to list out some of the advantages of Ansible and why I think this is a good tool for network automation.

From a network engineering point of view, rather than any one point below, I believe it is the combination of all the following reasons that makes Ansible ideal for network automation.

## Agentless

Unlike some of its peers, Ansible does not require a strict master-client model. There is no software or agent to be installed on the client that communicates back to the server. As you can see from the previous example, instead of relying on remote host agents, Ansible uses SSH to push its changes to the remote host. This is huge for network device management, as network vendors are typically reluctant to put third-party software on their platforms. SSH, on the other hand, already exist on the network equipment. This mentality has changed a bit in the last few years, but overall, SSH is the common denominator for all network equipment while configuration management agent support is not.

Because there is no agent on the remote host, Ansible uses a push model to push the changes to the device, as opposed to the pull model where the agent pulls the information from the master server. The push model, in my opinion, is more deterministic as everything originates from the control machine.

Again, the importance of being agentless cannot be stressed enough when it comes to working with the existing network equipment. This is usually one of the major the reasons network operators and vendors embrace Ansible.

## Idempotent

According to Wikipedia, Idempotence is the property of certain operations in mathematics and computer science that can be applied multiple times without changing the result beyond the initial application (https://en.wikipedia.org/wiki/Idempotence). In more common terms, it means running the same procedure over and over again does not change the system after the first time. Ansible aims to be idempotent, which is good for network operations that require certain order of operations.

The advantage of idempotence is best compared to the Pexpect and Paramiko scripts that we have written. Remember that these scripts were written to push out commands as if an engineer is sitting at the terminal. If you were to execute the script 10 times, there will be 10 times that the script makes changes. The Pexpect and Paramiko approach, compare to Ansible playbook that accomplishes the same change, the existing device configuration will be checked first, and the playbook will only execute if the changes does not exist.

Being idempotent means we can repeatedly execute the playbook without worrying that there will be unnecessary changes being made. This would be important as we need to automatically check for state consistency without any extra overhead.

## Simple and extensible

Ansible is written in Python and uses YAML for playbook language, both of which are considered relatively easy to learn. Remember Cisco IOS syntax? That is a domain-specific language that is only applicable when you are managing Cisco IOS devices or other similarly structured equipment; it is not a general purpose language beyond its limited scope. Luckily, unlike some other automation tools, there is no extra domain-specific language or DSL to learn for Ansible because YAML and Python are both widely used as general purpose languages.

As you can see from the previous example, even if you have not seen YAML before, it is easy to accurately guess what the playbook is trying to do. Ansible also uses Jinja2 as a template engine, which is a common tool used by Python web frameworks such as Django and Flask, so the knowledge is transferrable.

I cannot stress enough about the extensibility of Ansible. As illustrated by the example, Ansible starts out with automating servers (primarily Linux) in mind. It then branches out to manage Windows machines with PowerShell. As more and more people in the industry start to adapt Ansible, network became a topic that started to get more attention. The right people and team were hired and adopted at Ansible, network professionals starts to get involved, and customers started to demand vendors for support. Starting from Ansible 2.0; networking has become a first class citizen alongside server management.The ecosystem is alive and well.

Just like the Python community, the Ansible community is friendly and the attitude is inclusive of new member and ideas. I have first hand experience of being a noob and try to make sense of contributions and write modules. I can testify to the fact that I felt welcomed and respected for my opinion at all times.

The simplicity and extensibility really speaks well for future proofing. The technology world is evolving fast, and we are constantly trying to adapt to it. Wouldn't it be great to learn a technology once and continue to use it regardless of the latest trend? Obviously, nobody has a crystal ball to accurately predict the future, but Ansible's track record speaks well for the future environment adaptation.

## The vendor Support

Let's face it, we don't live in a vacuum. We wake up everyday needing to deal with network equipment made by various vendors. We saw the difference between Pexpect and API in previous chapters, and the API integration certainly did not appear out of thin air. Each vendor needs to invest time, money, and engineering resources to make the integration happen. The willingness for the vendor to support a technology matters greatly in our world. Luckily, all the major vendors support Ansible as clearly indicated by the ever increasingly available network modules (http://docs.ansible.com/ansible/list_of_network_modules.html).

Why do vendors support Ansible more than other automation tools? Being agentless certainly helps, having SSH as the only dependency greatly lowers the bar of entry. Engineers who have been on the vendor side know that the feature's request process is usually months long and has many hurdles to jump through. Any time a new feature is added, it means more time spent is on regression testing, compatibility checking, integration reviews, and many more. Lowering the bar of entry is usually the first step in getting vendor support.

The fact that Ansible is based on Python, a language liked by many networking professionals, is another great propeller for vendor support. For vendors such as Juniper and Arista who already made investments in PyEZ and Pyeapi, they can easily leverage the existing module and quickly integrate their features into Ansible. As you will see in the next chapter, we can use our existing Python knowledge to easily write our own modules.

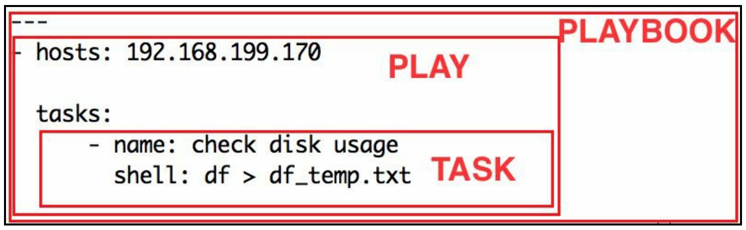
Ansible already has a large number of community-driven modules before its focus on networking. The contribution process is somewhat baked and established or as much an open source project as it can be. The core Ansible team is familiar in working with the community for submission and contribution.

Another reason for the increased network vendor support also has to do with Ansible's ability to give vendors the ability to express their own strength in the module context. We will see in the next section that besides SSH, the Ansible module can also be executed locally and communicated to the devices using API. This ensures that vendors can express their latest and greatest features as soon as they make their API available. In terms of network professionals, this means that you can use the features to select the vendors whenever you are using Ansible as an automation platform.

We have spent a relatively large portion of space discussing vendor support because I feel this is often an overlooked part in the Ansible story. Having vendors willing to put their weight behind the tool means you, the network engineer, can sleep at night knowing that the next big thing in networking will have a high chance of Ansible support, and you are not locked in to your current vendor as your network needs grow.

# The Ansible architecture

The Ansible architecture consist of playbooks, plays, and tasks. Take a look at previously seen `df_playbook.yml`:



```
---
- hosts: 192.168.199.170

  tasks:
    - name: check disk usage
      shell: df > df_temp.txt
```

Ansible Playbook

The whole file is called a playbook, which contains one ore more plays. Each play can consist of one or more tasks. In our simple example, we only have one play, which contains a single task. In this section, we will take a look at the following:

- **YAML**: This format is extensively used in Ansible to express playbooks and variables.
- **Inventory**: Inventory is where you can specify and group hosts in your infrastructure. You can also optionally specify host and group variables in the inventory file.
- **Variables**: Each of the network device is different. It has different hostname, IP, neighbor relations, and such. Variables allow for a standard set of plays while still accommodating these differences.
- **Templates**: Templates are nothing new in networking. In fact, you are probably using one without thinking it is a template. What do we typically do when we need to provision a new device or replace an RMA? We copy the old configuration over and replace the differences such as the hostname and the LoopBack IP addresses. Ansible standardizes the template formatting with Jinja2, which we will dive deeper into.

In the next chapter, we will cover some more advanced topics such as conditionals, loops, blocks, handlers, playbook roles, and how they can be included with network management.

# YAML

YAML is the syntax used for Ansible playbooks and other files. The official YAML documentation contains the full specifications of syntax. Here is a compact version as it pertains to the most common usage for Ansible:

- YAML file starts with three dashes (`---`)
- Whitespace indentation is used to denote structure when they are lined up, just like Python
- Comments begin with hash (`#`) sign
- List members are denoted by a leading hyphen (`-`) with one member per line
- Lists can also be denoted via square brackets (`[]`) with elements separated by comma (`,`)
- Dictionaries are denoted by key:value pair with colon for separation
- Dictionaries can denoted by curly braces with elements separated by comma (`,`)
- Strings can be unquoted, but can also be enclosed in double or single quotes

As you can see, YAML maps well into JSON and Python datatypes. If I were to rewrite the previous `df_playbook.yml` into `df_playbook.json`, this is how it would look like:

```
[
  {
    "hosts": "192.168.199.170",
    "tasks": [
    "name": "check disk usage",
    "shell": "df > df_temp.txt"
   ]
  }
]
```

This is obviously not a valid playbook but an aid in helping to understand the YAML formats in using JSON format as a comparison. Most of the time, comments (`#`), lists (`-`), and dictionaries (key:value) are what you will see in a playbook.

# Inventories

By default, Ansible looks at the `/etc/ansible/hosts` file for hosts specified in your playbook. As mentioned, I find it more clear to specify the host file via the `-i` option as we have been doing up to this point. To expand on our previous example, we can write our inventory host file as follows:

```
    [ubuntu]
    192.168.199.170

    [nexus]
    192.168.199.148
    192.168.199.149

    [nexus:vars]
    username=cisco
    password=cisco

    [nexus_by_name]
    switch1 ansible_host=192.168.199.148
    switch2 ansible_host=192.168.199.149
```

As you may have guessed, the square bracket headings specifies group names, so later on in the playbook, we can point to this group. For example, in `cisco_1.yml` and `cisco_2.yml`, I can act on all hosts specified under the `nexus` group by pointing to the group name of **nexus**:

```
---
- name: Configure SNMP Contact
  hosts: "nexus"
  gather_facts: false
  connection: local
<skip>
```

A host can exist in more than one groups. The group can also be nested as children:

```
[cisco]
router1
router2

[arista]
switch1
switch2

[datacenter:children]
cisco
```

```
    arista
```

In the previous example, the datacenter group includes both the `cisco` and `arista` members.

We will discuss variables in the next section. But you can optionally specify variables belonging to the host and group in the inventory file as well. In our first inventory file example, [`nexus:vars`] specifies variables for the whole nexus group. The `ansible_host` variable declares variable for each of the host on the same line.

For more information on inventory file, check out the official documentation (http://docs.ansible.com/ansible/intro_inventory.html).

# Variables

We discussed variables a bit in the inventory section. Because our managed nodes are not exactly alike, we need to accommodate the differences via variables. Variable names should be letters, numbers, and underscores and always start with a letter. Variables are commonly defined in three locations:

- Playbook
- The inventory file
- Separate files to be included in included files and roles

Let's look at an example of defining variables in a playbook, `cisco_1.yml`:

```
---
- name: Configure SNMP Contact
  hosts: "nexus"
  gather_facts: false
  connection: local

  vars:
    cli:
      host: "{{ inventory_hostname }}"
      username: cisco
      password: cisco
      transport: cli

  tasks:
    - name: configure snmp contact
      nxos_snmp_contact:
        contact: TEST_1
        state: present
        provider: "{{ cli }}"

      register: output

    - name: show output
      debug:
        var: output
```

You can see the `cli` variable declared under the vars section, which is being used in the task of `nxos_snmp_contact`.

*For more information on the `nxso_snmp_contact` module, check out the online documentation (http://docs.ansible.com/ansible/nxos_snmp_contact_module.html).*

To reference a variable, you use the Jinja2 templating system convention of double curly bracket. You don't need to put quotes around the curly bracket unless you are starting a value with it. I typically find it easier to put a quote around the variable value regardless.

You may have also noticed the `{{ inventory_hostname }}` reference, which is not declared in the playbook. It is one of the default variables that Ansible provides for you automatically, and it is sometimes referred to as the magic variable.

*There are not many many magic variables, and you can find the list on the documentation (http://docs.ansible.com/ansible/playbooks_variables.html#magic-variables-and-how-to-access-information-about-other-hosts).*

We have declared variables in an inventory file in the pervious section:

```
    [nexus:vars]
    username=cisco
    password=cisco

    [nexus_by_name]
    switch1 ansible_host=192.168.199.148
    switch2 ansible_host=192.168.199.149
```

To use the variables in the inventory file instead of declaring it in the playbook, let's add the group variables for `[nexus_by_name]` in the host file:

```
[nexus_by_name]
switch1 ansible_host=192.168.199.148
switch2 ansible_host=192.168.199.149

[nexus_by_name:vars]
username=cisco
password=cisco
```

Then, modify the playbook, `cisco_2.yml`, to reference the variables:

```
---
- name: Configure SNMP Contact
  hosts: "nexus_by_name"
  gather_facts: false
  connection: local

  vars:
    cli:
      host: "{{ ansible_host }}"
      username: "{{ username }}"
```

```
        password: "{{ password }}"
        transport: cli

    tasks:
      - name: configure snmp contact
        nxos_snmp_contact:
          contact: TEST_1
          state: present
          provider: "{{ cli }}"

        register: output

      - name: show output
        debug:
          var: output
```

Notice that in this example, we are referring to the `nexus_by_name` group in the inventory file, the `ansible_host` host variable, and the username and password group variable. This is a good way to hide the username and password in a write-protected file and publish the playbook without the fear of exposing your sensitive data.

*To see more examples of variables, check out the Ansible documentation (http://docs.ansible.com/ansible/playbooks_variables.html).*

To access complex variable data provided in nested data structure, you can use two different notations. Noted in the `nxos_snmp_contact` task, we registered the output in a variable and displayed it using the debug module. You will see something like the following:

```
TASK [show output]
*************************************************************
ok: [switch1] => {
 "output": {
  "changed": false,
    "end_state": {
      "contact": "TEST_1"
    },
  "existing": {
     "contact": "TEST_1"
    },
  "proposed": {
    "contact": "TEST_1"
    },
    "updates": []
   }
}
```

In order to access the nested data, we can use the following notation as specified in `cisco_3.yml`:

```
 msg: '{{ output["end_state"]["contact"] }}'
 msg: '{{ output.end_state.contact }}'
```

You will receive just the value indicated:

```
 TASK [show output in output["end_state"]["contact"]]
 ***************************
ok: [switch1] => {
 "msg": "TEST_1"
}
ok: [switch2] => {
 "msg": "TEST_1"
}

TASK [show output in output.end_state.contact]
*********************************
ok: [switch1] => {
 "msg": "TEST_1"
}
ok: [switch2] => {
 "msg": "TEST_1"
}
```

Lastly, we mentioned variables can also be stored in a separate file. To see how we can use variables in a role or included file, we should get a few more examples under our belt, because they are a bit complicated to start with. We will see more examples of roles in the next chapter.

## Templates with Jinja2

In the previous section, we used variables with the Jinja2 syntax of `{{ variable }}`. While you can do a lot of complex things in Jinja2, luckily we only need some of the basic things to get started.

*Jinja2 (http://jinja.pocoo.org/) is a full featured, powerful template engine for Python. It is widely used in Python web frameworks such as Django and Flask.*

For now, it is enough to just keep in mind that Ansible utilize Jinja2 as the template engine. We will revisit the topics of Jinja2 filters, tests, and lookups as the situation calls for them. You can find more information on Ansible Jinja2 template here (http://docs.ansible.com/ansible/playbooks_templating.html).

## Ansible networking modules

Ansible was originally made for managing nodes with full operating systems such as Linux and Windows; then, it extended to network equipment. You may have already noticed the subtle differences in playbooks that we have used so far, such as the lines of `gather_facts: false` and `connection: local`; we will take a look at closer look at the differences.

## Local connections and facts

Ansible modules are Python code executed on remote host by default. Because the fact that most network equipment does not expose Python directly, we are almost always executing the playbook locally. This means that the playbook is interpreted locally first and commands or configurations are pushed out later on as needed.

Recall that the remote host facts were gathered via the setup module, which was added by default. Since we are executing the playbook locally, the setup module will gather the facts on the local host instead of remote host. This is certainly not needed, therefore when the connection is set to local, we can reduce this unnecessary step by setting the fact gathering to false.

## Provider arguments

As we have seen from Chapter 2, *Low Level Network Device Interactions*, and *c hapter 3*, *API and Intent-Driven Networking*, network equipment can be connected via both SSH or API, depending on the platform and software release. All core networking modules implement a `provider` argument, which is a collection of arguments used to define how to connect to the network device. Some modules only support `cli` while some support other values, for example Arista EAPI and Cisco NXAPI. This is where Ansible's "let the vendor shine" philosophy is demonstrated. The module will have documentation on which transport method they support.

Some of the basic arguments supported by the transport are here:

- `host`: This defines the remote host
- `port`: This defines the port to connect to
- `username`: This is the username to be authenticated
- `password`: This  is the password to be authenticated
- `transport`: This is the type of transport for the connection
- `authorize`: This enables privilege escalation for devices that requires it
- `auth_pass`: This defines the privilege escalation password

As you can see, not all arguments need to be specified. For example, for our previous playbooks, our user is always at the admin privilege when log in, therefore we do not need to specify the authorize or the `auth_pass` arguments.

These arguments are just variables, so they follow the same rules for variable precedence. For example, if I change the `cisco_3.yml` to `cisco_4.yml` and observe the following precedence:

```
---
- name: Configure SNMP Contact
  hosts: "nexus_by_name"
  gather_facts: false
  connection: local

  vars:
    cli:
      host: "{{ ansible_host }}"
      username: "{{ username }}"
      password: "{{ password }}"
      transport: cli

  tasks:
    - name: configure snmp contact
      nxos_snmp_contact:
        contact: TEST_1
        state: present
        username: cisco123
        password: cisco123
        provider: "{{ cli }}"

      register: output

    - name: show output in output["end_state"]["contact"]
      debug:
        msg: '{{ output["end_state"]["contact"] }}'

    - name: show output in output.end_state.contact
      debug:
        msg: '{{ output.end_state.contact }}'
```

The username and password defined on the task level will override the username and password at the playbook level. I will receive the following error when trying to connect because the user does not exist on the device:

```
PLAY [Configure SNMP Contact]
**************************************************

TASK [configure snmp contact]
**************************************************
fatal: [switch2]: FAILED! => {"changed": false, "failed": true,
"msg": "failed to connect to 192.168.199.149:22"}
fatal: [switch1]: FAILED! => {"changed": false, "failed": true,
"msg": "failed to connect to 192.168.199.148:22"}
 to retry, use: --limit
@/home/echou/Master_Python_Networking/Chapter4/cisco_4.retry

PLAY RECAP
**********************************************************************
switch1 : ok=0 changed=0 unreachable=0 failed=1
switch2 : ok=0 changed=0 unreachable=0 failed=1
```

# The Ansible Cisco example

Cisco support in Ansible is categorized by the operating systems: IOS, IOSXR, and NXOS. We have already seen a number of NXOS examples, so in this section, let's try to manage IOS-based devices.

Our host file will consist of two hosts, `R1` and `R2`:

```
[ios_devices]
R1 ansible_host=192.168.24.250
R2 ansible_host=192.168.24.251

[ios_devices:vars]
username=cisco
password=cisco
```

Our playbook, `cisco_5.yml`, will use the `ios_command` module to execute arbitrary show commands:

```
    ---
    - name: IOS Show Commands
      hosts: "ios_devices"
      gather_facts: false
      connection: local

      vars:
        cli:
          host: "{{ ansible_host }}"
          username: "{{ username }}"
          password: "{{ password }}"
          transport: cli

      tasks:
- name: ios show commands
          ios_command:
            commands:
              - show version | i IOS
              - show run | i hostname
            provider: "{{ cli }}"

          register: output

        - name: show output in output["end_state"]["contact"]
          debug:
            var: output
```

The result is what we would expected as the show command output:

```
  $ ansible-playbook -i ios_hosts cisco_5.yml

  PLAY [IOS Show Commands]
  *******************************************************

  TASK [ios show commands]
  *******************************************************
  ok: [R1]
  ok: [R2]

  TASK [show output in output["end_state"]["contact"]]
  ***************************
  ok: [R1] => {
   "output": {
   "changed": false,
   "stdout": [
   "Cisco IOS Software, 7200 Software (C7200-A3JK9S-M), Version
  12.4(25g), RELEASE SOFTWARE (fc1)",
   "hostname R1"
   ],
   "stdout_lines": [
   [
   "Cisco IOS Software, 7200 Software (C7200-A3JK9S-M), Version
  12.4(25g), RELEASE SOFTWARE (fc1)"
   ],
   [
   "hostname R1"
   ]
   ]
   }
  }
  ok: [R2] => {
   "output": {
   "changed": false,
   "stdout": [
   "Cisco IOS Software, 7200 Software (C7200-A3JK9S-M), Version
  12.4(25g), RELEASE SOFTWARE (fc1)",
   "hostname R2"
   ],
   "stdout_lines": [
   [
   "Cisco IOS Software, 7200 Software (C7200-A3JK9S-M), Version
  12.4(25g), RELEASE SOFTWARE (fc1)"
   ],
   [
   "hostname R2"
   ]
   ]
   }
  }

  PLAY RECAP
*********************************************************************
  R1 : ok=2 changed=0 unreachable=0 failed=0
  R2 : ok=2 changed=0 unreachable=0 failed=0
```

I wanted to point out a few things illustrated by the example:

- The playbook between NXOS and IOS is largely identical

- The syntax `nxos_snmp_contact` and `ios_command` modules follow the same pattern with the only difference being the argument for the modules
- The IOS version of the devices are pretty old with no understanding of API, but the modules still have the same look and feel

As you can see from the example, once we have the basic syntax down for the playbooks, the subtle difference relies on the module underneath.

# The Ansible Juniper example

The Ansible Juniper module requires the Juniper PyEZ package and NETCONF. If you have been following the API example in Chapter 3, *API and Intent-Driven Networking*, you are good to go. If not, refer back to the section for installation instructions as well as some test script to make sure PyEZ works. The Python package called `jxmlease` is also required:

```
$ sudo pip install jxmlease
```

In the host file, we will specify the device and connection variables:

```
[junos_devices]
J1 ansible_host=192.168.24.252

[junos_devices:vars]
username=juniper
password=juniper!
```

In our module test, we will use the `junos_facts` module to gather basic facts for the device. This module is equivalent to the setup module and will come in handy if we need to take action depending on the returned value. Note the different value of transport and port in the example here:

```
---
- name: Get Juniper Device Facts
  hosts: "junos_devices"
  gather_facts: false
  connection: local

  vars:
    netconf:
      host: "{{ ansible_host }}"
      username: "{{ username }}"
      password: "{{ password }}"
      port: 830
      transport: netconf

  tasks:
    - name: collect default set of facts
      junos_facts:
        provider: "{{ netconf }}"

      register: output

    - name: show output
      debug:
            var: output
```

When executed, you will receive this output from the Juniper device:

```
 PLAY [Get Juniper Device Facts]
 ************************************************

 TASK [collect default set of facts]
 ********************************************
 ok: [J1]

 TASK [show output]
 *************************************************************
ok: [J1] => {
  "output": {
  "ansible_facts": {
  "HOME": "/var/home/juniper",
  "domain": "python",
  "fqdn": "master.python",
  "has_2RE": false,
  "hostname": "master",
  "ifd_style": "CLASSIC",
  "model": "olive",
  "personality": "UNKNOWN",
  "serialnumber": "",
  "switch_style": "NONE",
  "vc_capable": false,
  "version": "12.1R1.9",
  "version_info": {
  "build": 9,
  "major": [
  12,
  1
  ],
  "minor": "1",
  "type": "R"
  }
  },
  "changed": false
  }
 }

 PLAY RECAP
*********************************************************************
 J1 : ok=2 changed=0 unreachable=0 failed=0
```

# The Ansible Arista example

The final module example we will look at will be the Arista commands module. At this point, we are quite familiar with our playbook syntax and structure. The Arista device can be configured to use transport using `cli` or `eapi`, so in this example, we will use `cli`.

This is the host file:

```
[eos_devices]
A1 ansible_host=192.168.199.158
```

The playbook is also similar to what we have seen:

```
---
- name: EOS Show Commands
  hosts: "eos_devices"
  gather_facts: false
  connection: local

  vars:
    cli:
      host: "{{ ansible_host }}"
      username: "arista"
      password: "arista"
      authorize: true
      transport: cli

  tasks:
    - name: eos show commands
      eos_command:
        commands:
          - show version | i Arista
        provider: "{{ cli }}"
      register: output

    - name: show output
      debug:
        var: output
```

The output will show the standard output as we would expect from the command line:

```
PLAY [EOS Show Commands]
*******************************************************

TASK [eos show commands]
*******************************************************
```

```
  ok: [A1]

  TASK [show output]
 ************************************************************
  ok: [A1] => {
   "output": {
   "changed": false,
   "stdout": [
   "Arista DCS-7050QX-32-F"
   ],
   "stdout_lines": [
   [
   "Arista DCS-7050QX-32-F"
   ]
   ],
   "warnings": []
   }
  }

  PLAY RECAP
*******************************************************************
  A1 : ok=2 changed=0 unreachable=0 failed=0
```

## Summary

In this chapter, we took a grand tour of the open source automation framework of Ansible. Unlike Pexpect-based and API-driven network automation scripts, Ansible provides a higher layer abstraction called the playbook to automate your network devices.

Ansible was originally constructed to manage servers and was later extended to network devices; therefore we took a look at a server example. Then, we compared and contrasted the differences when it came to network management playbooks. Later, we looked at the example playbooks for Cisco IOS, Juniper JUNOS, and Arista EOS devices.

In the next chapter, we will leverage the knowledge we gained in this chapter and start to look at some of the more advanced features of Ansible.

# The Python Automation Framework - Ansible Advance Topics

In theÂ previous chapter, we looked at some of the basic structures to get Ansible running. We worked with Ansible inventory files, variables, and playbooks. We also looked at the examples of network modules for Cisco, Juniper, and Arista.Â

In this chapter, we will further build on the knowledge that we gained and dive deeper into the more advanced topics of Ansible. Many books have been written about Ansible. There isÂ more to Ansible than we can cover in two chapters. The goal is to introduce the majority of the features and functions of Ansible that I believe you would need as a network engineer and shorten the learning curve as much as possible.Â

It is important to point out that if you were not clear on some of the points made in the previous chapter, now is a good time to go back and review them as they are a prerequisite for this chapter.

In this chapter, we will look deeper into the following topics:

- Ansible conditionals
- Ansible loops
- Templates
- Group and host variables
- The Ansible vault
- Ansible roles
- Writing your own moduleÂ

We have a lot of ground to cover, so let's get started! Â

## Ansible conditionals

Ansible conditionals are similar to programming conditionals, as they are used to control the flow of your playbooks. In many cases, the result of a play may depend on the value of a fact, variable, or the previous task result. For example, if you have a playbook for upgrading router images, you want to make sure the router image is presented before you move onto the play of rebooting the router.

In this section, we will discuss the when clause that is supported for all modules as well as unique conditional states supported in Ansible networking command modules, such as follows:

- Equal (`eq`)
- Not equal (`neq`)
- Greater than (`gt`)
- Greater than or equal (`ge`)
- Less than (`lt`)
- Less than or equal (`le`)
- Contains

## The when clause

The when clause is useful when you need to check the output from the result and act accordingly. Let's look at a  simple example of its usage in `chapter5_1.yml`:

```
---
- name: IOS Command Output
  hosts: "iosv-devices"
  gather_facts: false
  connection: local
  vars:
    cli:
      host: "{{ ansible_host }}"
      username: "{{ username }}"
      password: "{{ password }}"
      transport: cli
  tasks:
    - name: show hostname
      ios_command:
        commands:
          - show run | i hostname
            provider: "{{ cli }}"
        register: output
    - name: show output
      when: '"iosv-2" in "{{ output.stdout }}"'
      debug:
        msg: '{{ output }}'
```

We have seen all the elements in this playbook before in Chapter 4, *Python Automation Framework – Ansbile Basics;* the only difference is on the second task, as we are using the `when` clause to check if the output contains `iosv-2`. If true, we will proceed to the task, which is using the debug module to display the output. When the playbook is run, we will see the following output:

```
<skip>
TASK [show output]
*************************************************************
skipping: [ios-r1]
ok: [ios-r2] => {
    "msg": {
        "changed": false,
       "stdout": [
            "hostname iosv-2"
        ],
        "stdout_lines": [
            [
                "hostname iosv-2"
            ]
        ],
        "warnings": []
    }
}
<skip>
```

We can see that the `iosv-r1` device is skipped from the output because the clause did not pass. We can further expand this example in `chapter5_2.yml` to only apply configuration changes when the condition is met:

```
<skip>
tasks:
  - name: show hostname
    ios_command:
      commands:
        - show run | i hostname
      provider: "{{ cli }}"
    register: output
  - name: config example
    when: '"iosv-2" in "{{ output.stdout }}"'
    ios_config:
      lines:
        - logging buffered 30000
      provider: "{{ cli }}"
```

We can see the execution output here:

```
TASK [config example]
**********************************************************
skipping: [ios-r1]
changed: [ios-r2]

PLAY RECAP
***********************************************************
ios-r1 : ok=1 changed=0 unreachable=0 failed=0
ios-r2 : ok=2 changed=1 unreachable=0 failed=0
```

Note that the output is showing that only `ios-r2` is changed while `ios-r1` is skipped. The when clause is also very useful in situations such as when the setup module is supported and you want to act on some `facts` that were gathered initially. For example, the following statement will ensure only the Ubuntu host with major release `16` will be acted upon by placing a conditional statement in the clause:

```
when: ansible_os_family == "Debian" and ansible_lsb.major_release|int >= 16
```

*For more conditionals, check out the Ansible conditionals documentation (http://docs.ansible.com/ansible/playbooks_conditionals.html).*

## Network module conditional

Let's take a look at a network device example. We can take advantage of the fact that both IOSv and Arista EOS provide the output in JSON format in the show commands. For example, we can check the status of the interface:

```
arista1#sh interfaces ethernet 1/3 | json
{
 "interfaces": {
 "Ethernet1/3": {
 "interfaceStatistics": {
<skip>
 "outPktsRate": 0.0
 },
 "name": "Ethernet1/3",
 "interfaceStatus": "disabled",
 "autoNegotiate": "off",
 <skip>
}
arista1#
```

If we have an operation that we want to preform and it depends on `Ethernet1/3` being disabled in order to have no user impact, such as when users are actively connected to `Ethernet1/3`. We can use the following tasks configuration in `chapter5_3.yml`, which is provided in the `eos_command` module to make sure that is the case before we proceed:

```
<skip>
 tasks:
   - name: "sh int ethernet 1/3 | json"
     eos_command:
       commands:
         - "show interface ethernet 1/3 | json"
       provider: "{{ cli }}"
       waitfor:
         - "result[0].interfaces.Ethernet1/3.interfaceStatus eq
disabled"
     register: output
   - name: show output
     debug:
       msg: "Interface Disabled, Safe to Proceed"
```

Upon the met condition, the subsequent task will be executed:

```
TASK [sh int ethernet 1/3 | json]
**********************************************
ok: [arista1]
```

```
TASK [show output]
**************************************************************
ok: [arista1] => {
 "msg": "Interface Disabled, Safe to Proceed"
}
```

Otherwise, an error will be given as follows:

```
TASK [sh int ethernet 1/3 | json]
**********************************************
fatal: [arista1]: FAILED! => {"changed": false, "commands": ["show
interface ethernet 1/3 | json | json"], "failed": true, "msg":
"matched error in response: show interface ethernet 1/3 | json |
jsonrn% Invalid input (privileged mode required)rn********1>"}
 to retry, use: --limit
@/home/echou/Master_Python_Networking/Chapter5/chapter5_3.retry

PLAY RECAP
*******************************************************************
arista1 : ok=0 changed=0 unreachable=0 failed=1
```

Check out the other conditions such as contains, greater than, and less than, as they fit in your situation.

## Ansible loops

Ansible provides a number of loops in the playbook, such as standard loops, looping over files, subelements, do-until, and many more. In this section, we will look at two of the most commonly used loop forms, standard loops, and looping over hash values.

## Standard loops

Standard loops in playbook are often used to easily perform similar tasks multiple times. The syntax for standard loops is very easy; the variable `{{ item }}` is the placeholder looping over the `with_items` list. For example, take a look at the following:

```
      tasks:
        - name: echo loop items
          command: echo {{ item }}
          with_items: ['r1', 'r2', 'r3', 'r4', 'r5']
```

It will loop over the five list items with the same `echo` command:

```
TASK [echo loop items] *********************************************************
changed: [192.168.199.185] => (item=r1)
changed: [192.168.199.185] => (item=r2)
changed: [192.168.199.185] => (item=r3)
changed: [192.168.199.185] => (item=r4)
changed: [192.168.199.185] => (item=r5)
```

Combining with network command module, the following task will add multiple vlans to the device:

```
 tasks:
   - name: add vlans
     eos_config:
       lines:
           - vlan {{ item }}
       provider: "{{ cli }}"
     with_items:
         - 100
         - 200
         - 300
```

The `with_items` list can also be read from a variable, which gives greater flexibility to the structure of your playbook:

```
vars:
  vlan_numbers: [100, 200, 300]
<skip>
tasks:
  - name: add vlans
    eos_config:
      lines:
          - vlan {{ item }}
      provider: "{{ cli }}"
    with_items: "{{ vlan_numbers }}"
```

The standard loop is a great time saver when it comes to performing redundant tasks in a playbook. In the next section, we will take a look at looping over dictionaries.

## Looping over dictionaries

Looping over a simple list is nice. However, we often have an entity with more than one attributes associate with it. If you think of the `vlan` example in the last section, each `vlan` would have several unique attributes to it, such as `vlan` description, the IP address, and possibly others. Often times, we can use a dictionary to represent the entity to incorporate multiple attributes to it.

Let's expand on the `vlan` example in the last section for a dictionary example in `chapter5_6.yml`. We defined the dictionary values for three `vlans,` each has a nested dictionary for description and the IP address:

```
<skip>
vars:
   cli:
     host: "{{ ansible_host }}"
     username: "{{ username }}"
     password: "{{ password }}"
     transport: cli
   vlans: {
       "100": {"description": "floor_1", "ip": "192.168.10.1"},
       "200": {"description": "floor_2", "ip": "192.168.20.1"}
       "300": {"description": "floor_3", "ip": "192.168.30.1"}
   }
```

We can configure the first task, `add vlans`, using the key of the each of the item as the `vlan` number:

```
 tasks:
   - name: add vlans
     nxos_config:
       lines:
         - vlan {{ item.key }}
       provider: "{{ cli }}"
     with_dict: "{{ vlans }}"
```

We can proceed with configuring the `vlan` interface. Notice that we use the parents parameter to uniquely identify the section the commands should be checked against. This is due to the fact that the description and the IP address are both configured under the `interface vlan <number>` subsection:

```
- name: configure vlans
  nxos_config:
     lines:
       - description {{ item.value.name }}
       - ip address {{ item.value.ip }}/24
     provider: "{{ cli }}"
     parents: interface vlan {{ item.key }}
  with_dict: "{{ vlans }}"
```

Upon execution, you will see the dictionary being looped through:

```
TASK [configure vlans] *******************************************************
changed: [nxos-r1] => (item={'key': u'300', 'value': {u'ip': u'192.168.30.1',
u'name': u'floor_3'}})
changed: [nxos-r1] => (item={'key': u'200', 'value': {u'ip': u'192.168.20.1',
u'name': u'floor_2'}})
changed: [nxos-r1] => (item={'key': u'100', 'value': {u'ip': u'192.168.10.1',
u'name': u'floor_1'}})
```

Let us check if the intended configuration is applied to the device:

```
nx-osv-1# sh run | i vlan
<skip>
vlan 1,10,100,200,300
nx-osv-1#
```

```
nx-osv-1# sh run | section "interface Vlan100"
interface Vlan100
  description floor_1
  ip address 192.168.10.1/24
nx-osv-1#
```

*For more loop types of Ansible, feel free to check out the documentation (http://docs.ansible.com/ansible/playbooks_loops.html).*

Looping over dictionaries takes some practice the first few times when you use them. But just like standard loops, looping over dictionaries will be an invaluable tool in your tool belt.

# Templates

For as long as I can remember, I have always used a kind of a network template. In my experience, many of the network devices have sections of network configuration that are identical, especially if these devices serve the same role in the network. Most of the time when we need to provision a new device, we use the same configuration in the form of a template, replace the necessary fields, and copy the file over to the new device. With Ansible, you can automate all the work using the template module (http://docs.ansible.com/ansible/template_module.html).

The base template file we are using utilizes the Jinja2 template language (http://jinja.pocoo.org/docs/). We briefly discussed Jinja2 templating language in the previous chapter, and we will look at it a bit more here. Just like Ansible, Jinja2 has its own syntax and method of doing loops and conditionals; fortunately, we just need to know the very basics of it for our purpose. You will learn the syntax by gradually building up your playbook.

The basic syntax for template usage is very simple; you just need to specify the source file and the destination location that you want to copy it to. We will create an empty file now:

```
$ touch file1
```

Then, we will use the following playbook to copy `file1` to `file2`, note the playbook is executed on the control machine only; now, specify the path of both the source and destination files as arguments:

```
---
- name: Template Basic
  hosts: localhost

  tasks:
    - name: copy one file to another
      template:
        src=./file1
        dest=./file2
```

We do not need to specify a host file since localhost is available by default; you will, however, get a warning:

```
$ ansible-playbook chapter5_7.yml
 [WARNING]: provided hosts list is empty, only localhost is available
<skip>
TASK [copy one file to another] ************************************************

changed: [localhost]
<skip>
```

The source file can have any extension, but since they are processed through the Jinja2 template engine, let's create a file called `nxos.j2` as the template source. The template will follow the Jinja2 convention of using double curly brackets to specify the variable:

```
    hostname {{ item.value.hostname }}
    feature telnet
    feature ospf
    feature bgp
    feature interface-vlan

    username {{ item.value.username }} password {{ item.value.password
    }} role network-operator
```

## The Jinja2 template

Let's also modify the playbook accordingly. In `chapter5_8.yml`, we will make the following changes:

1. Change the source file to the `nxos.j2`.
2. Change the destination file to be a variable.
3. Provide the variable values that we will substitute in the template that is indicated under the tasks section.

```
---
- name: Template Looping
  hosts: localhost

  vars:
    nexus_devices: {
      "nx-osv-1": {"hostname": "nx-osv-1", "username": "cisco",
"password": "cisco"}
    }

  tasks:
    - name: create router configuration files
      template:
        src=./nxos.j2
        dest=./{{ item.key }}.conf
      with_dict: "{{ nexus_devices }}"
```

After running the playbook, you will find the destination file called `nx-osv-1.conf` with the values filled in and ready to be used:

```
$ cat nx-osv-1.conf
hostname nx-osv-1

feature telnet
feature ospf
feature bgp
feature interface-vlan

username cisco password cisco role network-operator
```

# Jinja2 loops

We can also loop through a list as well as a dictionary as we have completed in the last section; make the following changes in `nxos.j2`:

```
{% for vlan_num in item.value.vlans %}
vlan {{ vlan_num }}
{% endfor %}

{% for vlan_interface in item.value.vlan_interfaces %}
interface {{ vlan_interface.int_num }}
  ip address {{ vlan_interface.ip }}/24
{% endfor %}
```

Provide the additional list and dictionary variables in the playbook:

```
vars:
 nexus_devices: {
   "nx-osv-1": {
   "hostname": "nx-osv-1",
   "username": "cisco",
   "password": "cisco",
   "vlans": [100, 200, 300],
   "vlan_interfaces": [
      {"int_num": "100", "ip": "192.168.10.1"},
      {"int_num": "200", "ip": "192.168.20.1"},
      {"int_num": "300", "ip": "192.168.30.1"}
    ]
   }
 }
```

Run the playbook, and you will see the configuration for both `vlan` and `vlan_interfaces` filled in on the router config.

## The Jinja2 conditional

Jinja2 also supports and an if conditional checking. Let's add this field in for turning on the netflow feature for certain devices. We will add the following to the `nxos.j2` template:

```
{% if item.value.netflow_enable %}
feature netflow
{% endif %}
```

The playbook is here:

```
vars:
  nexus_devices: {
  <skip>
         "netflow_enable": True
  <skip>
 }
```

The last step we will undertake is to make the `nxos.j2` more scalable by placing the `vlan` interface section inside of a `true-false` conditional check. In the real world, most often, we will have multiple devices with knowledge of the `vlan` information but only one device as the gateway for client hosts:

```
{% if item.value.l3_vlan_interfaces %}
{% for vlan_interface in item.value.vlan_interfaces %}
interface {{ vlan_interface.int_num }}
 ip address {{ vlan_interface.ip }}/24
{% endfor %}
{% endif %}
```

We will also add a second device called `nx-osv-2` in the playbook:

```
 vars:
   nexus_devices: {
   <skip>
     "nx-osv-2": {
       "hostname": "nx-osv-2",
       "username": "cisco",
       "password": "cisco",
       "vlans": [100, 200, 300],
       "l3_vlan_interfaces": False,
       "netflow_enable": False
     }
    <skip>
 }
```

Neat, huh? This can certainly save us a ton of time for something that required repeated copy and paste before.

## Group and host variables

Notice in the previous playbook, we have repeated ourselves in the username and password variables for the two devices under the `nexus_devices` variable:

```
vars:
  nexus_devices: {
    "nx-osv-1": {
      "hostname": "nx-osv-1",
      "username": "cisco",
      "password": "cisco",
      "vlans": [100, 200, 300],
    <skip>
    "nx-osv-2": {
      "hostname": "nx-osv-2",
      "username": "cisco",
      "password": "cisco",
      "vlans": [100, 200, 300],
    <skip>
```

This is not ideal. If we ever need to update the username and password values, we will need to remember to update at two locations. This increases the management burden as well as the chances of making mistakes if we ever forget to update all the locations. For best practice, Ansible suggests that we use `group_vars` and `host_vars` directories to separate out the variables.

TIP *For more Ansible best practices, check out http://docs.ansible.com/ansible/playbooks_best_practices.html.*

## Group variables

By default, Ansible will look for group variables in the same directory as the playbook called `group_vars` for variables that can be applied to the group. By default, it will look for the file name that matches the group name. For example, if we have a group called `[nexus-devices]` in the inventory file, we can have a file under `group_vars` named nexus-devices to house all the variables applied to the group. We can also use a file called all to include variables applied to all the groups.

We will utilize this feature for our username and password variables:

```
$ mkdir group_vars
```

Then, we can create a YAML file called `all` to include the username and password:

```
$ cat group_vars/all
---
username: cisco
password: cisco
```

We can then use variables for the playbook:

```
    vars:
      nexus_devices: {
       "nx-osv-1": {
          "hostname": "nx-osv-1",
          "username": "{{ username }}",
          "password": "{{ password }}",
          "vlans": [100, 200, 300],
        <skip>
         "nx-osv-2": {
          "hostname": "nx-osv-2",
          "username": "{{ username }}",
          "password": "{{ password }}",
          "vlans": [100, 200, 300],
        <skip>
```

## Host variables

We can further separate out the host variables in the same format as the group variables:

```
$ mkdir host_vars
```

In our case, we execute the commands on the localhost, therefore the file under `host_vars` should be named accordingly, such as `host_vars/localhost`. Note that we kept the variables declared in `group_vars`.

```
---
"nexus_devices":
  "nx-osv-1":
    "hostname": "nx-osv-1"
    "username": "{{ username }}"
    "password": "{{ password }}"
    <skip>
    "netflow_enable": True
  "nx-osv-2":
    "hostname": "nx-osv-2"
    "username": "{{ username }}"
    "password": "{{ password }}"
    <skip>
```

After we separate out the variables, the playbook now becomes very lightweight and only consists of the logic of our operation:

```
$ cat chapter5_9.yml
---
- name: Ansible Group and Host Varibles
  hosts: localhost

  tasks:
    - name: create router configuration files
      template:
        src=./nxos.j2
        dest=./{{ item.key }}.conf
      with_dict: "{{ nexus_devices }}"
```

The `group_vars` and `host_vars` directories not only decrease our operations overhead. They can help in securing the files, which we will see next.

## The Ansible vault

As you can see from the previous section, many a times, the Ansible variable often provides sensitive information such as username and password. It would be a good idea to put some security measures around the variables so that we can safeguard against these information. The Ansible vault provides encryption for files rather than using plain text.

All Ansible Vault functions start with the `ansible-vault` command. You can manually create a encrypted file via the create option. You will be asked to enter a password. If you try to view the file, you will find that the file is not in clear text:

```
$ ansible-vault create secret.yml
Vault password:

$ cat secret.yml
$ANSIBLE_VAULT;1.1;AES256
336564626462373962326635326361323639323635353630646665656430353261383737623<skip>65
3962
```

You can later on edit the file via the `edit` option or view the file via the `view` option:

```
$ ansible-vault edit secret.yml
Vault password:

$ ansible-vault view secret.yml
Vault password:
```

Let's encrypt the `group_vars/all and host_vars/localhost` variable files:

```
$ ansible-vault encrypt group_vars/all host_vars/localhost
Vault password:
Encryption successful
```

Now, when we run the playbook, we will get a decryption failed error message:

```
ERROR! Decryption failed on
/home/echou/Master_Python_Networking/Chapter5/Vaults/group_vars/all
```

We will need to use the `--ask-vault-pass` option when we run the playbook:

```
$ ansible-playbook chapter5_10.yml --ask-vault-pass
Vault password:
```

The decrypt will happen in memory for any vault encrypted files that are accessed.

*Currently, the vault requires all the files to be encrypted with the same password.*

We can also save the password in a file and make sure that the specific file has restricted permission:

```
$ chmod 400 ~/.vault_password.txt
$ ls -lia ~/.vault_password.txt
809496 -r-------- 1 echou echou 9 Feb 18 12:17 /home/echou/.vault_password.txt
```

We can then execute the playbook with the `--vault-password-file` option:

```
$ ansible-playbook chapter5_10.yml --vault-password-file ~/.vault_password.txt
```

## The Ansible include and roles

The best way to handle complex tasks is to break it down into smaller pieces. This approach is common in both Python and network engineering, of course. In Python, we break complicated code into functions, classes, modules, and packages. In Networking, we also break large networks into sections such as racks, rows, clusters, and datacenters.  In Ansible, uses `roles` and `includes` to segment and organize a large playbook into multiple files. Breaking up a large Ansible playbook simplifies the structure as each of file focuses on fewer tasks. It also allows the sections of the playbook easier to be reused.

# The Ansible include statement

As the playbook grows in size, it will eventually become obvious that many of the tasks and playbooks can be shared across different playbooks. The Ansible `include` statement is similar to many Linux configuration files that just tell the machine to extend the file the same way as if the file was directly written in. We can do an include statement for both playbooks and tasks. We will look at a simple example for extending our tasks.

Let's assume that we want to show outputs for two different playbooks. We can make a separate YAML file called `show_output.yml` as an additional task:

```
---
- name: show output
   debug:
     var: output
```

Then, we can reuse this task in multiple playbooks, such as in `chapter5_11_1.yml` that looked largely identical to the last playbook with the exception of registering the output and the include statement at the end:

```
---
- name: Ansible Group and Host Varibles
  hosts: localhost

  tasks:
    - name: create router configuration files
      template:
        src=./nxos.j2
        dest=./{{ item.key }}.conf
      with_dict: "{{ nexus_devices }}"
      register: output

    - include: show_output.yml
```

Another playbook, `chapter5_11_2.yml`, can reuse `show_output.yml` inn the same way:

```
---
- name: show users
  hosts: localhost

  tasks:
    - name: show local users
```

```
          command: who
          register: output

        - include: show_output.yml
```

Note that both playbooks use the same variable name, output, as the `show_output.yml` hard codes the variable name for simplicity in demonstration. You can also pass variables into the included file.

## Ansible roles

Ansible roles separate the logical function with physical host to fit your network better. For example, you can construct roles such as spines, leafs, core, as well as Cisco, Juniper, and Arista. The same physical host can belong to multiple roles; for example, a device can belong to both Juniper and the core. This makes logical sense, as we can perform operation such as upgrade all Juniper devices across the network without their location in the layer of the network.

Ansible roles can automatically load certain variables, tasks, and handlers based on a known file infrastructure. The key is that this is a known file structure that we automatically include; in fact, you can think of roles as premade and include statements by Ansible.

The Ansible playbook role documentation (http://docs.ansible.com/ansible/playbooks_roles.html#roles) describes a list of role directories that you can configure. You do not need to use all of them; in fact, we will only modify the tasks and vars folder. However, it is good to know that they exist.

The following is what we will use as an example for our roles:

```
├── chapter5_12.yml
├── chapter5_13.yml
├── hosts
└── roles
  ├── cisco_nexus
  │    ├── defaults
  │    ├── files
  │    ├── handlers
  │    ├── meta
  │    ├── tasks
  │    │   └── main.yml
  │    ├── templates
  │    └── vars
  │        └── main.yml
  └── spines
  ├── defaults
  ├── files
  ├── handlers
  ├── tasks
  │   └── main.yml
  ├── templates
```

```
│  └── vars
│      └── main.yml
```

You can see that at the top level, we have the hosts file as well as the playbooks. We also have a folder named `roles`; inside it, we have two roles defined: `cisco_nexus` and spines. Most of the subfolders under the roles were empty, with the exception of tasks and vars folder. There is a file named `main.yml` inside each of them. This is the default behavior, the `main.yml` file is your entry point that is automatically included in the playbook when you specify the role in the playbook. If you need to break out additional files, you can use the include statement in the `main.yml` file.

Here is our scenario:

- We have two Cisco Nexus devices, `nxos-r1` and `nxos-r2`. We will configure the logging server as well log link-status for all of them utilizing the `cisco_nexus` role for them.
- In addition, nxos-r1 is also a spine device, where we will want to configure more verbose logging, because they are of more critical positioning within our network.

For our `cisco_nexus` role, we have the following variables in `roles/cisco_nexus/vars/main.yml`:

```
---
cli:
  host: "{{ ansible_host }}"
  username: cisco
  password: cisco
  transport: cli
```

We have the following configuration tasks in roles`/cisco_nexus/tasks/main.yml`:

```
---
- name: configure logging parameters
  nxos_config:
    lines:
      - logging server 191.168.1.100
      - logging event link-status default
    provider: "{{ cli }}"
```

Our playbook is extremely simple, as it just needs to specify the hosts that we would like to be configured according to `cisco_nexus role`:

```
---
- name: playbook for cisco_nexus role
  hosts: "cisco_nexus"
  gather_facts: false
  connection: local

  roles:
    - cisco_nexus
```

When you run the playbook, the playbook will include the tasks and varirables defined in the `cisco_nexus` role and configure the devices accordingly.

For our `spine` role, we will have an additional task of more verbose logging in `roles/spines/tasks/mail.yml`:

```
---
- name: change logging level
  nxos_config:
    lines:
      - logging level local7 7
    provider: "{{ cli }}"
```

In our playbook, we can specify that it contains both the role of `cisco_nexus` as well as spiens:

```
---
- name: playbook for spine role
  hosts: "spines"
  gather_facts: false
  connection: local

  roles:
    - cisco_nexus
    - spines
```

Note that, when we do this, the `cisco_nexus` role tasks will be executed followed by the spines role:

```
TASK [cisco_nexus : configure logging parameters] ******************************
changed: [nxos-r1]

TASK [spines : change logging level] *******************************************
ok: [nxos-r1]
```

Ansible roles are flexible and scalable. Just like Python functions and classes. Once your code grows beyond a certain level, it is almost always a good idea to break them into smaller pieces for maintainability.

*You can find more examples of roles in the Ansible examples Git repository at https://github.com/ansible/ansible-examples.*

## Writing your own custom module

By now, you may get the feeling that network management is largely dependent on finding the right module for your device. There is certainly a lot of truth in that logic. Modules provide a way to abstract the interaction between the managed host and the control machine, while it allows you to focus on the logic of your operation. Up to this point, we have seen the major vendors providing a wide range of module support for Cisco, Juniper, and Arista.

Use Cisco Nexus modules as an example, besides specific tasks such as managing the BGP neighbor (`nxos_bgp`) and the aaa server (`nxos_aaa_server`). Most vendors also provide ways to run arbitrary show (`nxos_config`) and configuration commands (`nxos_config`). This generally covers most of our use cases.

What if the device you are using does not currently have any the modules that you are looking for? In this section, we will look at several ways that you can remedy this situation by writing your own custom module.

## The first custom module

Writing a custom module does not need to be complicated; in fact, it doesn't even need to be in Python. But since we are already familiar with Python, we will use Python for our custom modules. We are assuming that the module is what we will be using ourselves and our team without submitting back to Ansible, therefore ignoring some of the documentation and formatting for the time being.

By default, if you make a library folder in the same directory of the playbook, Ansible will include the directory in the module search path. All the module needs is to return a JSON output back to the playbook.

Recall that in Chapter 3, *API and Intent-Driven Networking*, we used the following NXAPI Python script to communicate to the NX-OS device:

```
import requests
import json

url='http://172.16.1.142/ins'
switchuser='cisco'
switchpassword='cisco'

myheaders={'content-type':'application/json-rpc'}
payload=[
 {
   "jsonrpc": "2.0",
   "method": "cli",
   "params": {
     "cmd": "show version",
     "version": 1.2
   },
   "id": 1
 }
]
response = requests.post(url,data=json.dumps(payload),
headers=myheaders,auth=(switchuser,switchpassword)).json()

print(response['result']['body']['sys_ver_str'])
```

When we executed it, we simply got the system version. If we simply modify the last line to be a JSON output, we will see the following:

```
version = response['result']['body']['sys_ver_str']
print json.dumps({"version": version})
```

We can then use the action plugin (https://docs.ansible.com/ansible/dev_guide/developing_plugins.html) in our playbook, `chapter5_14.yml`, to call this custom module:

```
---
- name: Your First Custom Module
  hosts: localhost
  gather_facts: false
  connection: local

  tasks:
    - name: Show Version
      action: custom_module_1
      register: output

    - debug:
        var: output
```

Note that just like the `ssh` connection, we are executing the module locally with the module making API calls outbound. When you execute this playbook, you will get the following output:

```
$ ansible-playbook chapter5_14.yml
 [WARNING]: provided hosts list is empty, only localhost is available

PLAY [Your First Custom Module] ************************************************

TASK [Show Version] ************************************************************
ok: [localhost]

TASK [debug] *******************************************************************
ok: [localhost] => {
 "output": {
 "changed": false,
 "version": "7.3(0)D1(1)"
 }
}

PLAY RECAP *********************************************************************
localhost : ok=2 changed=0 unreachable=0 failed=0
```

As you can see, you can write any module that is supported by API, and Ansible will happily take any returned JSON output.

# The second custom module

Building upon the last module, let's utilize the common module Boilerplate from Ansible as in the module development documentation at http://docs.ansible.com/ansible/dev_guide/developing_modules_general.html. We will modify the last custom module in `custom_module_2.py` to utilize ingesting inputs from the playbook.

First, we will import the boilerplate code from `ansible.module_utils.basic`:

```
from ansible.module_utils.basic import AnsibleModule

if __name__ == '__main__':
    main()
```

From there, we can then define the main function where we will house our code. `AnsibleModule` provides lots of common code for handling returns and parsing arguments. In the following example, we will parse three arguments for `host`, `username`, and `password`, and make them as a required field:

```
def main():
    module = AnsibleModule(
      argument_spec = dict(
      host = dict(required=True),
      username = dict(required=True),
  password = dict(required=True)
  )
)
```

The values can then be retrieved and used in our code:

```
 device = module.params.get('host')
 username = module.params.get('username')
 password = module.params.get('password')

 url='http://' + host + '/ins'
 switchuser=username
 switchpassword=password
```

Finally, we will follow the exit code and return the value:

```
module.exit_json(changed=False, msg=str(data))
```

Our new playbook will look identical to the last one with the exception now that we can pass values for different devices in the playbook:

```
     tasks:
       - name: Show Version
         action: custom_module_1 host="172.16.1.142" username="cisco"
    password="cisco"
         register: output
```

When executed, this playbook will produce the exact same output as the last playbook; however, notice that this playbook and module can be passed around for other people to use without them knowing the details of our module.

Of course, this is a functional but incomplete module; for one thing, we did not perform any error checking or documentation. However, it demonstrates how easy it is to build a custom module from scripts that we have already made.

## Summary

In this chapter, we covered a lot of ground. Building from our previous basic knowledge of Ansible, we expanded into more advanced topics such as conditionals, loops, and templates. We looked at how to make our playbook more scalable with host variables, group variables, include statements, and roles. We also looked at how to secure our playbook with the Ansible vault. Finally, we used Python to make our own custom modules.

Ansible is a very flexible Python framework that can be used for network automation. It provides another abstraction layer separated from the likes of the Pexpect and API-based scripts. Depending on your needs and network environment, it might be the ideal framework that you can use to save time and energy.

In the next chapter, we will look at network security with Python.

# Network Security with Python

In my opinion, network security is a tricky topic to write about. The reason is not a technical one, but rather one with setting up the right scope. The boundaries of network security are so wide that they touchÂ all seven layers of the OSI model. From layer 1 of wire tapping, to layer 4 of transport protocol vulnerability, to layer 7 of man-in-the-middle spoofing, network security is everywhere. The issue is exacerbated by all the newly discovered vulnerabilities, which are sometimes at a daily rate. This does not even include the human social engineering aspect of network security. Â

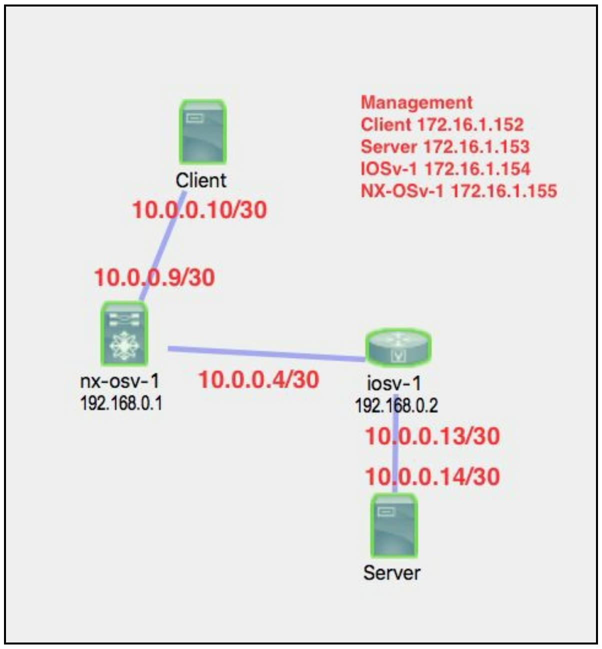
As such, in this chapter, I would like to set the scope for what we will discuss. As we have been doing up to this point, we will be primarily focused on using Python for network device security at OSI layers 3 and 4. We will look at Python tools that we can use to manage individual components as well as using Python as a glue to connect different components, so we can treat network security in a holistic view. In this chapter, we will take a look at the following:

- The lab setup
- Python Scapy for security testingÂ
- Access lists
- Forensic analysis with syslog and UFW using PythonÂ
- Other tools such as Mac address filter list, private VLAN, and Python IP table bindingÂ

## The lab setup

The devices being used in this chapter are a bit different from the previous chapters. In the previous chapters, we were isolating a particular device by focusing on the topic at hand. For this chapter, we will use a few more devices in our lab to illustrate the tools that we will be using.

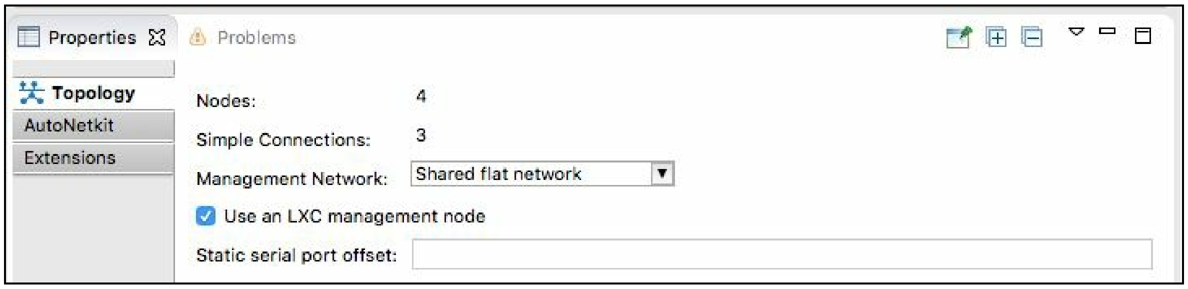
We will be using the same Cisco VIRL tool with four nodes, two server hosts, and two network devices. If you need a refresher on Cisco VIRL, feel free to go back to Chapter 2, *Low-Level Network Device Interactions* where we first introduced the tool:



Lab Topology

*Note that the IP addresses listed will be different in your own lab. They are listed here for an easy reference for the rest of the chapter.*

As illustrated, we will rename the host on the top as the client and the bottom host as the server. This is analogous to an Internet client trying to access a corporate server within our network. We will again use the Shared flat network option for the management network to access the devices for the out-of-band management:



Nodes: 4
Simple Connections: 3
Management Network: Shared flat network
✓ Use an LXC management node
Static serial port offset:

For the two switches, I am choosing to use **Open Shortest Path First** (OSPF) as IGP and putting both the devices in area 0. By default, `BGP` is turned on and both the devices are using AS 1. From configuration auto-generation, the interfaces connected to the Ubuntu hosts are put into OSPF area 1, so they will show up as inter-area routes. The NX-OSv configurations is shown here and the IOSv configuration and output are similar:

```
    interface Ethernet2/1
     description to iosv-1
      no switchport
      mac-address fa16.3e00.0001
      ip address 10.0.0.6/30
      ip router ospf 1 area 0.0.0.0
      no shutdown

    interface Ethernet2/2
      description to Client
      no switchport
      mac-address fa16.3e00.0002
      ip address 10.0.0.9/30
      ip router ospf 1 area 0.0.0.0
      no shutdown

    nx-osv-1# sh ip route
    <skip>
    10.0.0.12/30, ubest/mbest: 1/0
     *via 10.0.0.5, Eth2/1, [110/41], 04:53:02, ospf-1, intra
    192.168.0.2/32, ubest/mbest: 1/0
     *via 10.0.0.5, Eth2/1, [110/41], 04:53:02, ospf-1, intra
    <skip>
```

The OSPF neighbor and the BGP output for NX-OSv is shown here; the IOSv output is similar:

```
nx-osv-1# sh ip ospf neighbors
 OSPF Process ID 1 VRF default
 Total number of neighbors: 1
 Neighbor ID Pri State Up Time Address Interface
 192.168.0.2 1 FULL/DR 04:53:00 10.0.0.5 Eth2/1

nx-osv-1# sh ip bgp summary
BGP summary information for VRF default, address family IPv4 Unicast
BGP router identifier 192.168.0.1, local AS number 1
BGP table version is 5, IPv4 Unicast config peers 1, capable peers 1
2 network entries and 2 paths using 288 bytes of memory
BGP attribute entries [2/288], BGP AS path entries [0/0]
BGP community entries [0/0], BGP clusterlist entries [0/0]

Neighbor V AS MsgRcvd MsgSent TblVer InQ OutQ Up/Down State/PfxRcd
192.168.0.2 4 1 321 297 5 0 0 04:52:56 1
```

The hosts in our network are running Ubuntu 14.04, similar to the Ubuntu VM 16.04 that we have been using up to this point:

```
cisco@Server:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description: Ubuntu 14.04.2 LTS
Release: 14.04
Codename: trusty
```

On both of the Ubuntu hosts, there are two network interfaces, Eth0 connects to the management network while Eth1 connects to the network devices. The routes to the device loopback are directly connected to the network block, and the remote host network is statically routed to Eth1 with the default route going toward the management network:

```
cisco@Client:~$ route -n
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
0.0.0.0 172.16.1.2 0.0.0.0 UG 0 0 0 eth0
10.0.0.4 10.0.0.9 255.255.255.252 UG 0 0 0 eth1
10.0.0.8 0.0.0.0 255.255.255.252 U 0 0 0 eth1
10.0.0.8 10.0.0.9 255.255.255.248 UG 0 0 0 eth1
172.16.1.0 0.0.0.0 255.255.255.0 U 0 0 0 eth0
192.168.0.1 10.0.0.9 255.255.255.255 UGH 0 0 0 eth1
192.168.0.2 10.0.0.9 255.255.255.255 UGH 0 0 0 eth1
```

Just to make sure, let's ping and trace the route to make sure that traffic between our hosts is going through the network devices instead of the default route:

```
## Our server IP is 10.0.0.14
cisco@Server:~$ ifconfig
<skip>
eth1 Link encap:Ethernet HWaddr fa:16:3e:d6:83:02
 inet addr:10.0.0.14 Bcast:10.0.0.15 Mask:255.255.255.252
```

```
## From the client toward server
cisco@Client:~$ ping -c 1 10.0.0.14
PING 10.0.0.14 (10.0.0.14) 56(84) bytes of data.
64 bytes from 10.0.0.14: icmp_seq=1 ttl=62 time=6.22 ms

--- 10.0.0.14 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 6.223/6.223/6.223/0.000 ms

cisco@Client:~$ traceroute 10.0.0.14
traceroute to 10.0.0.14 (10.0.0.14), 30 hops max, 60 byte packets
 1 10.0.0.9 (10.0.0.9) 11.335 ms 11.745 ms 12.113 ms
 2 10.0.0.5 (10.0.0.5) 24.221 ms 41.635 ms 41.638 ms
 3 10.0.0.14 (10.0.0.14) 37.916 ms 38.275 ms 38.588 ms
cisco@Client:~$
```

Great! We have our lab, and we are ready to look at some security tools and measures using Python.

# Python Scapy

Scapy (http://www.secdev.org/projects/scapy/) is a powerful Python-based interactive packet manipulation program. Outside of some commercial program, very few tools can do what Scapy can do, that I know of. The main difference is Scapy allows you to craft your own packet from the very basic level. It is able to forge or decode network packets. Let's take a look at the tool.

## Installing Scapy

At the time of this writing, Scapy 2.3.1 supported Python 2.7. While attempts and forks has been done for the Python 3 support, it is still a prototype (in Phil's own words), so we will use Python 2.7 for our usage. The idea is that Scapy 3 will be Python 3 only and will not be backward compatible to Scapy 2.x.

*If you are interested in testing out the Scapy 3 prototype, here is the latest version: https://bitbucket.org/secdev/scapy3-prototype2. If you want to learn more about the issue and the fork, here is the issue Bitbucket link: https://bitbucket.org/secdev/scapy/issues/5082/compatibility-with-python-3.*

In our lab, since we are crafting packets going from the client to the server, Scapy needs to be installed on the client:

```
cisco@Client:~$ sudo apt-get update
cisco@Client:~$ sudo apt-get install git
cisco@Client:~$ git clone https://github.com/secdev/scapy
cisco@Client:~$ cd scapy/
cisco@Client:~/scapy$ sudo python setup.py install
```

Here is a quick test to make sure the packages installed correctly:

```
cisco@Client:~/scapy$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from scapy.all import *
```

# Interactive examples

In our first example, we will craft an **Internet Control Message Protocol** (**ICMP**) packet on the client and send it to the server. On the server side, we will use `tcpdump` with host filter to see the packet coming in:

```
## Client Side
cisco@Client:~/scapy$ sudo scapy
<skip>
Welcome to Scapy (2.3.3.dev274)
>>> send(IP(dst="10.0.0.14")/ICMP())
.
Sent 1 packets.
>>>

## Server Side
cisco@Server:~$ sudo tcpdump -i eth1 host 10.0.0.10
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 65535 bytes
02:45:16.400162 IP 10.0.0.10 > 10.0.0.14: ICMP echo request, id 0, seq 0, length 8
02:45:16.400192 IP 10.0.0.14 > 10.0.0.10: ICMP echo reply, id 0, seq 0, length 8
```

As you can see, it is very simple to craft a packet. Scapy allows you to build the packet layer by layer using the slash (/) as the separator. The `send` function operates at the layer 3 level, which takes care of routing and layer 2 for you. There is also a `sendp()` alternative that operates at layer 2, which means you will need to specify the interface and link layer protocol.

Let's look at capturing the returned packet by using the send-request (`sr`) function. We are using a special variation, called sr1, of the function that only returns one packet that answers the packet sent:

```
>>> p = sr1(IP(dst="10.0.0.14")/ICMP())
>>> p
<IP version=4L ihl=5L tos=0x0 len=28 id=26713 flags= frag=0L ttl=62 proto=icmp
chksum=0x71 src=10.0.0.14 dst=10.0.0.10 options=[] |<ICMP type=echo-reply code=0
chksum=0xffff id=0x0 seq=0x0 |>>
```

One thing to note is that the `sr()` function itself returns a tuple containing answered and unanswered lists:

```
>>> p = sr(IP(dst="10.0.0.14")/ICMP())
>>> type(p)
<type 'tuple'>
```

```
## unpacking
>>> ans,unans = sr(IP(dst="10.0.0.14")/ICMP())
>>> type(ans)
<class 'scapy.plist.SndRcvList'>
>>> type(unans)
<class 'scapy.plist.PacketList'>
```

If we were to only take a look at the answered packet list, we can see it is another tuple containing the packet that we have sent as well as the returned packet:

```
>>> for i in ans:
...     print(type(i))
...
<type 'tuple'>
>>> for i in ans:
...     print i
...
(<IP frag=0 proto=icmp dst=10.0.0.14 |<ICMP |>>, <IP version=4L ihl=5L tos=0x0
len=28 id=27062 flags= frag=0L ttl=62 proto=icmp chksum=0xff13 src=10.0.0.14
dst=10.0.0.10 options=[] |<ICMP type=echo-reply code=0 chksum=0xffff id=0x0 seq=0x0
|>>)
```

Scapy also provides a layer 7 construct as well, such as a DNS query. In the following example, we are querying an open DNS server for the resolution of `www.google.com`:

```
>>> p = sr1(IP(dst="8.8.8.8")/UDP()/DNS(rd=1,qd=DNSQR(qname="www.google.com")))
>>> p
<IP version=4L ihl=5L tos=0x0 len=76 id=21743 flags= frag=0L ttl=128 proto=udp
chksum=0x27fa src=8.8.8.8 dst=172.16.1.152 options=[] |<UDP sport=domain
dport=domain len=56 chksum=0xc077 |<DNS id=0 qr=1L opcode=QUERY aa=0L tc=0L rd=1L
ra=1L z=0L ad=0L cd=0L rcode=ok qdcount=1 ancount=1 nscount=0 arcount=0 qd=<DNSQR
qname='www.google.com.' qtype=A qclass=IN |> an=<DNSRR rrname='www.google.com.'
type=A rclass=IN ttl=299 rdata='172.217.3.164' |> ns=None ar=None |>>>
>>>
```

# Sniffing

Scapy can also be used to easily capture packets:

```
>>> a = sniff(filter="icmp and host 172.217.3.164", count=5)
>>> a.show()
0000 Ether / IP / TCP 192.168.225.146:ssh > 192.168.225.1:50862 PA / Raw
0001 Ether / IP / ICMP 192.168.225.146 > 172.217.3.164 echo-request 0 / Raw
0002 Ether / IP / ICMP 172.217.3.164 > 192.168.225.146 echo-reply 0 / Raw
0003 Ether / IP / ICMP 192.168.225.146 > 172.217.3.164 echo-request 0 / Raw
0004 Ether / IP / ICMP 172.217.3.164 > 192.168.225.146 echo-reply 0 / Raw
>>>
```

We can look at the packets in some more detail, including the raw format:

```
>>> for i in a:
...     print i.show()
...
<skip>
###[ Ethernet ]###
 dst= <>
 src= <>
 type= 0x800
###[ IP ]###
 version= 4L
 ihl= 5L
 tos= 0x0
 len= 84
 id= 15714
 flags= DF
 frag= 0L
 ttl= 64
 proto= icmp
 chksum= 0xaa8e
 src= 192.168.225.146
 dst= 172.217.3.164
 options
###[ ICMP ]###
 type= echo-request
 code= 0
 chksum= 0xe1cf
 id= 0xaa67
 seq= 0x1
###[ Raw ]###
 load=
'xd6xbfxb1Xx00x00x00x00x1axdcnx00x00x00x00x00x10x11x12x13x14x15x16x17x18x19x1ax1bx1
 !"#$%&'()*+,-./01234567'
None
```

Let's continue on and see how we can use Scapy for some of the common security testing.

# The TCP port scan

The first step for any potential hackers is almost always try to learn which service is open on the network, so they can concentrate their effort on the attack. Of course, we need to open certain ports in order to service our customer, but we should also close any open port that is not necessary to decrease the risk. We can use Scapy to do a simple open port scan.

We can send a `SYN` packet and see whether the server will return with `SYN-ACK`:

```
>>> p = sr1(IP(dst="10.0.0.14")/TCP(sport=666,dport=23,flags="S"))
>>> p.show()
###[ IP ]###
 version= 4L
 ihl= 5L
 tos= 0x0
 len= 40
 id= 25373
 flags= DF
 frag= 0L
 ttl= 62
 proto= tcp
 chksum= 0xc59b
 src= 10.0.0.14
 dst= 10.0.0.10
 options
###[ TCP ]###
 sport= telnet
 dport= 666
 seq= 0
 ack= 1
 dataofs= 5L
 reserved= 0L
 flags= RA
 window= 0
 chksum= 0x9907
 urgptr= 0
 options= {}
```

Notice that in the output here, the server is responding with a RESET+ACK for TCP port `23`. However, TCP port `22` (SSH) is open, therefore a SYN-ACK is returned:

```
>>> p = sr1(IP(dst="10.0.0.14")/TCP(sport=666,dport=22,flags="S"))
>>> p.show()
###[ IP ]###
```

```
 version= 4L
<skip>
 proto= tcp
 chksum= 0x28b5
 src= 10.0.0.14
 dst= 10.0.0.10
 options
###[ TCP ]###
 sport= ssh
 dport= 666
<skip>
 flags= SA
<skip>
```

We can also scan a range of destination ports from 20 to 22; note that we are using `sr()` for send-receive instead of the `sr1()` send-receive-one-packet:

```
>>> ans,unans = sr(IP(dst="10.0.0.14")/TCP(sport=666,dport=(20,22),flags="S"))
>>> for i in ans:
...     print i
...
(<IP frag=0 proto=tcp dst=10.0.0.14 |<TCP sport=666 dport=ftp_data flags=S |>>, <IP
version=4L ihl=5L tos=0x0 len=40 id=4126 flags=DF frag=0L ttl=62 proto=tcp
chksum=0x189b src=10.0.0.14 dst=10.0.0.10 options=[] |<TCP sport=ftp_data dport=666
seq=0 ack=1 dataofs=5L reserved=0L flags=RA window=0 chksum=0x990a urgptr=0 |>>)
(<IP frag=0 proto=tcp dst=10.0.0.14 |<TCP sport=666 dport=ftp flags=S |>>, <IP
version=4L ihl=5L tos=0x0 len=40 id=4127 flags=DF frag=0L ttl=62 proto=tcp
chksum=0x189a src=10.0.0.14 dst=10.0.0.10 options=[] |<TCP sport=ftp dport=666
seq=0 ack=1 dataofs=5L reserved=0L flags=RA window=0 chksum=0x9909 urgptr=0 |>>)
(<IP frag=0 proto=tcp dst=10.0.0.14 |<TCP sport=666 dport=ssh flags=S |>>, <IP
version=4L ihl=5L tos=0x0 len=44 id=0 flags=DF frag=0L ttl=62 proto=tcp
chksum=0x28b5 src=10.0.0.14 dst=10.0.0.10 options=[] |<TCP sport=ssh dport=666
seq=4187384571 ack=1 dataofs=6L reserved=0L flags=SA window=29200 chksum=0xaaab
urgptr=0 options=[('MSS', 1460)] |>>)
>>>
```

We can also specify a destination network instead of a single host. As you can see from the `10.0.0.8/29` block, hosts `10.0.0.9`, `10.0.0.13`, and `10.0.0.14` returned with SA, which corresponds to the two network devices and the host:

```
>>> ans,unans = sr(IP(dst="10.0.0.8/29")/TCP(sport=666,dport=(22),flags="S"))
>>> for i in ans:
...     print(i)
...
(<IP frag=0 proto=tcp dst=10.0.0.9 |<TCP sport=666 dport=ssh flags=S |>>, <IP
version=4L ihl=5L tos=0x0 len=44 id=7304 flags= frag=0L ttl=64 proto=tcp
chksum=0x4a32 src=10.0.0.9 dst=10.0.0.10 options=[] |<TCP sport=ssh dport=666
seq=541401209 ack=1 dataofs=6L reserved=0L flags=SA window=17292 chksum=0xfd18
urgptr=0 options=[('MSS', 1444)] |>>)
(<IP frag=0 proto=tcp dst=10.0.0.14 |<TCP sport=666 dport=ssh flags=S |>>, <IP
version=4L ihl=5L tos=0x0 len=44 id=0 flags=DF frag=0L ttl=62 proto=tcp
chksum=0x28b5 src=10.0.0.14 dst=10.0.0.10 options=[] |<TCP sport=ssh dport=666
seq=4222593330 ack=1 dataofs=6L reserved=0L flags=SA window=29200 chksum=0x6a5b
urgptr=0 options=[('MSS', 1460)] |>>)
```

```
(<IP frag=0 proto=tcp dst=10.0.0.13 |<TCP sport=666 dport=ssh flags=S |>>, <IP
version=4L ihl=5L tos=0x0 len=44 id=41992 flags= frag=0L ttl=254 proto=tcp
chksum=0x4ad src=10.0.0.13 dst=10.0.0.10 options=[] |<TCP sport=ssh dport=666
seq=2167267659 ack=1 dataofs=6L reserved=0L flags=SA window=4128 chksum=0x1252
urgptr=0 options=[('MSS', 536)] |>>)
```

Based on what you have learned so far, you can make a simple script for reusability, `scapy_tcp_scan_1.py`. We start with the suggested importing of Scapy and the sys module for taking in arguments:

```
  #!/usr/bin/env python2

  from scapy.all import *
  import sys
```

The `tcp_scan()` function is similar to what we have seen up to this point:

```
  def tcp_scan(destination, dport):
      ans, unans = sr(IP(dst=destination)/TCP(sport=666,dport=dport,flags="S"))
      for sending, returned in ans:
          if 'SA' in str(returned[TCP].flags):
              return destination + " port " + str(sending[TCP].dport) + " is open"
          else:
              return destination + " port " + str(sending[TCP].dport) + " is not
open"
```

We can then acquire the input from arguments, and then call the `tcp_scan()` function in `main()`:

```
  def main():
      destination = sys.argv[1]
      port = int(sys.argv[2])
      scan_result = tcp_scan(destination, port)
      print(scan_result)

  if __name__ == "__main__":
      main()
```

Remember that Scapy requires root access, therefore our script needs to be executed as `sudo`:

```
cisco@Client:~$ sudo python scapy_tcp_scan_1.py "10.0.0.14" 23
<skip>
10.0.0.14 port 23 is not open
cisco@Client:~$ sudo python scapy_tcp_scan_1.py "10.0.0.14" 22
<skip>
10.0.0.14 port 22 is open
```

This was a relatively lengthy example of the TCP scan, which demonstrated the power of crafting your own packet with Scapy. It tests out in the interactive mode, and then finalizes the usage with a simple script. Let's look at some more examples of Scapy's usage for security testing.

## The ping collection

Let's say our network contains a mix of Windows, Unix, and Linux machines with users adding their own **Bring Your Own Device** (**BYOD**); they may or may not support ICMP ping. We can now construct a file with three types of common pings for our network, the ICMP, TCP, and UDP pings in `scapy_ping_collection.py`:

```
#!/usr/bin/env python2

from scapy.all import *

def icmp_ping(destination):
    # regular ICMP ping
    ans, unans = sr(IP(dst=destination)/ICMP())
    return ans

def tcp_ping(destination, dport):
    # TCP SYN Scan
    ans, unans = sr(IP(dst=destination)/TCP(dport=dport,flags="S"))
    return ans

def udp_ping(destination):
    # ICMP Port unreachable error from closed port
    ans, unans = sr(IP(dst=destination)/UDP(dport=0))
    return ans
```

In this example, we will also use `summary()` and `sprintf()` for the output:

```
def answer_summary(answer_list):
 # example of lambda with pretty print
    answer_list.summary(lambda(s, r): r.sprintf("%IP.src% is alive"))
```

TIP *If you were wondering what a lambda is from the `answer_summary()` function mentioned previously, it is a way to create a small anonymous function; it is a function without a name. More information on it can be found at https://docs.python.org/3.5/tutorial/controlflow.html#lambda-expressions.*

We can then execute all the three types of pings on the network in one script:

```
def main():
    print("** ICMP Ping **")
    ans = icmp_ping("10.0.0.13-14")
    answer_summary(ans)
    print("** TCP Ping **")
    ans = tcp_ping("10.0.0.13", 22)
    answer_summary(ans)
    print("** UDP Ping **")
    ans = udp_ping("10.0.0.13-14")
    answer_summary(ans)

if __name__ == "__main__":
    main()
```

At this point, hopefully you will agree with me that by having the ability to construct your own packet, you can be in charge of the type of operations and tests that you would like to run.

## Common attacks

In this example for the Scapy usage, let's look at how we can construct our packet to conduct some of the class attacks, such as *Ping of Death* (https://en.wikipedia.org/wiki/Ping_of_death) and *Land Attack* (https://en.wikipedia.org/wiki/Denial-of-service_attack). This is perhaps something that you previously paid a commercial software for penetration testing. With Scapy, you can conduct the test while maintaining full control as well as adding more tests in the future.

The first attack basically sends the destination host with a bogus IP header, such as the length of 2 and the IP version 3:

```
def malformed_packet_attack(host):
    send(IP(dst=host, ihl=2, version=3)/ICMP())
```

The *Ping of Death* attack consists of the regular ICMP packet with a payload bigger than 65,535 bytes:

```
 def ping_of_death_attack(host):
     # https://en.wikipedia.org/wiki/Ping_of_death
     send(fragment(IP(dst=host)/ICMP()/("X"*60000)))
```

The *Land Attack* wants to redirect the client response back to the client itself and exhausted the host's resources:

```
def land_attack(host):
    # https://en.wikipedia.org/wiki/Denial-of-service_attack
    send(IP(src=host, dst=host)/TCP(sport=135,dport=135))
```

These are pretty old vulnerabilities or classic attacks that modern operating system are no longer susceptible to. For our Ubuntu 14.04 host, none of the preceding attacks will bring it down. However, as more security issues are being discovered, Scapy is a great tool to start tests against our own network and host without having to wait for the impacted vendor to give you a validation tool. This is especially true for the zero-day (published without prior notification) attacks that seem to be more and more common on the Internet.

## Scapy resources

We have spent quite a bit of effort working with Scapy in this chapter, and this is due to how highly I personally think of the tool. I hope you agree with me that Scapy is a great tool to keep in your toolset as a network engineer. The best part about Scapy is that it is constantly being developed with an engaged community of users.

TIP *I would highly recommend at least going through the Scapy tutorial at http://scapy.readthedocs.io/en/latest/usage.html#interactive-tutorial, as well as any of the documentation that is of interest to you.*

# Access lists

The network access lists are usually the first line of defense against outside intrusions and attacks. Generally speaking, routers, and switches process packets at a much faster rate than servers, because they utilize hardware such as **Ternary Content-Addressable Memory** (**TCAM**). They do not need to see the application layer information, rather just examine the layer 3 and layer 4 information, and decide whether the packets can be forwarded on or not. Therefore, we generally utilize network device access lists as the first step in safeguarding our network resources.

As a rule of thumb, we want to place access lists as close to the source as possible. Inherently, we also trust the inside host and distrust the clients outside of our network boundary. The access list is therefore usually placed on the inbound direction on the external facing network interface(s). In our lab scenario, this means we will place an inbound access list at Ethernet2/2 that is directly connected to the client host.

If you are unsure of the direction and placement of the access list, a few points might help here:

- Think of the access list from the perspective of the network device
- Simplify the packet in terms of just source and destination IP and use one host as an example:
    - In our lab, traffic from our server will have a source IP of `10.0.0.14` with the destination IP of `10.0.0.10`
    - The traffic from the client will have a source IP of 10.10.10.10 and the destination IP of `10.0.0.14`

Obviously, every network is different and how the access list should constructed depends on the services provided by your server. But as an inbound border access list, you must do the following:

- Deny RFC 3030 special-use address sources, that is `127.0.0.0/8`

- Deny RFC 1918 space, that is `10.0.0.0/8`
- Deny our own space as source, in this case `10.0.0.12/30`
- Permit inbound TCP port `22` (SSH) and `80` (HTTP) to host `10.0.0.14`
- Deny everything else

# Implementing access lists with Ansible

The easiest way to implement this access list would be to use Ansible. We have already looked at Ansible in the last two chapters, but it is worth repeating the advantages of using Ansible in this scenario:

- **Easier management**: For a long access list, we are able to utilize the include statement to break the long access list into more manageable pieces. The smaller pieces can then be managed by other teams or service owners.
- **Idempotency**: We can schedule the playbook at a regular interval and only the necessary changes will be made.
- **Each task is explicit**: We can separate the construct of the entries as well as apply the access list to the proper interface.
- **Reusability**: In the future, if we add additional external facing interfaces, we just need to add the device to the list of devices for the access list.
- **Extensible**: You will notice that we can use the same playbook for constructing the access list and apply it to the right interface. We can start small and expand to separate playbooks in the future as needed.

The host file is pretty standard:

```
[nxosv-devices]
nx-osv-1 ansible_host=172.16.1.155 ansible_username=cisco ansible_password=cisco
```

We will declare the variables in the playbook for the time being:

```
---
- name: Configure Access List
  hosts: "nxosv-devices"
  gather_facts: false
  connection: local

  vars:
    cli:
      host: "{{ ansible_host }}"
      username: "{{ ansible_username }}"
      password: "{{ ansible_password }}"
      transport: cli
```

To save space, we will illustrate denying the RFC 1918 space only. Implementing the denying of RFC 3030 and our own space will be identical to the steps used for the RFP 1918 space. Note that we did not deny `10.0.0.0/8` in our playbook, because our configuration currently uses `10.0.0.0` network for addressing. Of course, we perform the single host permit first and deny `10.0.0.0/8` for completeness, but in this example, we just choose to omit it:

```
tasks:
  - nxos_acl:
      name: border_inbound
      seq: 20
      action: deny
      proto: tcp
      src: 172.16.0.0/12
      dest: any
      log: enable
      state: present
      provider: "{{ cli }}"
  - nxos_acl:
      name: border_inbound
      seq: 40
      action: permit
      proto: tcp
      src: any
      dest: 10.0.0.14/32
      dest_port_op: eq
      dest_port1: 22
      state: present
      log: enable
      provider: "{{ cli }}"
  - nxos_acl:
      name: border_inbound
      seq: 50
      action: permit
      proto: tcp
      src: any
      dest: 10.0.0.14/32
      dest_port_op: eq
      dest_port1: 80
      state: present
      log: enable
      provider: "{{ cli }}"
  - nxos_acl:
      name: border_inbound
      seq: 60
      action: permit
      proto: tcp
      src: any
      dest: any
      state: present
      log: enable
      established: enable
      provider: "{{ cli }}"
  - nxos_acl:
      name: border_inbound
      seq: 1000
      action: deny
      proto: ip
      src: any
      dest: any
      state: present
      log: enable
      provider: "{{ cli }}"
```

Notice that we are allowing the established connection sourcing from the server inside to be allowed back in. We use the final explicit `deny ip any any` statement as a high sequence number (1000), so we can insert any new entries later on.

We can then apply the access list to the right interface:

```
- name: apply ingress acl to Ethernet 2/2
  nxos_acl_interface:
    name: border_inbound
    interface: Ethernet2/2
    direction: ingress
    state: present
    provider: "{{ cli }}"
```

*The access list on VIRL NX-OSv is only supported on the management interface. You will see this warning: Warning: ACL may not behave as expected since only management interface is supported. If you were to configure acl via CLI. This is okay for our lab, as our purpose is only to demonstrate the configuration automation of the access list.*

This might seem to be a lot of work for a single access list. For an experienced engineer, using Ansible to do this task will take longer than just logging into the device and configuring the access list. However, remember that this playbook can be reused many times in the future, so it will save you time in the long run.

It is in my experience that many times for a long access list, a few entries will be for one service, a few entries will be for another, and so on. The access lists tend to grow organically over time, and it becomes very hard to keep track of the origin and purpose of each entry. The fact that we can break them apart makes management of a long access list much simpler.

## MAC access lists

In the case where you have an L2 environment or where you are using non-IP protocols on Ethernet interfaces, you can still use a MAC address access list to allow or deny hosts based on MAC addresses. The steps are similar to the IP access list but more specific to MAC addresses. Recall that for MAC addresses, or physical addresses, the first 6 hexadecimal symbols belong to an **Organizationally Unique Identifier** (**OUI**). So, we can use the same access list matching pattern to deny a certain group of hosts.

*Note that we are testing this on IOSv with the `ios_config` module; due to the nature of the module, the change will be pushed out every single time the playbook is executed. Therefore, we lose one of the benefits of 'no change made unless necessary of Ansible.*

The host file and the top portion of the playbook are similar to the IP access list; the tasks portion is where the different modules and arguments are configured:

```
<skip>
  tasks:
    - name: Deny Hosts with vendor id fa16.3e00.0000
      ios_config:
        lines:
          - access-list 700 deny fa16.3e00.0000 0000.00FF.FFFF
          - access-list 700 permit 0000.0000.0000 FFFF.FFFF.FFFF
        provider: "{{ cli }}"
    - name: Apply filter on bridge group 1
      ios_config:
        lines:
          - bridge-group 1
          - bridge-group 1 input-address-list 700
        parents:
          - interface GigabitEthernet0/1
        provider: "{{ cli }}"
```

As more virtual networks become popular, the L3 information sometimes becomes transparent to the underlying virtual links. In these scenarios, the MAC access list becomes a good option if you need to restrict access on those links.

## The syslog search

There are plenty of documented network security breaches that took place over an extended period of time. In these slow breaches, often times evidence indicates that there were signs and traces in both the server and network logs that indicates suspicious activities. The undetected activities were not detected not because there was a lack of information, but rather there are too much information. The critical information that we were looking for are usually buried deep in a mountain of information that are hard to sort out.

*Besides syslog, **Uncomplicated Firewall (UFW)** is another great source of log information for servers. It is a frontend to iptable, which is a server firewall. UFW makes managing firewall rules very simple and logs good amount of information. See Other tools section for more information on UFW.*

In this section, we will try to use Python to search through the syslog text in order to detect the activities that we were looking for. Of course, the exact terms that we will search for depends on the device we are using. For example, Cisco provides a list of messages to look for in syslog for any the access list violation logging, available at http://www.cisco.com/c/en/us/about/security-center/identify-incidents-via-syslog.html.

*For more understanding of access control list logging, go to http://www.cisco.com/c/en/us/about/security-center/access-control-list-logging.html.*

For our exercise, we will use a Nexus switch anonymized syslog containing about 65,000 lines of log messages:

```
$ wc -l sample_log_anonymized.log
65102 sample_log_anonymized.log
```

We have inserted some syslog messages from the Cisco documentation, http://www.cisco.com/c/en/us/support/docs/switches/nexus-7000-series-switches/118907-configure-nx7k-00.

html, as the log message that we will be looking for:

```
2014 Jun 29 19:20:57 Nexus-7000 %VSHD-5-VSHD_SYSLOG_CONFIG_I: Configured from vty
by
 admin on console0
2014 Jun 29 19:21:18 Nexus-7000 %ACLLOG-5-ACLLOG_FLOW_INTERVAL: Src IP: 10.1
0.10.1,
 Dst IP: 172.16.10.10, Src Port: 0, Dst Port: 0, Src Intf: Ethernet4/1, Pro tocol:
 "ICMP"(1), Hit-count = 2589
2014 Jun 29 19:26:18 Nexus-7000 %ACLLOG-5-ACLLOG_FLOW_INTERVAL: Src IP: 10.1
0.10.1,
 Dst IP: 172.16.10.10, Src Port: 0, Dst Port: 0, Src Intf: Ethernet4/1, Pro tocol:
 "ICMP"(1), Hit-count = 4561
```

We will be using simple examples with regular expressions for our example. If you are already familiar with the regular expression in Python, feel free to skip the rest of the section.

## Searching with regular expressions

For our first search, we will simply use the regular expression module to look for the terms we are looking for. We will use a simple loop to the following:

```
#!/usr/bin/env python3

import re, datetime

startTime = datetime.datetime.now()

with open('sample_log_anonymized.log', 'r') as f:
   for line in f.readlines():
       if re.search('ACLLOG-5-ACLLOG_FLOW_INTERVAL', line):
           print(line)

endTime = datetime.datetime.now()
elapsedTime = endTime - startTime
print("Time Elapsed: " + str(elapsedTime))
```

The result is about 6/100th of a second to search through the log file:

```
$ python3 python_re_search_1.py
2014 Jun 29 19:21:18 Nexus-7000 %ACLLOG-5-ACLLOG_FLOW_INTERVAL: Src IP: 10.1
0.10.1,

2014 Jun 29 19:26:18 Nexus-7000 %ACLLOG-5-ACLLOG_FLOW_INTERVAL: Src IP: 10.1
0.10.1,

Time Elapsed: 0:00:00.065436
```

It is recommended to compile the search term for a more efficient search. It will not impact us much since we are already pretty fast. In fact, the Python interpretative nature will actually make it slower. However, it will make a difference when we search through larger text body, so let's make the change:

```
searchTerm = re.compile('ACLLOG-5-ACLLOG_FLOW_INTERVAL')

with open('sample_log_anonymized.log', 'r') as f:
   for line in f.readlines():
       if re.search(searchTerm, line):
           print(line)
```

The time result is actually slower:

```
Time Elapsed: 0:00:00.081541
```

Let's expand the example a bit. Assuming we have several files and multiple terms to search through, we will copy the original file to a new file:

```
$ cp sample_log_anonymized.log sample_log_anonymized_1.log
```

We will also include searching for the `PAM: Authentication failure` term. We will add another loop to search both the files:

```
term1 = re.compile('ACLLOG-5-ACLLOG_FLOW_INTERVAL')
term2 = re.compile('PAM: Authentication failure')

fileList = ['sample_log_anonymized.log', 'sample_log_anonymized_1.log']

for log in fileList:
    with open(log, 'r') as f:
       for line in f.readlines():
           if re.search(term1, line) or re.search(term2, line):
               print(line)
```

We can now see the difference in performance as well as expand our search capabilities:

```
$ python3 python_re_search_2.py
2016 Jun 5 16:49:33 NEXUS-A %DAEMON-3-SYSTEM_MSG: error: PAM: Authentication
failure for illegal user AAA from 172.16.20.170 - sshd[4425]

2016 Sep 14 22:52:26.210 NEXUS-A %DAEMON-3-SYSTEM_MSG: error: PAM: Authentication
failure for illegal user AAA from 172.16.20.170 - sshd[2811]

<skip>

2014 Jun 29 19:21:18 Nexus-7000 %ACLLOG-5-ACLLOG_FLOW_INTERVAL: Src IP: 10.1
0.10.1,

2014 Jun 29 19:26:18 Nexus-7000 %ACLLOG-5-ACLLOG_FLOW_INTERVAL: Src IP: 10.1
0.10.1,

<skip>

Time Elapsed: 0:00:00.330697
```

Of course, when it comes to performance tuning, it is a never ending race to zero and sometimes depends on the hardware you are using. But the important point is to regularly perform audits of your log files using Python, so you can catch the early signals of any potential breach.

## Other tools

There are other network security tools that we can use and automate with Python. Let's take a look at a few of them.

# Private VLANs

**Virtual Local Area Networks** (**VLANs**), has been around for a long time. They are essentially a broadcast domain where all hosts can be connected to a single switch, but are petitioned out to different domains, so we can separate the hosts out according to which host can see others via broadcasts. The reality is that most of the time, VLANs are mapped out to IP subnets. For example, in an enterprise building, I would likely have one IP subnet per physical floor, `192.168.1.0/24` for the first floor, `192.168.2.0/24` for the second floor. In this pattern, we use 1 /24 block for each floor. This gives a clear delineation of my physical network as well as my logical network. All hosts wanting to communicate beyond its own subnet will need to traverse through its layer 3 gateway, where I can use an access list to enforce security.

What happens when different departments resides on the same floor? Perhaps the finance and sales teams are both on the second floor, and I would not want the sales team's hosts in the same broadcast domain as the finance team's. I can further break the subnet down more, but that might become tedious and breaks the standard subnet scheme that was previously set up. This is a where private VLAN can help.

The private VLAN essentially breaks up the existing VLAN into sub-VLANs. There are three categories within a private VLAN:

- **The Promiscuous (P) port**: This port is allowed to send and receive layer 2 frames from any other port on the VLAN; this usually belongs to the port connecting to the layer 3 router
- **The Isolated (I) port**: This port is only allowed to communicated with P ports, ang they are typically connected to hosts when you do not want it to communicate with other hosts in the same VLAN.
- **The Community (C) port**: They are allowed to communicate with other C ports in the same community and P ports

We can again use Ansible or any of the other Python scripts introduced so far to accomplish this task. By now, we should have enough practice and confidence to implement this feature if needed using automation, so I will not repeat the steps here. Being aware of the private VLAN feature would come in handy at times when you need to isolate ports even further in a L2 VLAN.

# UFW with Python

We briefly mentioned UFW as the frontend for iptable on Ubuntu hosts. Here is a quick overview:

```
$ sudo apt-get install ufw
$ sudo ufw status
$ sudo ufw default outgoing
$ sudo ufw allow 22/tcp
$ sudo ufw allow www
$ sudo ufw default deny incoming
```

We can see the status of UFW:

```
$ sudo ufw status verbose
Status: active
Logging: on (low)
Default: deny (incoming), allow (outgoing), disabled (routed)
New profiles: skip

To Action From
-- ------ ----
22/tcp ALLOW IN Anywhere
80/tcp ALLOW IN Anywhere
22/tcp (v6) ALLOW IN Anywhere (v6)
80/tcp (v6) ALLOW IN Anywhere (v6)
```

As you can see, the advantage of UFW is a simple interface to construct otherwise complicated IP table rules. There are several Python-related tools we can use with UFW to make things even simpler:

- We can use the Ansible UFW module to streamline our operations, http://docs.ansible.com/ansible/ufw_module.html. Because Ansible is written in Python, we can go further and examine what is inside the Python module source code, https://github.com/ansible/ansible/blob/devel/lib/ansible/modules/system/ufw.py.
- There are Python wrapper modules around UFW as an API, https://gitlab.com/dhj/easyufw. This can make integration easier if you need to dynamically modify UFW rules based on certain events.
- UFW itself is written in Python, https://launchpad.net/ufw. Therefore, you can use the existing Python knowledge if you ever need to extend the current command sets.

UFW proves to be a good tool to safeguard your network server.

## Summary

In this chapter, we looked at network security with Python. We used the Cisco VIRL tool to set up our lab with both hosts and network equipments of NX-OSv and IOSv. We then took a tour around Scapy, which allows us to construct packets from the ground up. Scapy can be used in the interactive mode for quick testing, once completed in interactive mode, we can put the steps a file for more scalable testing. It can be used to perform various network penetration testing for known vulnerabilities.

We also looked at how we can use both an IP access list as well as a MAC list to protect our network. They are usually the first line of defense in our network protection. Using Ansible, we are able to deploy access liss consistently and quickly to multiple devices.

syslog and other log files  contain useful information that we should regularly comb through to detect any early signs of breach. Using Python regular expressions, we can systematically search for known log entries that can point us to security events that requires our attention. Besides the tools we have discussed, private VLAN and UFW are among the other useful tools that we can use for more security protection.

In the next chapter, we will look at how to use Python for network monitoring. Monitoring allows us to know what is happening in our network and the state of the network.

# Network Monitoring with Python - Part 1

Imagine you get a call at 2 a.m. in the morning. The person on the other endÂ asks, *Hi, we are facing a difficult issue that is impacting production. We think it might be network-related. Can you check for us?*Â Where would you check first? Of course, you would look at your monitoring tool and confirm whether any of the metrics changed in the last few hours. Throughout the book, we have been discussing various ways to programmatically make predictable changes to our network, with the goal of keeping the network running as smoothly as possible.Â

However, networks are not static. Far from it, they are probably one of the most fluent parts of the entire infrastructure. By definition, aÂ network connects differentÂ parts together, constantly passing traffic back and forth. There are lots of moving parts that can cause your network to stop working as expected:Â hardwareÂ fails, software with bugs, human mistakes, despite their best intentions, and so on. We need ways to make sure our network works as expected and that we are hopefully notified when things are not.Â

In the next two chapters, we will look at various ways to perform network monitoring tasks. Many of the tools we have looked at thus far can be tied together or directly managed by Python. Like everything we have looked at up to this point, network monitoring has to do with two parts. First, we need to know with what information the equipment is capable of transmitting. Second, we need to identify what useful information we can interpret from them.

We will look at a few tools that allow us to monitor the network effectively:Â

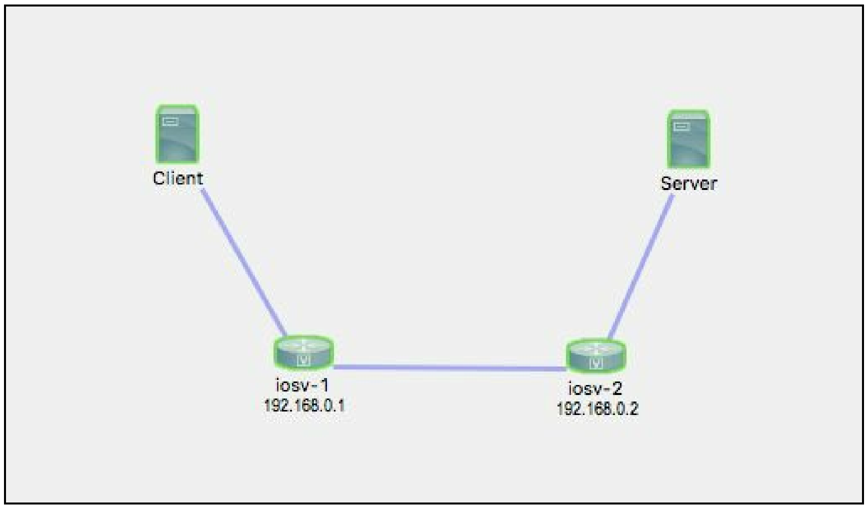
- **Simple Network Management Protocol** (SNMP)
- Matplotlib and pygal visualization

- MRTG and Cacti

This list is not exhaustive, and there is certainly no lack of commercial vendors in the space. The basics of network monitoring that we will look at, however, carry well for both open source and commercial tools.Â

## Lab setup

The lab for this chapter is similar to the one in the last chapter but with this difference: both the network devices are IOSv devices. Here's an illustration of this:

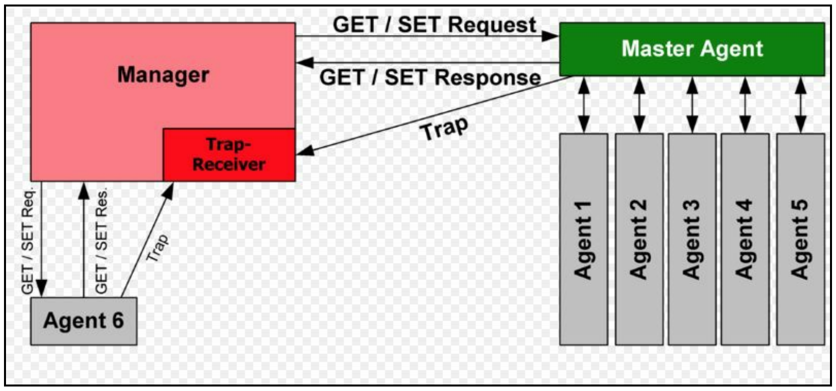


The two Ubuntu hosts will be used to generate traffic across the network so we can look at some non-zero counters.

# SNMP

SNMP is a standardized protocol used to collect and manage devices. Although the standard allows you to use SNMP for device management, in my experience, most network administrators prefer to keep SNMP as an information collection mechanism only. Since SNMP operates on UDP that is connectionless and considering the relatively weak security mechanism in versions 1 and 2, making device changes via SNMP tend to make network operators a bit uneasy. SNMP Version 3 has added cryptographic security and new concepts and terminologies to the protocol, but the way it's adapted varies among network device vendors.

SNMP is widely used in network monitoring and has been around since 1988 and was part of RFC 1065. The operations are straightforward with the network manager sending GET and SET requests toward the device and the device with the SNMP agent responding with the information per request. The most widely adapted standard is SNMPv2c, which is defined in RFC 1901 - RFC 1908. It uses a simple community-based security scheme for security. It has also introduced new features, such as the ability to get bulk information. The diagram below display the high-level operation for SNMP.



SNMP operations (source: https://en.wikipedia.org/wiki/Simple_Network_Management_Protocol)

The information residing in the device is structured in the **Management Information Base** (**MIB**). The MIB uses a hierarchical namespace containing an **Object Identifier** (**OID**), which represents the information that can be read and fed back to the requester. When we talk about using SNMP to query device information, we are really talking about using the management station to query the specific OID that represents the information we are after. You're required to put some efforts for consolidating basic common information into a common OID structure; however, the output of the effort varies in terms of how successful it is. At least in my experience, I typically need to consult with vendor documentation to find the OID that I need.

Some of the main points to take away from the operation are:

- The implementation heavily relies on the amount of information the device agent can provide. This, in turn, relies on how the vendor treats SNMP: as a core feature or an added feature.
- SNMP agents generally require CPU cycles from the control plane to return a value. Not only is this inefficient for devices with, say, large BGP tables, it is also not feasible to use SNMP to constantly query the data.
- The user needs to know the OID in order to query the data.

Since SNMP has been around for a while, my assumption is that you have some experience with it already. Let's jump directly into our first example.

## Setup

First, let's make sure the SNMP managing device and agent works in our setup. The SNMP bundle can be installed on either the hosts in our lab or the managing device on the management network. As long as the manager has IP reachability to the device and the managed device allows the connection, SNMP should work well.

In my setup, I have installed SNMP on both the Ubuntu host on the management network as well as the client host in the lab to test security:

```
$ sudo apt-get install snmp
```

There are many optional parameters you can configure on the network device, such as contact, location, chassis ID, and SNMP packet size. The options are device-specific and you should check the documentation on your device. For IOSv devices, we will configure an access list to limit only the desired host for querying the device as well as tying the access list with the SNMP community string. In our case, we will use the word `secret` as the community string and `permit_snmp` as the access list name:

```
!
ip access-list standard permit_snmp
 permit 172.16.1.173 log
 deny any log
!
!
snmp-server community secret RO permit_snmp
!
```

The SNMP community string is acting as a shared password between the manager and the agent; therefore, it needs to be included anytime you want to query the device.

We can use tools, such as the MIB locater (http://tools.cisco.com/ITDIT/MIBS/servlet/index), for finding specific OIDs to query. Alternatively, we can just start walking through the SNMP tree, starting from the top of Cisco's enterprise tree at `.1.3.6.1.4.1.9`:

```
$ snmpwalk -v2c -c secret 172.16.1.189 .1.3.6.1.4.1.9
iso.3.6.1.4.1.9.2.1.1.0 = STRING: "
Bootstrap program is IOSv
"
iso.3.6.1.4.1.9.2.1.2.0 = STRING: "reload"
iso.3.6.1.4.1.9.2.1.3.0 = STRING: "iosv-1"
iso.3.6.1.4.1.9.2.1.4.0 = STRING: "virl.info"
...
```

We can be very specific about the OID we need to query as well:

```
$ snmpwalk -v2c -c secret 172.16.1.189 .1.3.6.1.4.1.9.2.1.61.0
iso.3.6.1.4.1.9.2.1.61.0 = STRING: "cisco Systems, Inc.
170 West Tasman Dr.
San Jose, CA 95134-1706
U.S.A.
Ph +1-408-526-4000
Customer service 1-800-553-6387 or +1-408-526-7208
24HR Emergency 1-800-553-2447 or +1-408-526-7209
Email Address tac@cisco.com
World Wide Web http://www.cisco.com"
```

The last thing to check would be to make sure the access list would deny unwanted SNMP queries. Because we had the `log` keyword for both permit and deny entries, only `172.16.1.173` is permitted for querying the device:

```
*Mar 3 20:30:32.179: %SEC-6-IPACCESSLOGNP: list permit_snmp permitted 0
172.16.1.173 -> 0.0.0.0, 1 packet
*Mar 3 20:30:33.991: %SEC-6-IPACCESSLOGNP: list permit_snmp denied 0 172.16.1.187 -
> 0.0.0.0, 1 packet
```

As you can see, the biggest challenge in setting up SNMP is to find the right OID. Some of the OIDs are defined in standardized MIB-2; others are under the enterprise portion of the tree. Vendor documentation is the best bet, though. There are a number of tools that can help, such as the MIB Browser; you can add MIBs (again, provided by the vendors) to the browser and see the description of the enterprise-based OIDs. A tool such as Cisco's SNMP Object Navigator (http://snmp.cloudapps.cisco.com/Support/SNMP/do/BrowseOID.do?local=en) proves to be very valuable when you need to find the correct OID of the object you are looking for.

# PySNMP

PySNMP is a cross-platform, pure Python SNMP engine implementation developed by *Ilya Etingof* (https://github.com/etingof). It abstracts a lot of SNMP details for you, as great libraries do, and supports both Python 2 and Python 3.

PySNMP requires the PyASN1 package. According to Wikipedia:

*"ASN.1 is a standard and notation that describes rules and structures for representing, encoding, transmitting, and decoding data in telecommunication and computer networking. - https://asn1js.org/"*

PyASN1 conveniently provides a Python wrapper around ASN.1. Let's install the package first:

```
cd /tmp
git clone https://github.com/etingof/pyasn1.git
cd pyasn1/
sudo python3 setup.py install
```

Next, install the PySNMP package:

```
git clone https://github.com/etingof/pysnmp
cd pysnmp/
sudo python3 setup.py install
```

*At the time of writing this, there are some bug fixes that have not been pushed to the PyPI repository yet. The packages can also be installed via pip;* `sudo pip3 install pysnmp` *will automatically install pyasn1.*

Let's look at how to use PySNMP to query the same Cisco contact information we used in the previous example, which is slightly modified from the PySNMP example at http://pysnmp.sourceforge.net/faq/response-values-mib-resolution.html. We will import the necessary module and create a `CommandGenerator` object first:

```
>>> from pysnmp.entity.rfc3413.oneliner import cmdgen
>>> cmdGen = cmdgen.CommandGenerator()
>>> cisco_contact_info_oid = "1.3.6.1.4.1.9.2.1.61.0"
```

We can perform SNMP using the `getCmd` method. The result is unpacked into various variables; of these, we care most about `varBinds` that contain the query result:

```
>>> errorIndication, errorStatus, errorIndex, varBinds = cmdGen.getCmd(
...     cmdgen.CommunityData('secret'),
...     cmdgen.UdpTransportTarget(('172.16.1.189', 161)),
...     cisco_contact_info_oid
... )
>>> for name, val in varBinds:
...     print('%s = %s' % (name.prettyPrint(), str(val)))
...
SNMPv2-SMI::enterprises.9.2.1.61.0 = cisco Systems, Inc.
170 West Tasman Dr.
San Jose, CA 95134-1706
U.S.A.
Ph +1-408-526-4000
Customer service 1-800-553-6387 or +1-408-526-7208
24HR Emergency 1-800-553-2447 or +1-408-526-7209
Email Address tac@cisco.com
World Wide Web http://www.cisco.com
>>>
```

Note that the response values are PyASN1 objects. The `prettyPrint()` method will convert some of these values into a human-readable format, but since the result in our case was not converted, we will convert it into a string manually.

We can put a script using the preceding interactive example into `pysnmp_1.py` with the referenced error checking if we run into problems. We can also include multiple OIDs in the `getCmd()` method:

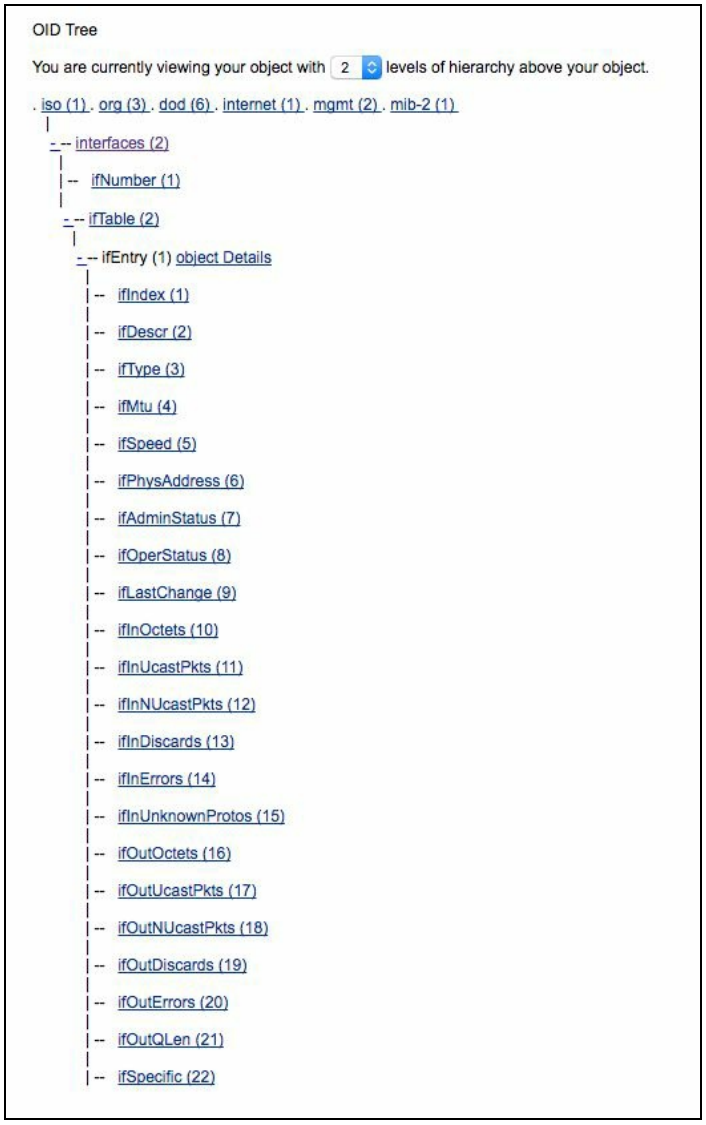
```
system_up_time_oid = "1.3.6.1.2.1.1.3.0"
cisco_contact_info_oid = "1.3.6.1.4.1.9.2.1.61.0"

errorIndication, errorStatus, errorIndex, varBinds = cmdGen.getCmd(
 cmdgen.CommunityData('secret'),
 cmdgen.UdpTransportTarget(('172.16.1.189', 161)),
 system_up_time_oid,
 cisco_contact_info_oid
)
```

The result will just be unpacked and listed out in our example:

```
$ python3 pysnmp_1.py
SNMPv2-MIB::sysUpTime.0 = 16052379
```

```
SNMPv2-SMI::enterprises.9.2.1.61.0 = cisco Systems, Inc.
170 West Tasman Dr.
....
```

In the next example, we will persist the values we received from the queries so we can perform other functions, such as visualization, with the data. For our example, we will use the `ifEntry` within the MIB-2 tree. You can find a number of resources mapping out the `ifEntry` tree; here is a screenshot of the Cisco SNMP Object Navigator site that we accessed before:



```
OID Tree

You are currently viewing your object with 2 levels of hierarchy above your object.

. iso (1) . org (3) . dod (6) . internet (1) . mgmt (2) . mib-2 (1)
  |
  - -- interfaces (2)
     |
     |--  ifNumber (1)
     |
     - -- ifTable (2)
        |
        - -- ifEntry (1) object Details
           |
           |--  ifIndex (1)
           |
           |--  ifDescr (2)
           |
           |--  ifType (3)
           |
           |--  ifMtu (4)
           |
           |--  ifSpeed (5)
           |
           |--  ifPhysAddress (6)
           |
           |--  ifAdminStatus (7)
           |
           |--  ifOperStatus (8)
           |
           |--  ifLastChange (9)
           |
           |--  ifInOctets (10)
           |
           |--  ifInUcastPkts (11)
           |
           |--  ifInNUcastPkts (12)
           |
           |--  ifInDiscards (13)
           |
           |--  ifInErrors (14)
           |
           |--  ifInUnknownProtos (15)
           |
           |--  ifOutOctets (16)
           |
           |--  ifOutUcastPkts (17)
           |
           |--  ifOutNUcastPkts (18)
           |
           |--  ifOutDiscards (19)
           |
           |--  ifOutErrors (20)
           |
           |--  ifOutQLen (21)
           |
           |--  ifSpecific (22)
```

SNMP ifEntry OID Tree

A quick test would illustrate the mapping of the interfaces on the device:

```
$ snmpwalk -v2c -c secret 172.16.1.189 .1.3.6.1.2.1.2.2.1.2
iso.3.6.1.2.1.2.2.1.2.1 = STRING: "GigabitEthernet0/0"
iso.3.6.1.2.1.2.2.1.2.2 = STRING: "GigabitEthernet0/1"
iso.3.6.1.2.1.2.2.1.2.3 = STRING: "GigabitEthernet0/2"
iso.3.6.1.2.1.2.2.1.2.4 = STRING: "Null0"
iso.3.6.1.2.1.2.2.1.2.5 = STRING: "Loopback0"
```

From the documentation, we can map the values of `ifInOctets(10)`, `ifInUcastPkts(11)`, `ifOutOctets(16)`, and `ifOutUcastPkts(17)`. A quick check on the value of `GigabitEthernet0/0` to the OID `1.3.6.1.2.1.2.2.1.17.1` can be compared to the command line and the SNMP. The values should be close but not exact since there might be some traffic on the wire:

```
# Command Line Output
iosv-1#sh int gig 0/0 | i packets
  5 minute input rate 0 bits/sec, 0 packets/sec
  5 minute output rate 0 bits/sec, 0 packets/sec
    38532 packets input, 3635282 bytes, 0 no buffer
    53965 packets output, 4723884 bytes, 0 underruns

# SNMP Output
$ snmpwalk -v2c -c secret 172.16.1.189 .1.3.6.1.2.1.2.2.1.17.1
iso.3.6.1.2.1.2.2.1.17.1 = Counter32: 54070
```

At the time of production, we will write the query results in a database.

To simplify our example, we will write the query values to a flat file. In `pysnmp_3.py`, we have defined various OIDs that we need to query:

```
  # Hostname OID
  system_name = '1.3.6.1.2.1.1.5.0'

  # Interface OID
  gig0_0_in_oct = '1.3.6.1.2.1.2.2.1.10.1'
  gig0_0_in_uPackets = '1.3.6.1.2.1.2.2.1.11.1'
  gig0_0_out_oct = '1.3.6.1.2.1.2.2.1.16.1'
  gig0_0_out_uPackets = '1.3.6.1.2.1.2.2.1.17.1'
```

The values were consumed in the `snmp_query()` function with the host, community, and OID as input:

```
  def snmp_query(host, community, oid):
      errorIndication, errorStatus, errorIndex, varBinds = cmdGen.getCmd(
      cmdgen.CommunityData(community),
      cmdgen.UdpTransportTarget((host, 161)),
      oid
      )
```

All the values are put in a dictionary with various keys and written to a file called `results.txt`:

```
result = {}
result['Time'] = datetime.datetime.utcnow().isoformat()
result['hostname'] = snmp_query(host, community, system_name)
result['Gig0-0_In_Octet'] = snmp_query(host, community, gig0_0_in_oct)
result['Gig0-0_In_uPackets'] = snmp_query(host, community, gig0_0_in_uPackets)
result['Gig0-0_Out_Octet'] = snmp_query(host, community, gig0_0_out_oct)
result['Gig0-0_Out_uPackets'] = snmp_query(host, community, gig0_0_out_uPackets)

with open('/home/echou/Master_Python_Networking/Chapter7/results.txt', 'a') as f:
    f.write(str(result))
    f.write('\n')
```

The outcome will be a file with results showing the number representing the point of time of the query:

```
# Sample output
$ cat results.txt
{'hostname': 'iosv-1.virl.info', 'Gig0-0_In_uPackets': '42005', 'Time': '2017-03-
06T02:11:54.989034', 'Gig0-0_Out_uPackets': '59733', 'Gig0-0_In_Octet': '3969762',
'Gig0-0_Out_Octet': '5199970'}
```

We can make this script executable and schedule a `cron` job for execution every 5 minutes:

```
$ chmod +x pysnmp_3.py

# Crontab configuration
*/5 * * * * /home/echou/Master_Python_Networking/Chapter7/pysnmp_3.py
```

As mentioned, in a production environment, we would put the information in a database. In a NoSQL database, we might use time as the primary index (or key) because it is always unique, followed by various key-value pairs.

We will wait for the script to be executed a few times and move on to see how we can use Python to visualize the data.

## Python visualization

We gather network data for the purpose of gaining insight into our network. One of the best ways to know what the data means is to visualize them with graphs. This is true for almost all data, but especially true for time series data in the context of network monitoring. How much data was transmitted on the wire in the last week? What is the percentage of the TCP protocol among all of the traffic? These are values we can glean from using data-gathering mechanisms, such as SNMP, and visualize with some of the popular Python libraries.

In this section, we will use the data we collected from the last section on SNMP and use two popular Python libraries--Matplotlib and Pygal--to graph them.

## Matplotlib

Matplotlib (http://matplotlib.org/) is a plotting library for the Python library and its NumPy mathematical extension. It can produce publication-quality figures, such as plots, histograms, and bar graphs with a few lines of code.

*NumPy is an extension of the Python programming language. It is open source and widely used in various data science projects. You can learn more about it at* https://en.wikipedia.org/wiki/NumPy.

## Installation

The installation can be done using the Linux package management system, depending on your distribution:

```
$ sudo apt-get install python-matplotlib
$ sudo apt-get install python3-matplotlib
```

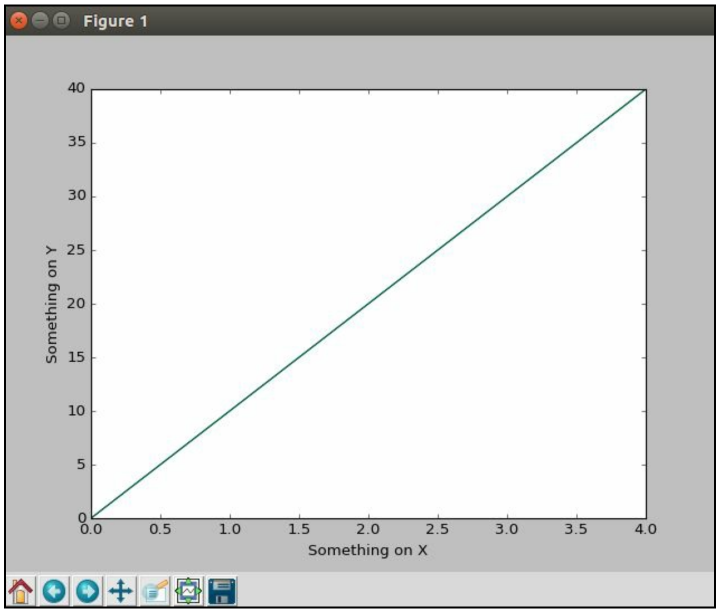
# Matplotlib - the first example

For the following examples, the figures are displayed as standard output by default; it is easier if you try them out while on standard output. If you have been following along the book via a virtual machine, it is recommended that you use the VM Window instead of SSH. If you do not have access to the standard output, you can save the figure and view it after you download it (as you will see soon). Note that you will need to set the `$DISPLAY` variable in some of the following graphs.

A line plot graph simply gives two lists of numbers that correspond to the *x* axis and *y* axis values:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot([0,1,2,3,4], [0,10,20,30,40])
[<matplotlib.lines.Line2D object at 0x7f932510df98>]
>>> plt.ylabel('Something on Y')
<matplotlib.text.Text object at 0x7f93251546a0>
>>> plt.xlabel('Something on X')
<matplotlib.text.Text object at 0x7f9325fdb9e8>
>>> plt.show()
```

The graph will show as a line graph:



Alternatively, if you do not have access to standard output or have saved the figure first, you can use the `savefig()` method:

```
>>> plt.savefig('figure1.png')
or
>>> plt.savefig('figure1.pdf')
```

With this basic knowledge of graphing plots, we can now graph the results we received from SNMP queries.

# Matplotlib for SNMP results

In our first example, namely `matplotlib_1.py`, we will import the *dates* module besides `pyplot`. We will use the matplotlib.dates module instead of Python standard library dates module because the way we use the module is how Matplotlib will convert the date value internally into the float that is required.

```
import matplotlib.pyplot as plt
import matplotlib.dates as dates
```

*Matplotlib provides sophisticated date plotting capabilities; you'll find more information on this at http://matplotlib.org/api/dates_api.html.*

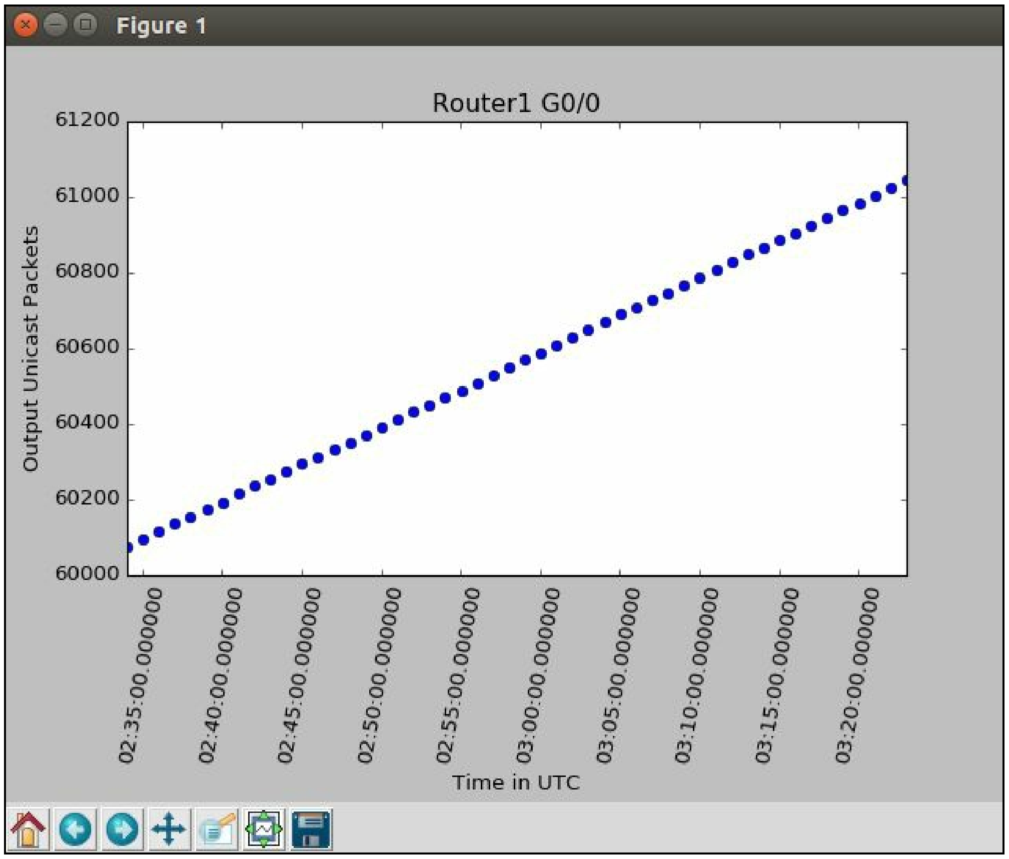
We will create two empty lists, each representing the *x* axis and *y* axis values. Note that on line 12, we used the built-in `eval()` Python to read the input as a dictionary instead of a default string:

```
x_time = []
y_value = []

with open('results.txt', 'r') as f:
    for line in f.readlines():
        line = eval(line)
        x_time.append(dates.datestr2num(line['Time']))
        y_value.append(line['Gig0-0_Out_uPackets'])
```

In order to read the *x* axis value back in a human-readable date format, we will need to use the `plot_date()` function instead of `plot()`. We will also tweak the size of the figure a bit as well as rotating the value on the *x* axis so we can read the value in full:

```
plt.subplots_adjust(bottom=0.3)
plt.xticks(rotation=80)

plt.plot_date(x_time, y_value)
plt.title('Router1 G0/0')
plt.xlabel('Time in UTC')
plt.ylabel('Output Unicast Packets')
plt.savefig('matplotlib_1_result.png')
plt.show()
```

The final result would display the Router1 Gig0/0 Unicast Output Packet, as follows:



Router1 Matplotlib Graph

Note that if you prefer a straight line instead of dots, you can use the third optional parameter in the `plot_date()` function:

```
    plt.plot_date(x_time, y_value, "-")
```

We can repeat the steps for the rest of the values for output octets, input unicast packets, and input as individual graphs. However, in our next example, that is `matplotlib_2.py`, we will show how to graph multiple values against the same time range, as well as additional Matplotlib options.

In this case, we will create additional lists and populate the values accordingly:

```
x_time = []
out_octets = []
```

```
out_packets = []
in_octets = []
in_packets = []

with open('results.txt', 'r') as f:
    for line in f.readlines():
...
        out_packets.append(line['Gig0-0_Out_uPackets'])
        out_octets.append(line['Gig0-0_Out_Octet'])
        in_packets.append(line['Gig0-0_In_uPackets'])
        in_octets.append(line['Gig0-0_In_Octet'])
```

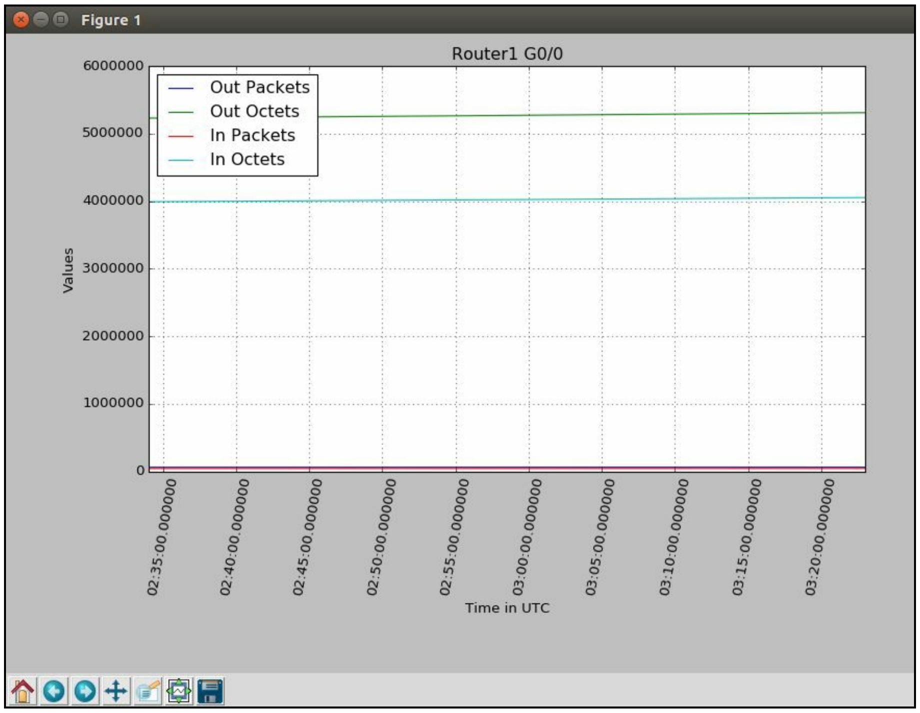
Since we have identical *x* axis values, we can just add the different *y* axis values to the same graph:

```
# Use plot_date to display x-axis back in date format
plt.plot_date(x_time, out_packets, '-', label='Out Packets')
plt.plot_date(x_time, out_octets, '-', label='Out Octets')
plt.plot_date(x_time, in_packets, '-', label='In Packets')
plt.plot_date(x_time, in_octets, '-', label='In Octets')
```

Also, add grid and legend to the graph:

```
plt.legend(loc='upper left')
plt.grid(True)
```

The final result will combine all the values in a single graph. Note that some of the values in the upper-left corner is blocked by the legend. You can resize the figure and/or use the pan/zoom option to move around the graph in order to see the value.

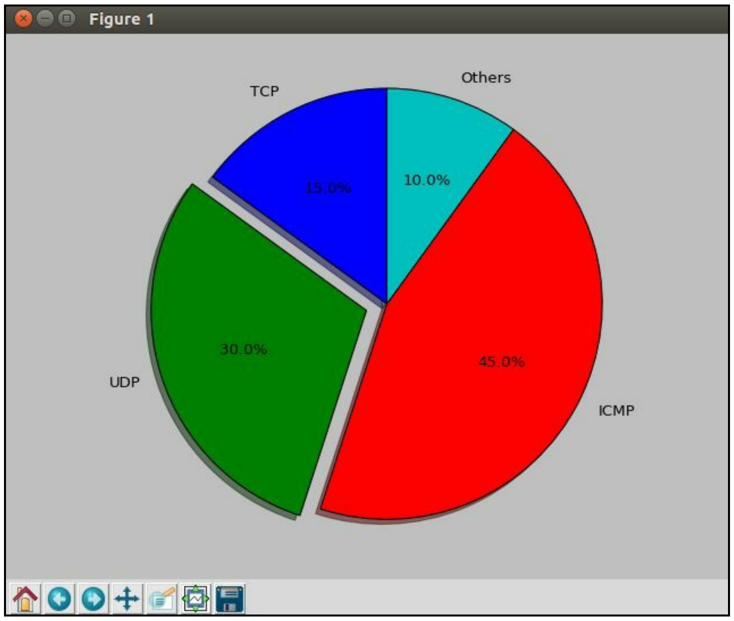


Router 1 - Matplotlib Multiline Graph

There are many more graphing options available in Matplotlib; we are certainly not limited to plot graphs. For example, we can use the following mock data to graph the percentage of different traffic types that we see on the wire:

```
#!/usr/bin/env python3
# Example from
http://matplotlib.org/2.0.0/examples/pie_and_polar_charts/pie_demo_features.html
import matplotlib.pyplot as plt

# Pie chart, where the slices will be ordered and plotted counter-clockwise:
labels = 'TCP', 'UDP', 'ICMP', 'Others'
sizes = [15, 30, 45, 10]
explode = (0, 0.1, 0, 0) # Make UDP stand out

fig1, ax1 = plt.subplots()
ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
 shadow=True, startangle=90)
ax1.axis('equal') # Equal aspect ratio ensures that pie is drawn as a circle.

plt.show()
```

The preceding code leads to this pie diagram from `plt.show()`:



Matplotlib Pie Graph

## Additional Matplotlib resources

Matplotlib is one of the best Python plotting libraries that produces publication-quality figures. Like Python, its aim is to make complex tasks simple. With over 4,800 stars on GitHub, it is also one of the most popular open source projects. It directly translates into bug fixes, user community, and general usability. It takes a bit of time to learn the package, but it is well worth the effort.

*In this section, we barely scratched the surface of Matplotlib. You'll find additional resources at http://matplotlib.org/2.0.0/index.html (the Matplotlib project page) and https://github.com/matplotlib/matplotlib (Matplotlib GitHub Repository).*

In the next section, we will take a look at another popular Python graph library: Pygal.

## Pygal

Pygal (http://www.pygal.org/) is a dynamic SVG charting library written in Python. The biggest advantage of Pygal, in my opinion, is it produces the **Scalable Vector Graphics** (**SVG**) format easily and natively. There are many advantages of SVG over other graph formats, but two of the main advantages are that it is web browser friendly and it provides scalability without sacrificing image quality. In other words, you can display the resulting image in any modern web browser and zoom in and out of the image without losing the details of the graph.

# Installation

The installation is done via pip:

```
$ sudo pip install pygal
$ sudo pip3 install pygal
```

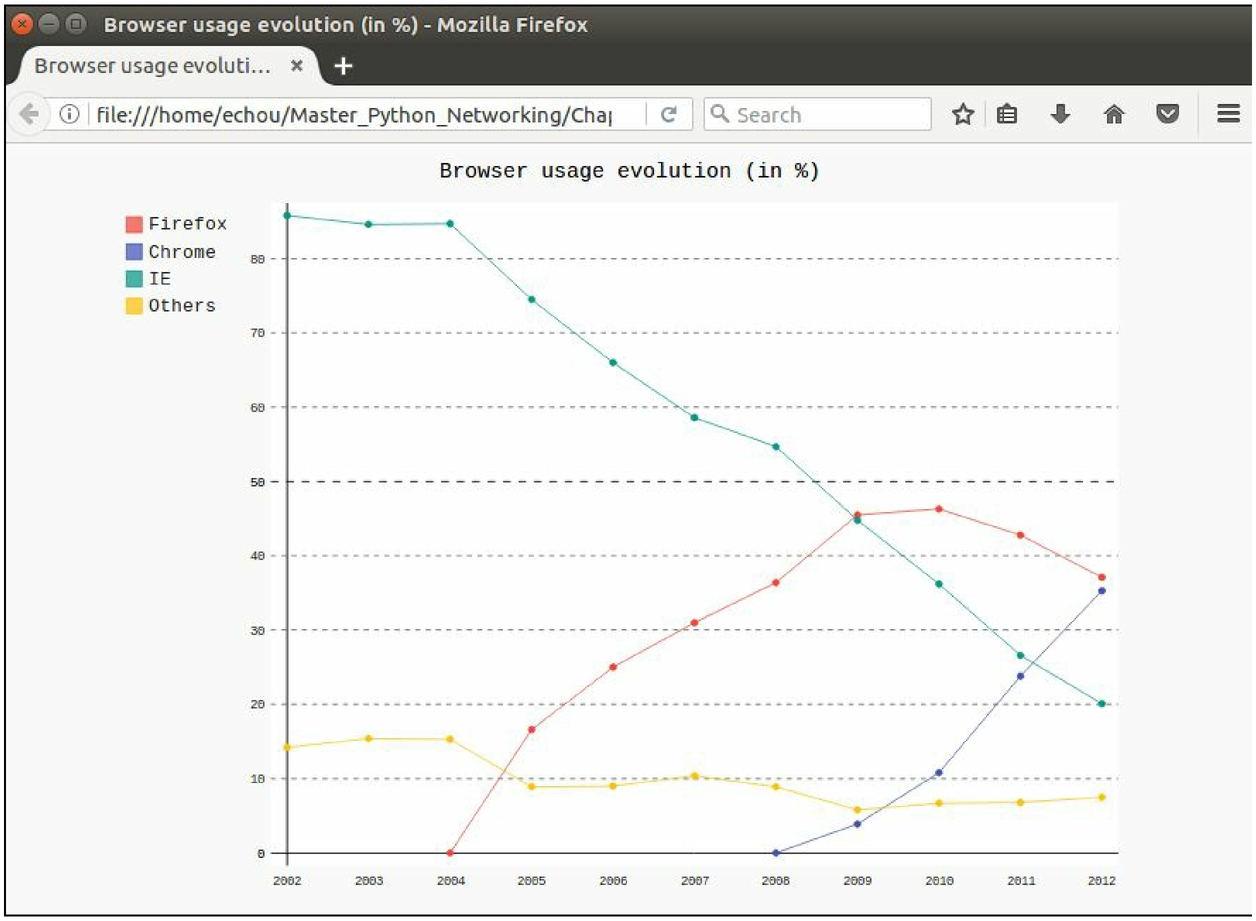
## Pygal - the first example

Let's look at the line chart example demonstrated on Pygal's documentation, available at http://pygal.org/en/stable/documentation/types/line.html:

```
>>> import pygal
>>> line_chart = pygal.Line()
>>> line_chart.title = 'Browser usage evolution (in %)'
>>> line_chart.x_labels = map(str, range(2002, 2013))
>>> line_chart.add('Firefox', [None, None, 0, 16.6, 25, 31, 36.4, 45.5, 46.3, 42.8,
37.1])
<pygal.graph.line.Line object at 0x7fa0bb009c50>
>>> line_chart.add('Chrome', [None, None, None, None, None, None, 0, 3.9, 10.8,
23.8, 35.3])
<pygal.graph.line.Line object at 0x7fa0bb009c50>
>>> line_chart.add('IE', [85.8, 84.6, 84.7, 74.5, 66, 58.6, 54.7, 44.8, 36.2, 26.6,
20.1])
<pygal.graph.line.Line object at 0x7fa0bb009c50>
>>> line_chart.add('Others', [14.2, 15.4, 15.3, 8.9, 9, 10.4, 8.9, 5.8, 6.7, 6.8,
7.5])
<pygal.graph.line.Line object at 0x7fa0bb009c50>
>>> line_chart.render_to_file('pygal_example_1.svg')
```

*In the example, we created a line object with the* `x_labels` *automatically rendered as strings for 11 units. Each of the objects can be added with the label and the value in a list format, such as Firefox, Chrome, and IE.*

Here's the resulting graph as viewed in Firefox:



Pygal Sample Graph

We can use the same method to graph the SNMP results we have in hand. We will do this in the next section.

# Pygal for SNMP results

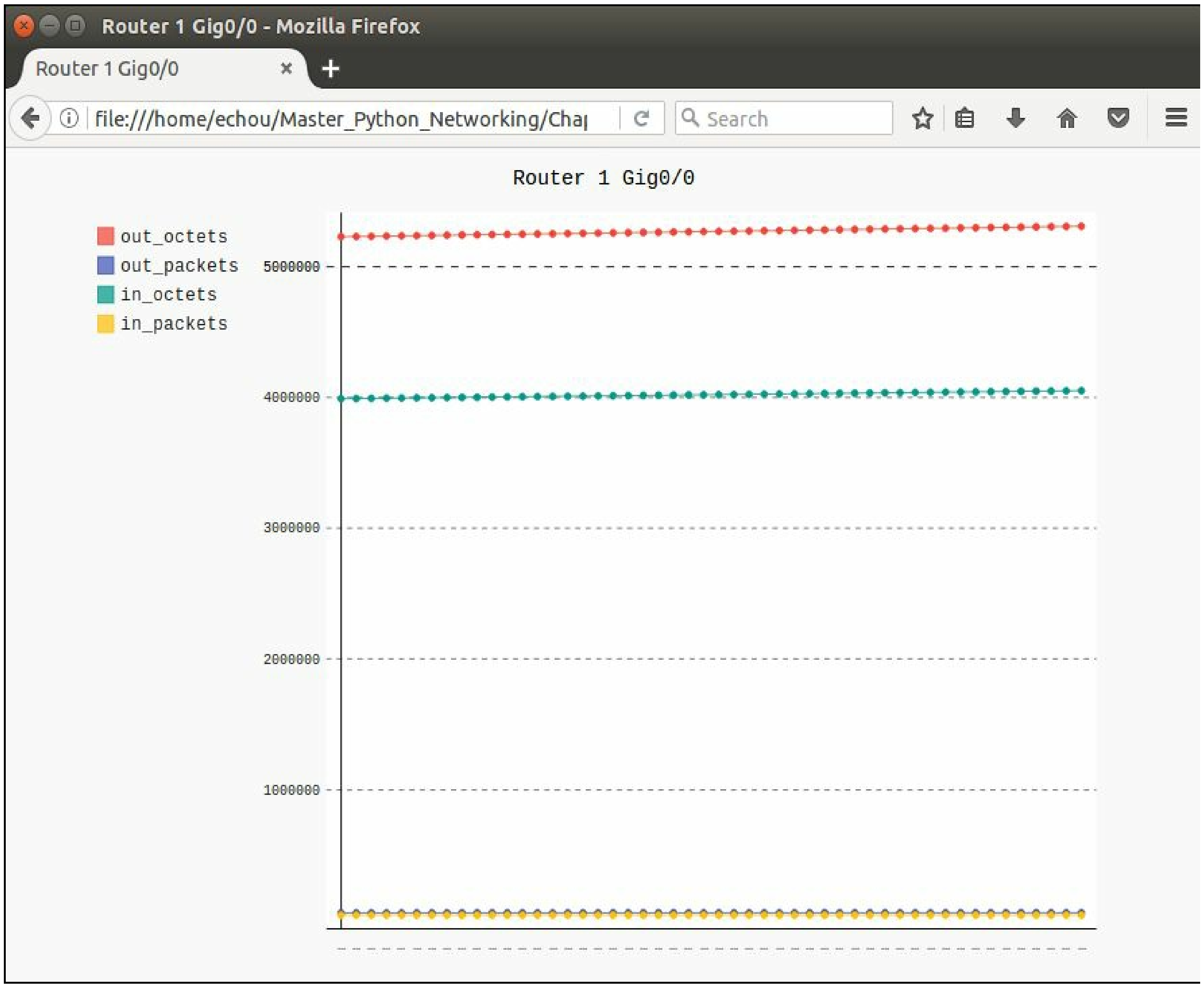
For the Pygal line graph, we can largely follow the same pattern as our Matplotlib example, where we create lists of values by reading the file. We no longer need to convert the *x* axis value into an internal float, as we did for Matplotlib; however, we do need to convert the numbers in each of the values we would have received in the float:

```
with open('results.txt', 'r') as f:
    for line in f.readlines():
        line = eval(line)
        x_time.append(line['Time'])
        out_packets.append(float(line['Gig0-0_Out_uPackets']))
        out_octets.append(float(line['Gig0-0_Out_Octet']))
        in_packets.append(float(line['Gig0-0_In_uPackets']))
        in_octets.append(float(line['Gig0-0_In_Octet']))
```

We can use the same mechanism that we saw to construct the line graph:

```
line_chart = pygal.Line()
line_chart.title = "Router 1 Gig0/0"
line_chart.x_labels = x_time
line_chart.add('out_octets', out_octets)
line_chart.add('out_packets', out_packets)
line_chart.add('in_octets', in_octets)
line_chart.add('in_packets', in_packets)
line_chart.render_to_file('pygal_example_2.svg')
```

The outcome is similar to what we have already seen, but this result is in SVG format that is easier for displaying on a web page. It can be viewed from a browser as follows:



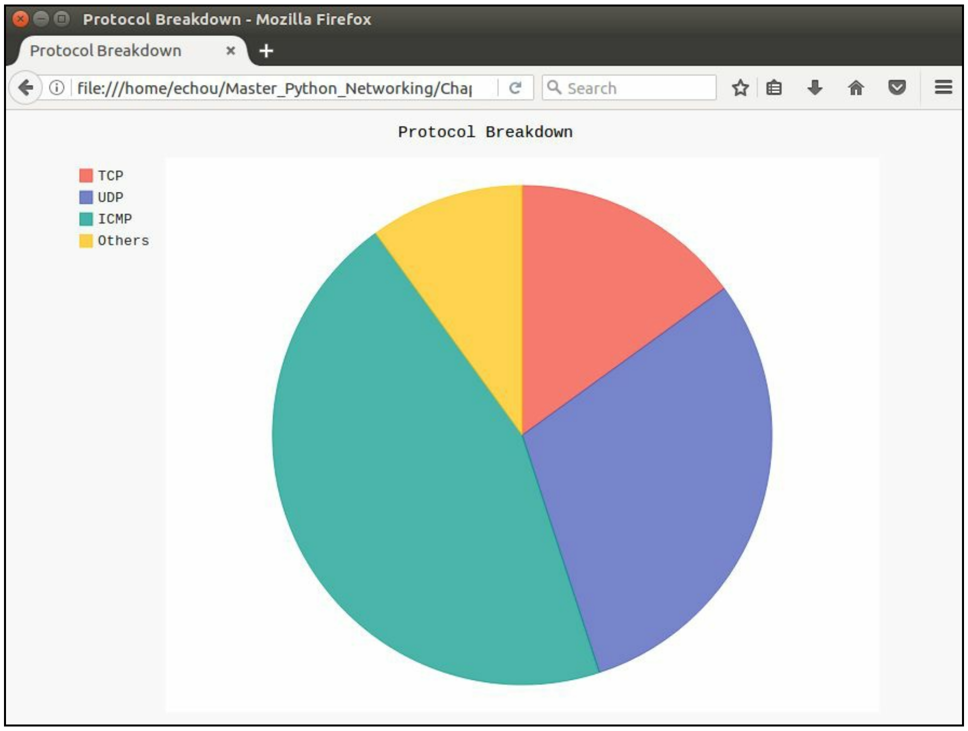
**Router 1 Pygal Multiline Graph**

Just like Matplotlib, Pygal provides many more options for graphs. For example, to regraph the pie chart we saw before in Pygal, we can use the `pygal.Pie()` object:

```
#!/usr/bin/env python3

import pygal

line_chart = pygal.Pie()
line_chart.title = "Protocol Breakdown"
line_chart.add('TCP', 15)
line_chart.add('UDP', 30)
line_chart.add('ICMP', 45)
line_chart.add('Others', 10)
line_chart.render_to_file('pygal_example_3.svg')
```

The resulting SVG file would be similar to the PNG generated by

Matplotlib:



Pygal Pie Graph

## Additional Pygal resources

Pygal provides many more customizable features and graphing capabilities for the data you collect from more basic network monitoring tools such as SNMP. We demonstrated a simple line graph and pie graphs here. You can find more information about the project here:

- Pygal documentation: http://www.pygal.org/en/stable/index.html
- Pygal GitHub project page: https://github.com/Kozea/pygal

In the next section, we will continue with the SNMP theme of network monitoring but with a fully featured network monitoring system called **Cacti**.

## Python for Cacti

In my early days, when I was working as a network engineer, we used the open source cross-platform **Multi Router Traffic Grapher** (**MRTG**, https://en.wikipedia.org/wiki/Multi_Router_Traffic_Grapher) tool to check the traffic load on network links. It was one of the first open source high-level network monitoring system that abstracted the details of SNMP, database, and HTML for network engineers. Then came **Round-Robin Database Tool** (**RRDtool**, https://en.wikipedia.org/wiki/RRDtool). In its first release in 1999, it was referred to as "MRTG done right". It had greatly improved the database and poller performance in the backend.

Released in 2001, Cacti (https://en.wikipedia.org/wiki/Cacti_(software)) is an open source web-based network monitoring and graphing tool designed as an improved frontend for RRDtool. Because of the heritage of MRTG and RRDtool, you will notice a familiar graph layout, templates, and SNMP poller. As a packaged tool, the installation and usage will need to stay within the boundary of the tool itself. However, Cacti offers the custom data query feature that we can use Python for. In this section, we will see how we can use Python as an input method for Cacti.

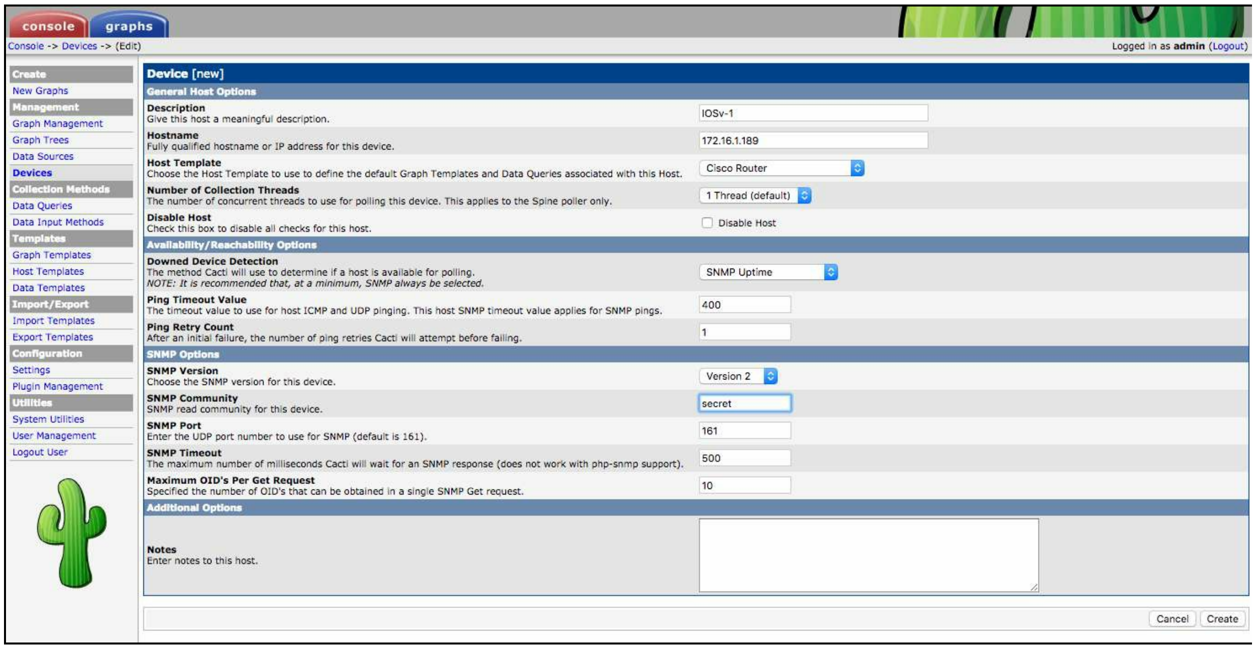
# Installation

Installation on Ubuntu is straightforward; install Cacti on the Ubuntu management VM:

```
$ sudo apt-get install cacti
```

It will trigger a series of installation and setup steps, including the MySQL database, web server (Apache or lighthttpd), and various configuration tasks. Once installed, navigate to *`http://<ip>/cacti`* to get started. The last step is to log in with the default username and password (`admin`/`admin`); you will be prompted to change the password.

Once you are logged in, you can follow the documentation to add a device and associate it with a template. There is a Cisco router premade template that you can go with. Cacti has good documentation on http://docs.cacti.net/ for adding a device and creating your first graph, so we will quickly look at some screenshots that you can expect:



**Device** [new]

**General Host Options**

| | |
|---|---|
| **Description** Give this host a meaningful description. | IOSv-1 |
| **Hostname** Fully qualified hostname or IP address for this device. | 172.16.1.189 |
| **Host Template** Choose the Host Template to use to define the default Graph Templates and Data Queries associated with this Host. | Cisco Router |
| **Number of Collection Threads** The number of concurrent threads to use for polling this device. This applies to the Spine poller only. | 1 Thread (default) |
| **Disable Host** Check this box to disable all checks for this host. | ☐ Disable Host |

**Availability/Reachability Options**

| | |
|---|---|
| **Downed Device Detection** The method Cacti will use to determine if a host is available for polling. *NOTE: It is recommended that, at a minimum, SNMP always be selected.* | SNMP Uptime |
| **Ping Timeout Value** The timeout value to use for host ICMP and UDP pinging. This host SNMP timeout value applies for SNMP pings. | 400 |
| **Ping Retry Count** After an initial failure, the number of ping retries Cacti will attempt before failing. | 1 |

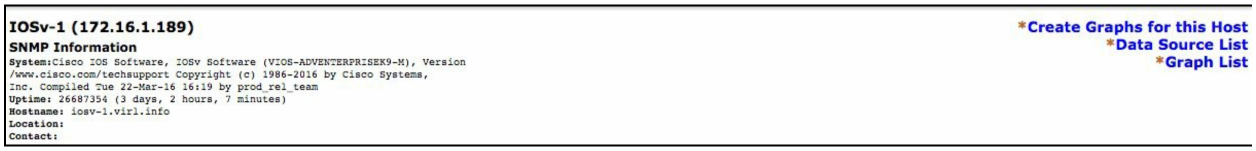
**SNMP Options**

| | |
|---|---|
| **SNMP Version** Choose the SNMP version for this device. | Version 2 |
| **SNMP Community** SNMP read community for this device. | secret |
| **SNMP Port** Enter the UDP port number to use for SNMP (default is 161). | 161 |
| **SNMP Timeout** The maximum number of milliseconds Cacti will wait for an SNMP response (does not work with php-snmp support). | 500 |
| **Maximum OID's Per Get Request** Specified the number of OID's that can be obtained in a single SNMP Get request. | 10 |

**Additional Options**

| | |
|---|---|
| **Notes** Enter notes to this host. | |

A sign indicating your SNMP is working would be that you to see able to see the device uptime:

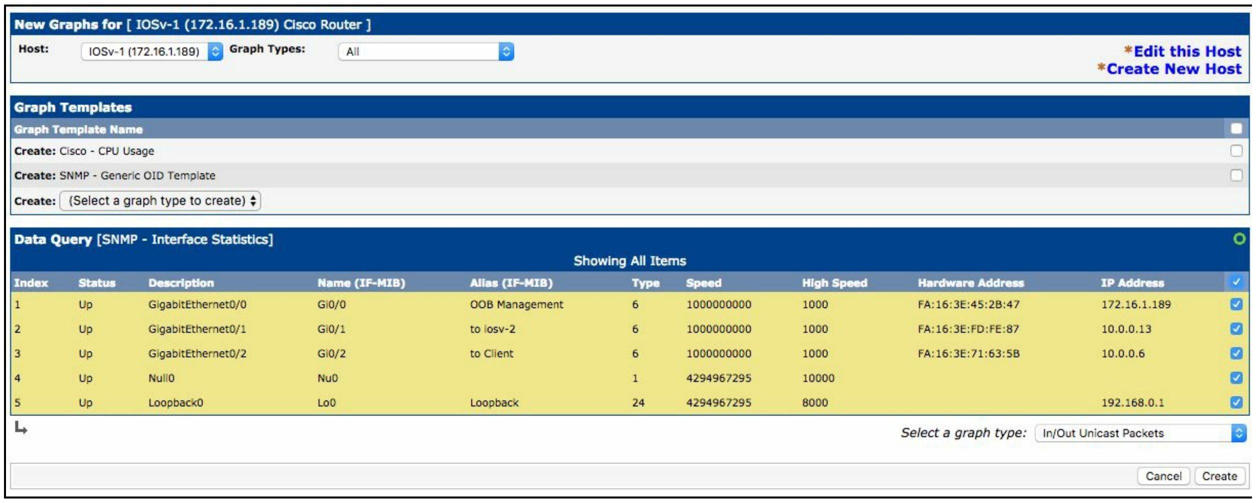


**IOSv-1 (172.16.1.189)**

**SNMP Information**

```
System:Cisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M), Version
/www.cisco.com/techsupport Copyright (c) 1986-2016 by Cisco Systems,
Inc. Compiled Tue 22-Mar-16 16:19 by prod_rel_team
Uptime: 26687354 (3 days, 2 hours, 7 minutes)
Hostname: iosv-1.virl.info
Location:
Contact:
```

You can add graphs to the device for interface traffic and other statistics:



**New Graphs for** [ IOSv-1 (172.16.1.189) Cisco Router ]

**Host:** IOSv-1 (172.16.1.189) **Graph Types:** All

**Graph Templates**

| Graph Template Name | |
|---|---|
| **Create:** Cisco - CPU Usage | ☐ |
| **Create:** SNMP - Generic OID Template | ☐ |
| **Create:** (Select a graph type to create) | |

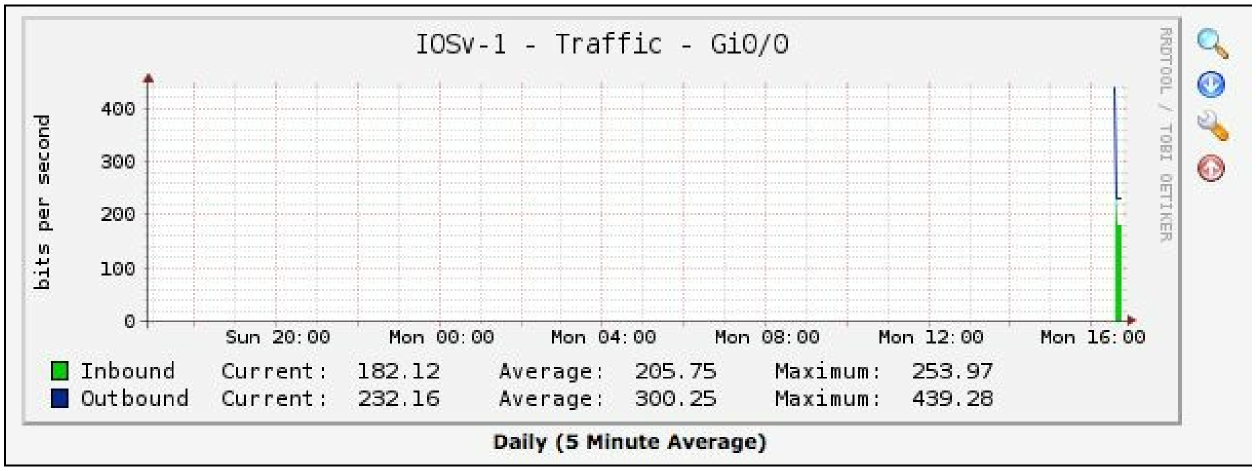
**Data Query** [SNMP - Interface Statistics]

Showing All Items

| Index | Status | Description | Name (IF-MIB) | Alias (IF-MIB) | Type | Speed | High Speed | Hardware Address | IP Address | ✓ |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Up | GigabitEthernet0/0 | Gi0/0 | OOB Management | 6 | 1000000000 | 1000 | FA:16:3E:45:2B:47 | 172.16.1.189 | ✓ |
| 2 | Up | GigabitEthernet0/1 | Gi0/1 | to iosv-2 | 6 | 1000000000 | 1000 | FA:16:3E:FD:FE:87 | 10.0.0.13 | ✓ |
| 3 | Up | GigabitEthernet0/2 | Gi0/2 | to Client | 6 | 1000000000 | 1000 | FA:16:3E:71:63:5B | 10.0.0.6 | ✓ |
| 4 | Up | Null0 | Nu0 | | 1 | 4294967295 | 10000 | | | ✓ |
| 5 | Up | Loopback0 | Lo0 | Loopback | 24 | 4294967295 | 8000 | | 192.168.0.1 | ✓ |

*Select a graph type:* In/Out Unicast Packets

After some time, you will start seeing traffic as shown here:



```
🟩 Inbound   Current:  182.12   Average:  205.75   Maximum:  253.97
🟦 Outbound  Current:  232.16   Average:  300.25   Maximum:  439.28
```

Daily (5 Minute Average)

We are now ready to look at how to use Python scripts to extend Cacti's data-gathering functionality.

# Python script as an input source

There are two documents that we should read before we use our Python script as an input source:

- Data input methods: http://www.cacti.net/downloads/docs/html/data_input_methods.html
- Making your scripts work with Cacti: http://www.cacti.net/downloads/docs/html/making_scripts_work_with_cacti.html

One might wonder what are the use cases for using Python script as an extension for data inputs? One of the use cases would be to provide monitoring to resources that do not have a corresponding OID. Say, we would like to know how many times the access list `permit_snmp` has allowed the host `172.16.1.173` for conducting an SNMP query. We know we can see the number of matches via the CLI:

```
iosv-1#sh ip access-lists permit_snmp | i 172.16.1.173
    10 permit 172.16.1.173 log (6362 matches)
```

However, chances are that there are no OIDs associated with this value (or we can pretend that there is none). This is where we can use an external script to produce an output that can be consumed by the Cacti host.

We can reuse the Pexpect script we discussed in Chapter 2, *Low-Level Network Device Interactions,* `chapter1_1.py`. We will rename it to `cacti_1.py`. Everything should be familiar to the original script, except that we will execute the CLI command and save the output:

```
for device in devices.keys():
...
    child.sendline('sh ip access-lists permit_snmp | i 172.16.1.173')
    child.expect(device_prompt)
    output = child.before
...
```

```
The output in its raw form will appear as below:
b'sh ip access-lists permit_snmp | i 172.16.1.173\r\n 10 permit 172.16.1.173 log
(6428 matches)\r\n'
```

We will use the `split()` function for the string to only leave the number of matches and print them out on standard output in the script:

```
print(str(output).split('(')[1].split()[0])
```

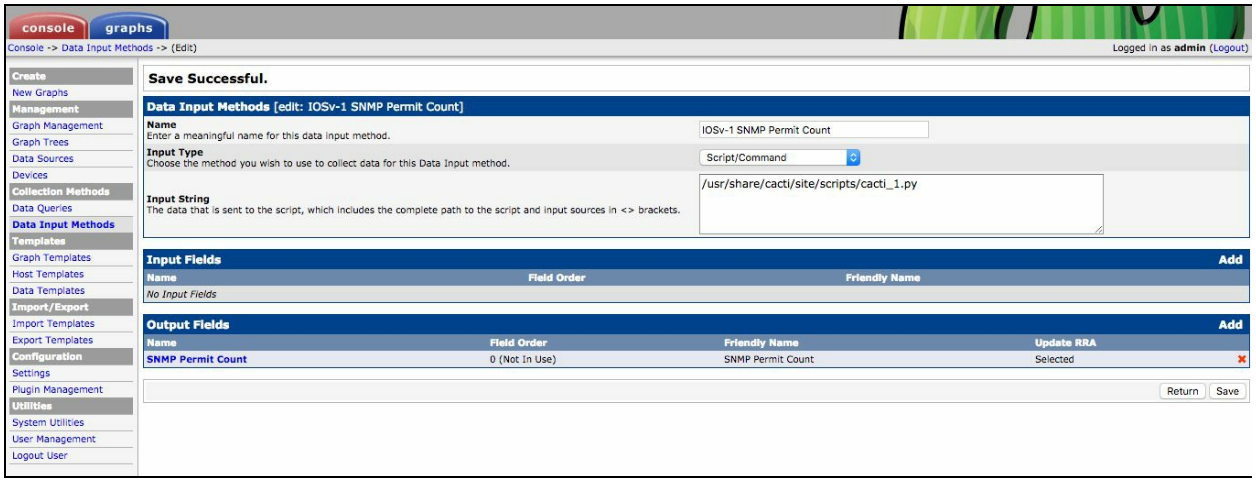
To test, we can see the number of increments by executing the script a number of times:

```
$ ./cacti_1.py
6428
$ ./cacti_1.py
6560
$ ./cacti_1.py
6758
```

We can make the script executable and put it into the default Cacti script location:

```
$ chmod a+x cacti_1.py
$ sudo cp cacti_1.py /usr/share/cacti/site/scripts/
```

Cacti documentation, available at http://www.cacti.net/downloads/docs/html/how_to.html, provides detailed steps on how to add the script result to the output graph. The steps include adding the script as a data input method, adding the input method to a data source, then creating a graph to be viewed:



Save Successful.

**Data Input Methods** [edit: IOSv-1 SNMP Permit Count]

| | |
|---|---|
| **Name**<br>Enter a meaningful name for this data input method. | IOSv-1 SNMP Permit Count |
| **Input Type**<br>Choose the method you wish to use to collect data for this Data Input method. | Script/Command |
| **Input String**<br>The data that is sent to the script, which includes the complete path to the script and input sources in <> brackets. | /usr/share/cacti/site/scripts/cacti_1.py |

**Input Fields** Add

| Name | Field Order | Friendly Name |
|---|---|---|
| *No Input Fields* | | |

**Output Fields** Add

| Name | Field Order | Friendly Name | Update RRA |
|---|---|---|---|
| SNMP Permit Count | 0 (Not In Use) | SNMP Permit Count | Selected |

SNMP is a common way to provide network monitoring services to the devices. RRDtool with Cacti as the frontend provides a good platform to be used for all the network devices via SNMP.

## Summary

In this chapter, we explored how to perform network monitoring via SNMP. We configured SNMP-related commands on the network devices and used our network management VM with SNMP poller to query the devices. We used the PySNMP module to simplify and automate our SNMP queries. You also learned how to save the query results in a flat file to be used for future examples.

Later in the chapter, we used two different Python visualization packages, namely Matplotlib and Pygal, to graph SNMP results. Each package has its distinct advantages. Matplotlib is mature, feature-rich, and widely used in data science projects. Pygal generates SVG format graphs that are flexible and web friendly.

Toward the end of the chapter, we looked at Cacti, an SNMP-based network monitoring system, and how to use Python to extend the platform's monitoring capabilities.

In the next chapter, we will continue to discuss the tools we can use to monitor our networks and gain insight into whether the network is behaving as expected. We will look at flow-based monitoring using NetFlow, sFlow, and IPFIX. We will also use tools such as Graphviz to visualize our network topology and detect any topological changes. Finally, we will use tools as well as Elasticsearch, Logstash, and Kibana, commonly referred to as the ELK stack, to monitor network log data as well as other network-related input.

# Network Monitoring with Python - Part 2

In the previous chapter, we used SNMP to query information from network devices. We did this using an SNMP manager to query the SNMP agent residing on the network device with specific tree-structured OID as the way to specify which value we intend to receive. Most of the time, the value we care about is a number, such as CPU load, memory usage, and interface traffic. It's something we can graph against time to give us a sense of how the value has changed over time.Â

We can typically classify the SNMP approach as a `pull` method, as if we are constantly asking the device for a particular answer. This particular method adds burden to the device because now it needs to spend a CPU cycle on the control plane to find answers from the subsystem and then accordingly answer the management station. Over time, if we have multiple SNMP pollers querying theÂ same device every 30 seconds (you would be surprised how often this happens), the management overhead would become substantial. In a human-based example analogous to SNMP, imagine that you have multiple people interrupting you every 30 seconds toÂ ask you a question. I know I would be annoyed even if it is a simple question (or worse if all of them are asking the same question).Â

Another way we can provide network monitoring is to reverse the relationship between the management station from a pull to a push. In other words, the information can be pushed from the device toward the management station in an already agreed upon format. This concept is what flow-based monitoring is based on. In a flow-based model, the network device streams the traffic information, called flow, to the management station. The format can be the Cisco proprietary NetFlow (version 5 or version 9), industry standard IPFIX, or the open source sFlow format. In this chapter, we will spend some time looking into NetFlow, IPFIX, and sFlow with Python.Â

Not all monitoring comes in the form of time-series data. Information such as network topology and syslog can be represented in a time-series format but most likely, this is not ideal. Network topology information checks whether the topology of the network has changed over time. We can use tools, such as Graphviz, with a Python wrapper to illustrate the topology. As already seen in Chapter 6, *Network Security with Python*, syslog contains security information.Â In this chapter, we will look at how to use the ELK stack (Elasticsearch, Logstash, Kibana) to collect network log information.

Specifically, we will cover the following topics:

- Graphviz, which is an open source graph visualization software that can help us quickly and efficiently graph our networkÂ
- Flow-based monitoring, such as NetFlow, IPFIX, and sFlowÂ
- UsingÂ ntop to visualize the flow informationÂ
- Elasticsearch for analyzing our collected dataÂ

Let's start by looking at how to use Graphviz as a tool to monitor network topology changes.Â

## Graphviz

Graphviz is an open source graph visualization software. Imagine you have to describe your network topology to a colleague without the benefit of a picture. You might say, "Our network consists of three layers: core, distribution, and access. The core layer comprises two routers for redundancy, and both the routers are fully meshed toward the four distribution routers; they are also fully meshed toward the access routers. The internal routing protocol is OSPF, and externally, we use BGP for our provider." While this description lacks some details, it is probably enough for your colleague to paint a pretty good high-level picture of your network.

Graphviz works similarly to the process of describing the graph in text term, then asking the Graphviz program to construct the graph for us. Here, the graph is described in a text format called DOT (https://en.wikipedia.org/wiki/DOT_(graph_description_language)) and Graphviz renders the graph based on the description. Of course, because the computer lacks human imagination, the language has to be precise and detailed.

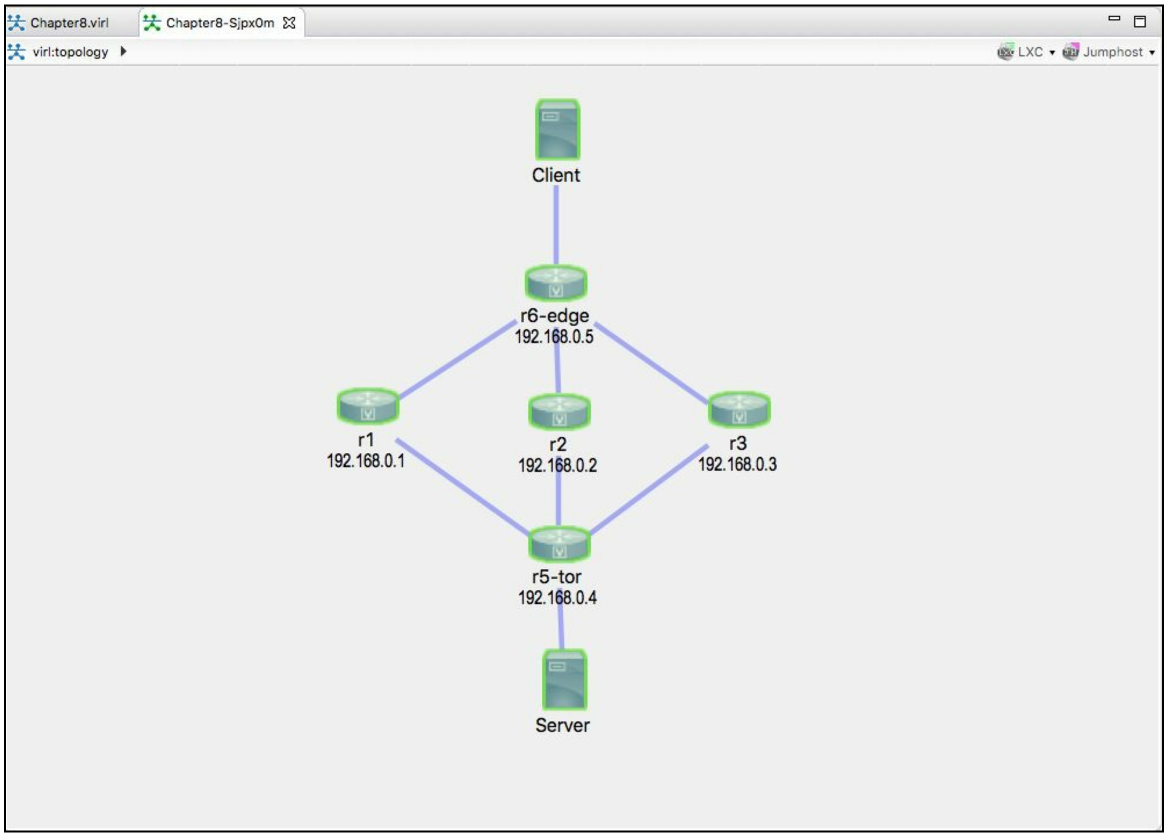
*For Graphviz-specific DOT grammar definitions, take a look at http://www.graphviz.org/doc/info/lang.html.*

In this section, we will use the **Link Layer Discovery Protocol** (**LLDP**) to query the device neighbors and create a network topology graph via Graphviz. Upon completing this extensive example, we will see how we can take something new, such as Graphviz, and combine it with things we have already learned to solve interesting problems.

Let's start by constructing the lab we will be using.

## Lab setup

In our lab, we will use VIRL, as in the previous chapters, because our goal is to illustrate a network topology. We will use five IOSv network nodes along with two server hosts:

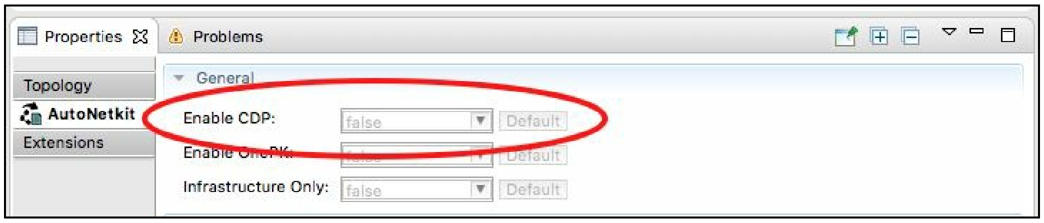


If you are wondering about my choice of IOSv as opposed to NX-OS or IOS-XR and the number of devices, here are a few points for you to consider when you build your own lab:

- Nodes virtualized by NX-OS and IOS-XR are much more memory intensive than IOS
- The VIRL virtual manager I am using has 8 GB of RAM, which seems enough to sustain nine nodes but could be a bit more unstable (nodes changing from reachable to unreachable at random)
- If you wish to use NX-OS, consider using NX-API or other API calls

that would return structured data

For our example, I am going to use LLDP as the protocol for link layer neighbor discovery because it is vendor-neutral. Note that VIRL provides an option to automatically enable CDP, which can save you some time and is similar to LLDP in functionality; however, it is also Cisco proprietary:



Enable CDP: false
Enable OnePK: false
Infrastructure Only: false

Once the lab is up and running, proceed to installing the necessary software packages.

## Installation

Graphviz can be obtained via apt:

```
$ sudo apt-get install graphviz
```

After the installation is complete, note that the way for verification is by using the `dot` command:

```
$ dot -V
dot - graphviz version 2.38.0 (20140413.2041)
```

We will use the Python wrapper for Graphviz, so let's install it now while we are at it:

```
$ sudo pip install graphviz
$ sudo pip3 install graphviz
```

Let's take a look at how to use the software.

# Graphviz examples

Like most popular open source projects, the documentation of Graphviz (http://www.graphviz.org/Documentation.php) is extensive. The challenge is often where to start. For our purpose, we will focus on *dot*, which draws directed graphs as hierarchies (not to be confused with the DOT language).

Let's start with the basic steps:

1. Nodes represent our network entities, such as routers, switches, and servers.
2. The edge represents the link between the network entities.
3. The graph, nodes, and edges each have attributes (http://www.graphviz.org/content/attrs) that can be tweaked.
4. After describing the network, output the network graph (http://www.graphviz.org/content/output-formats) in either the PNG, JPEG, or PDF format.

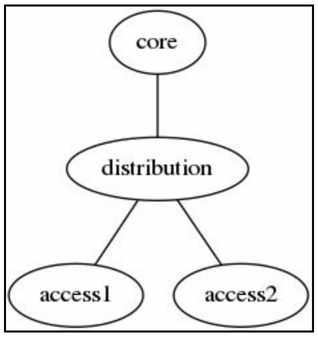
Our first example is an undirected dot graph consisting of four nodes (core, distribution, `access1`, and `access2`). The edges join the core node to the distribution node as well as distribution to both the access nodes:

```
$ cat chapter8_gv_1.gv
graph my_network {
    core -- distribution;
    distribution -- access1;
    distribution -- access2;
}
```

The graph can be output in the `dot -T<format> source -o <output file>` command line:

```
$ dot -Tpng chapter8_gv_1.gv -o output/chapter8_gv_1.png
```

The resulting graph can be viewed from the following output folder:



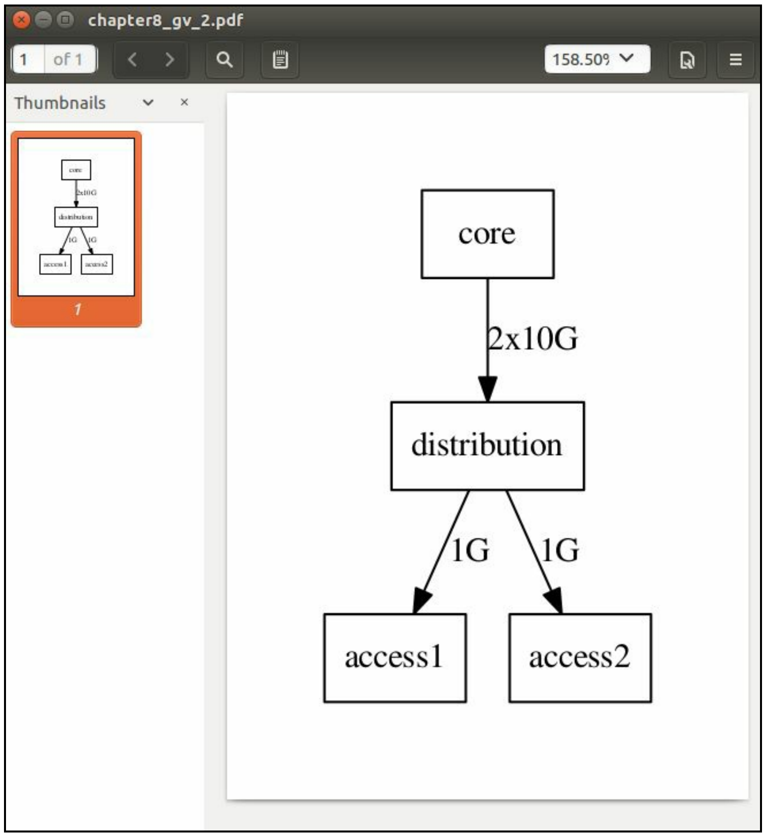
Note that we can use a directional graph by specifying it as a digraph instead of a graph as well as using the arrow (->) sign to represent the edges. There are several attributes we can modify in the case of nodes and edges, such as the node shape, edge labels, and so on. The same graph can be modified as follows:

```
$ cat chapter8_gv_2.gv
digraph my_network {
    node [shape=box];
    size = "50 30";
    core -> distribution [label="2x10G"];
    distribution -> access1 [label="1G"];
    distribution -> access2 [label="1G"];
}
```

We will output the file in PDF this time:

```
$ dot -Tpdf chapter8_gv_2.gv -o output/chapter8_gv_2.pdf
```

Take a look at the directional arrows in the new graph:



Now let's take a look at the Python wrapper around Graphviz.

# Python with Graphviz examples

We can reproduce the same topology graph as before using the Python Graphviz package that we have installed:

```
$ python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
>>> from graphviz import Digraph
>>> my_graph = Digraph(comment="My Network")
>>> my_graph.node("core")
>>> my_graph.node("distribution")
>>> my_graph.node("access1")
>>> my_graph.node("access2")
>>> my_graph.edge("core", "distribution")
>>> my_graph.edge("distribution", "access1")
>>> my_graph.edge("distribution", "access2")
```

The code basically produces what you would normally write in the DOT language. You can view the source:

```
>>> print(my_graph.source)
// My Network
digraph {
        core
        distribution
        access1
        access2
                core -> distribution
                distribution -> access1
                distribution -> access2
}
```

The graph can be rendered by the `.gv` file; by default, the format is PDF:

```
>>> my_graph.render("output/chapter8_gv_3.gv")
'output/chapter8_gv_3.gv.pdf'
```

The Python package wrapper closely mimics all the API options for Graphviz. You can find documentation on this on their website (http://graphviz.readthedocs.io/en/latest/index.html) You can also refer to the source code on GitHub for more information (https://github.com/xflr6/graphviz). We can now use the tools we have to map out our network.

# LLDP neighbor graphing

In this section, we will use the example of mapping out LLDP neighbors to illustrate a problem-solving pattern that has helped me over the years:

1. Modularize each task into smaller pieces, if possible. In our example, we can combine a few steps, but if we break them into smaller pieces, we will be able to reuse and improve them easily.
2. Use an automation tool to interact with the network devices, but keep the more complex logic aside at the management station. For example, the router has provided an LLDP neighbor output that is a bit messy. In this case, we will stick with the working command and the output and use a Python script at the management station to parse and receive the output we need.

3. When presented with choices for the same task, pick the one that can be reused. In our example, we can use low-level Pexpect, Paramiko, or Ansible playbooks to query the routers. In my opinion, it will be easier to reuse Ansible in future, so that is what I have picked.

To get started, since LLDP is not enabled on the routers by default, we will need to configure them on the devices first. By now, we know we have a number of options to choose from; in this case, I chose the Ansible playbook with the `ios_config` module for the task. The hosts file consists of five routers:

```
$ cat hosts
[devices]
r1 ansible_hostname=172.16.1.218
r2 ansible_hostname=172.16.1.219
r3 ansible_hostname=172.16.1.220
r5-tor ansible_hostname=172.16.1.221
r6-edge ansible_hostname=172.16.1.222
```

The `cisco_config_lldp.yml` playbook consists of one play with variables embedded in the playbook to configure the LLDP:

```
<skip>
 vars:
   cli:
     host: "{{ ansible_hostname }}"
     username: cisco
     password: cisco
     transport: cli tasks:
  - name: enable LLDP run
       ios_config:
         lines: lldp run
         provider: "{{ cli }}"
<skip>
```

After a few seconds, to allow LLDP exchange, we can verify that LLDP is indeed active on the routers:

```
r1#show lldp neighbors

Capability codes: (R) Router, (B) Bridge, (T) Telephone, (C) DOCSIS Cable Device
(W) WLAN Access Point, (P) Repeater, (S) Station, (O) Other

Device ID Local Intf Hold-time Capability Port ID
r2.virl.info Gi0/0 120 R Gi0/0
r3.virl.info Gi0/0 120 R Gi0/0
r5-tor.virl.info Gi0/0 120 R Gi0/0
r5-tor.virl.info Gi0/1 120 R Gi0/1
r6-edge.virl.info Gi0/2 120 R Gi0/1
r6-edge.virl.info Gi0/0 120 R Gi0/0

Total entries displayed: 6
```

In the output, you will see that `G0/0` is configured as the MGMT interface; therefore, you will see LLDP peers as if they are on a flat management network. What we really care about is the `G0/1` and `G0/2` interfaces connected to other peers. This knowledge will come in handy as we prepare to parse the output and construct our topology graph.

## Information retrieval

We can now use another Ansible playbook, namely `cisco_discover_lldp.yml`, to execute the LLDP command on the device and copy the output of each device to a `tmp` directory:

```
<skip>
 tasks:
   - name: Query for LLDP Neighbors
     ios_command:
       commands: show lldp neighbors
       provider: "{{ cli }}"
<skip>
```

The `./tmp` directory now consists of all the routers' output (showing LLDP neighbors) in its own file:

```
$ ls -l tmp/
total 20
-rw-rw-r-- 1 echou echou 630 Mar 13 17:12 r1_lldp_output.txt
-rw-rw-r-- 1 echou echou 630 Mar 13 17:12 r2_lldp_output.txt
-rw-rw-r-- 1 echou echou 701 Mar 12 12:28 r3_lldp_output.txt
-rw-rw-r-- 1 echou echou 772 Mar 12 12:28 r5-tor_lldp_output.txt
-rw-rw-r-- 1 echou echou 630 Mar 13 17:12 r6-edge_lldp_output.txt
```

The `r1_lldp_output.txt` content is the `output.stdout_lines` variable from our Ansible playbook:

```
$ cat tmp/r1_lldp_output.txt

[["Capability codes:", " (R) Router, (B) Bridge, (T) Telephone, (C) DOCSIS Cable
Device", " (W) WLAN Access Point, (P) Repeater, (S) Station, (O) Other", "",
"Device ID Local Intf Hold-time Capability Port ID", "r2.virl.info Gi0/0 120 R
Gi0/0", "r3.virl.info Gi0/0 120 R Gi0/0", "r5-tor.virl.info Gi0/0 120 R Gi0/0",
"r5-tor.virl.info Gi0/1 120 R Gi0/1", "r6-edge.virl.info Gi0/0 120 R Gi0/0", "",
"Total entries displayed: 5", ""]]
```

# Python parser script

We can now use a Python script to parse the LLDP neighbor output from each device and construct a network topology graph from it. The purpose is to automatically check the device to see whether any of the LLDP neighbors has disappeared due to link failure or other issues. Let's take a look at the `cisco_graph_lldp.py` file and see how that is done.

We start with the necessary imports of the packages--an empty list that we will populate with tuples of node relationships. We also know that `Gi0/0` on the devices are connected to the management network; therefore, we are only searching for `Gi0/[1234]` as our regular expression pattern in the show LLDP neighbors output:

```
import glob, re
from graphviz import Digraph, Source
pattern = re.compile('Gi0/[1234]')
device_lldp_neighbors = []
```

We will use the `glob.glob()` method to traverse through the `./tmp` directory of all the files, parse out the device name, and find the neighbors that the device is connected to. There are some embedded print statements in the script that were commented out; if you uncomment them, you will see that what we wanted to end up with was the parsed results:

```
device: r1
  neighbors: r5-tor
  neighbors: r6-edge
device: r5-tor
  neighbors: r2
  neighbors: r3
  neighbors: r1
device: r2
  neighbors: r5-tor
  neighbors: r6-edge
device: r3
  neighbors: r5-tor
  neighbors: r6-edge
device: r6-edge
  neighbors: r2
  neighbors: r3
  neighbors: r1
```

The fully populated edge list contains tuples that consist of the device and its neighbors:

```
Edges: [('r1', 'r5-tor'), ('r1', 'r6-edge'), ('r5-tor', 'r2'), ('r5-tor', 'r3'),
('r5-tor', 'r1'), ('r2', 'r5-tor'), ('r2', 'r6-edge'), ('r3', 'r5-tor'), ('r3',
'r6-edge'), ('r6-edge', 'r2'), ('r6-edge', 'r3'), ('r6-edge', 'r1')]
```

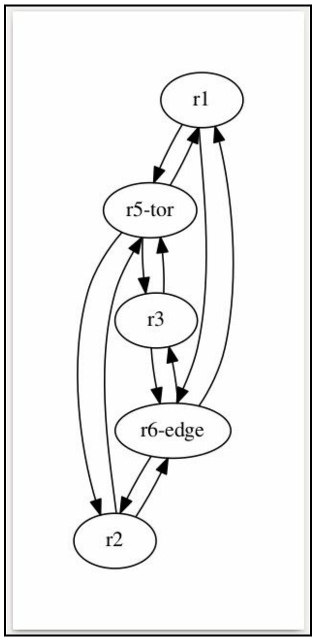
We can now construct the network topology graph using the Graphviz package. The most important part is the unpacking of the tuples that represent the edge relationship:

```
my_graph = Digraph("My_Network")
<skip>
# construct the edge relationships
for neighbors in device_lldp_neighbors:
    node1, node2 = neighbors
    my_graph.edge(node1, node2)
```

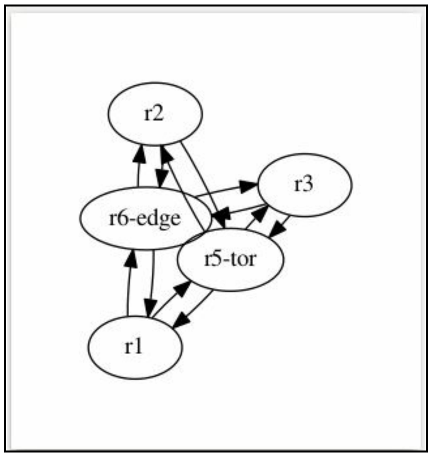
If you were to print out the resulting source dot file, it would be an accurate representation of your network:

```
digraph My_Network {
   r1 -> "r5-tor"
   r1 -> "r6-edge"
   "r5-tor" -> r2
   "r5-tor" -> r3
   "r5-tor" -> r1
   r2 -> "r5-tor"
   r2 -> "r6-edge"
   r3 -> "r5-tor"
   r3 -> "r6-edge"
   "r6-edge" -> r2
   "r6-edge" -> r3
   "r6-edge" -> r1
}
```

However, the placement of the nodes will be a bit funky, as it is autorendered. The following diagram illustrates the rendering in a default layout as well as the `neato` layout, namely digraph (`My_Network`, `engine='neato'`):



The `neato` layout represents an attempt to draw undirected graphs with even less hierarchy:



Sometimes the layout presented by the tool is just fine, especially if your goal is to detect faults as opposed to making it visually appealing. However, in this case, let's see how we can insert raw DOT language knobs into the source file. From research, we know that we can use the rank command to specify the level where some nodes can stay. However, there is no option presented in the Graphviz Python API. Luckily, the dot source file is just a string, which we can insert as raw dot comments using the `replace()` method with:

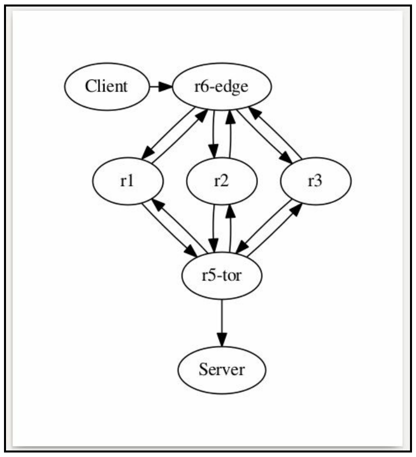
```
source = my_graph.source
original_text = "digraph My_Network {"
new_text = 'digraph My_Network {n{rank=same Client "r6-edge"}n{rank=same r1 r2
r3}n'
```

```
new_source = source.replace(original_text, new_text)
new_graph = Source(new_source)new_graph.render("output/chapter8_lldp_graph.gv")
```

The end result is a new source that we can render the final topology graph from:

```
digraph My_Network {
{rank=same Client "r6-edge"}
{rank=same r1 r2 r3}
                Client -> "r6-edge"
                "r5-tor" -> Server
                r1 -> "r5-tor"
                r1 -> "r6-edge"
                "r5-tor" -> r2
                "r5-tor" -> r3
                "r5-tor" -> r1
                r2 -> "r5-tor"
                r2 -> "r6-edge"
                r3 -> "r5-tor"
                r3 -> "r6-edge"
               "r6-edge" -> r2
               "r6-edge" -> r3
               "r6-edge" -> r1
}
```

The graph is now good to go:



## Final playbook

We are now ready to incorporate this new parser script back into our playbook. We can now add the additional task of rendering the output with graph generation in `cisco_discover_lldp.yml`:

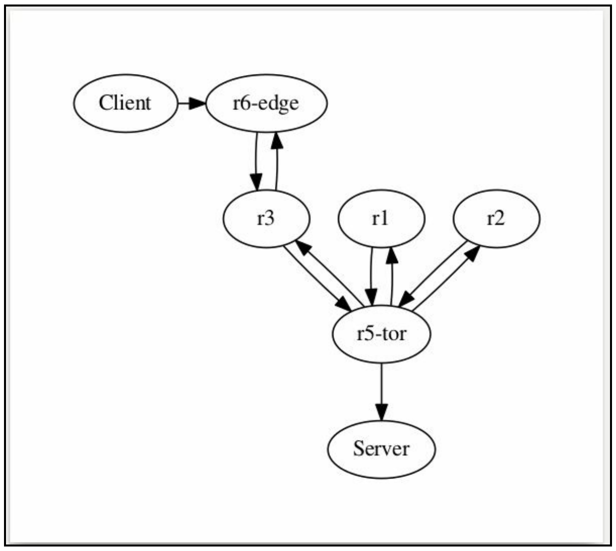
```
  - name: Execute Python script to render output
    command: ./cisco_graph_lldp.py
```

The playbook can now be scheduled to run regularly via cron or other means. It will automatically query the devices for LLDP neighbors and construct the graph, and the graph will represent the current topology as known by the routers.

We can test this by shutting down the `Gi0/1` and `Go0/2` interfaces on `r6-edge` when the LLDP neighbor passes the hold time and disappears from the LLDP table:

```
r6-edge#sh lldp neighbors
...
Device ID Local Intf Hold-time Capability Port ID
r2.virl.info Gi0/0 120 R Gi0/0
r3.virl.info Gi0/3 120 R Gi0/2
r3.virl.info Gi0/0 120 R Gi0/0
r5-tor.virl.info Gi0/0 120 R Gi0/0
r1.virl.info Gi0/0 120 R Gi0/0

Total entries displayed: 5
```

The graph will automatically show that `r6-edge` only connects to `r3`:



This is a relatively long example. We used the tools we have learned so far in the book--Ansible and Python--to modularize and break tasks into reusable pieces. We then used the new tool, namely Graphviz, to help monitor the network for non-time-series data, such as network topology relationships.

# Flow-based monitoring

As mentioned in the chapter introduction, besides polling technology, such as SNMP, we can also use a push strategy, which allows the device to push network information toward the management station. NetFlow and its closely associated cousins--IPFIX and sFlow--are examples of such information push from the direction of the network device toward the management station.

A flow, as defined by IETF (https://www.ietf.org/proceedings/39/slides/int/ip1394-background/tsld004.htm), is a sequence of packets moving from an application sending something to the application receiving it. If we refer back to the OSI model, a flow is what constitutes a single unit of communication between two applications. Each flow comprises a number of packets; some flows have more packets (such as a video stream), while some have few (such as an HTTP request). If you think about flows for a minute, you'll notice that routers and switches might care about packets and frames, but the application and user usually care more about flows.

Flow-based monitoring usually refers to NetFlow, IPFIX, and sFlow:

- **NetFlow**: NetFlow v5 is a technology where the network device caches flow entries and aggregate packets by matching the set of tuples (source interface, source IP/port, destination IP/port, and such). Here, once a flow is completed, the network device exports the flow characteristics, including total bytes and packet counts in the flow, to the management station.
- **IPFIX**: IPFIX is the proposed standard for structured streaming and is similar to NetFlow v9, also known as Flexible NetFlow. Essentially, it is a definable flow export, which allows the user to export nearly anything that the network device knows about. The flexibility often comes at the expense of simplicity, though; in other words, it is a lot more complex than the traditional NetFlow. Additional complexity makes it less than ideal for introductory learning. However, once you are familiar with NetFlow v5, you will be able to parse IPFIX as long as you match the

template definition.

- **sFlow**: sFlow actually has no notion of a flow or packet aggregation by itself. It performs two types of sampling of packets. It randomly samples one out of n packets/applications and has time-based sampling counters. It sends the information to the management station, and the station derives the network flow information by referring to the type of packet sample received along with the counters. As it doesn't perform any aggregation on the network device, you can argue that sFlow is more scalable than others.

The best way to learn about each one of these is to probably dive right into examples.

## NetFlow parsing with Python

We can use Python to parse the NetFlow datagram being transported on the wire. This gives us a way to look at the NetFlow packet in detail as well as troubleshoot any NetFlow issue when they are not working as expected.

First, let's generate some traffic between the client and server across the VIRL network. We can use the built-in HTTP server module from Python to quickly launch a simple HTTP server on the VIRL host acting as the server:

```
cisco@Server:~$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 ...
```

TIP *For Python 2, the module is named* `SimpleHTTPServer`*, for example,* `python2 -m SimpleHTTPServer`.

We can create a short loop to continuously send HTTP GET to the web server on the client:

```
sudo apt-get install python-pip python3-pip
sudo pip install requests
sudo pip3 install requests

$ cat http_get.py
import requests, time
while True:
    r = requests.get('http://10.0.0.5:8000')
    print(r.text)
    time.sleep(5)
```

The client should get a very plain HTML page:

```
cisco@Client:~$ python3 http_get.py
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>
<title>Directory listing for /</title>
<body>
...
</body>
</html>
```

We should also see the requests continuously coming in from the client every 5 seconds:

```
cisco@Server:~$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 ...
10.0.0.9 - - [15/Mar/2017 08:28:29] "GET / HTTP/1.1" 200 -
10.0.0.9 - - [15/Mar/2017 08:28:34] "GET / HTTP/1.1" 200 -
```

We can export NetFlow from any of the devices, but since `r6-edge` is the first hop for the client host, we will have this host export NetFlow to the management host at port `9995`.

*In this example, we use only one device for demonstration, therefore we manually configure it with the necessary commands. In the next section, when we enable NetFlow on all the devices, we will use the Ansible playbook.*

```
!
ip flow-export version 5
ip flow-export destination 172.16.1.173 9995 vrf Mgmt-intf
!
interface GigabitEthernet0/4
 description to Client
 ip address 10.0.0.10 255.255.255.252
 ip flow ingress
 ip flow egress
...
!
```

Next, let's take a look at the Python parser script.

# Python socket and struct

The script, `netFlow_v5_parser.py`, was modified from *Brian Rak's* blog post on http://blog.devicenull.org/2013/09/04/python-netflow-v5-parser.html; this was done mostly for Python 3 compatibility as well as additional NetFlow version 5 fields. The reason we choose v5 instead of v9 is because v9 is more complex as it introduces templates; therefore, it will provide a very difficult-to-grasp introduction to NetFlow. Since NetFlow version 9 is an extended format of the original NetFlow version 5, all the concepts we introduced in this section are applicable to it.

Because NetFlow packets are represented in bytes over the wire, we will use the struct module included in the standard library to convert bytes into native Python data types.

*You'll find more information about the two modules at https://docs.python.org/3.5/library/socket.html and https://docs.python.org/3.5/library/struct.html.*

We will start by using the socket module to bind and listen for the UDP datagram. With `socket.AF_INET`, we intend on listing for the IPv4 address; with `socket.SOCK_DGRAM`, we specify that we'll see the UDP datagram:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('0.0.0.0', 9995))
```

We will start a loop and retrieve information off the wire:

```
while True:
        buf, addr = sock.recvfrom(1500)
```

The following line is where we begin to deconstruct or unpack the packet. The first argument of `!HH` specifies the network's big-endian byte order with the exclamation sign (big-endian) as well as the format of the C type (`H = 2` byte unsigned short integer):

```
(version, count) = struct.unpack('!HH',buf[0:4])
```

If you do not remember the NetFlow version 5 header off the top of your head (that was a joke by the way), you can always refer to http://www.cisco.com/c/en/us/td/docs/net_mgmt/netflow_collection_engine/3-6/user/guide/format.html#wp1006108 for a quick refresher. The rest of the header can be parsed accordingly, depending on the byte location and data type:

```
 (sys_uptime, unix_secs, unix_nsecs, flow_sequence) = struct.unpack('!IIII',
buf[4:20])
 (engine_type, engine_id, sampling_interval) = struct.unpack('!BBH', buf[20:24])
```

The while loop that follows will fill the `nfdata` dictionary with the flow record that unpacks the source address and port, destination address and port, packet count, and byte count, and prints the information out on the screen:

```
for i in range(0, count):
    try:
        base = SIZE_OF_HEADER+(i*SIZE_OF_RECORD)
        data = struct.unpack('!IIIIHH',buf[base+16:base+36])
        input_int, output_int = struct.unpack('!HH', buf[base+12:base+16])
        nfdata[i] = {}
        nfdata[i]['saddr'] = inet_ntoa(buf[base+0:base+4])
        nfdata[i]['daddr'] = inet_ntoa(buf[base+4:base+8])
        nfdata[i]['pcount'] = data[0]
        nfdata[i]['bcount'] = data[1]
...
```

The output of the script allows you to visualize the header as well as the flow content at a glance:

```
Headers:
NetFlow Version: 5
Flow Count: 9
System Uptime: 290826756
Epoch Time in seconds: 1489636168
Epoch Time in nanoseconds: 401224368
Sequence counter of total flow: 77616
0 192.168.0.1:26828 -> 192.168.0.5:179 1 packts 40 bytes
1 10.0.0.9:52912 -> 10.0.0.5:8000 6 packts 487 bytes
2 10.0.0.9:52912 -> 10.0.0.5:8000 6 packts 487 bytes
3 10.0.0.5:8000 -> 10.0.0.9:52912 5 packts 973 bytes
4 10.0.0.5:8000 -> 10.0.0.9:52912 5 packts 973 bytes
5 10.0.0.9:52913 -> 10.0.0.5:8000 6 packts 487 bytes
6 10.0.0.9:52913 -> 10.0.0.5:8000 6 packts 487 bytes
7 10.0.0.5:8000 -> 10.0.0.9:52913 5 packts 973 bytes
8 10.0.0.5:8000 -> 10.0.0.9:52913 5 packts 973 bytes
```

Note that in NetFlow version 5, the size of the record is fixed at 48 bytes; therefore, the loop and script are relatively straightforward. However, in the case of NetFlow version 9 or IPFIX, after the header, there is a template

FlowSet (http://www.cisco.com/en/US/technologies/tk648/tk362/technologies_white_paper09186a00800a3db9.html) that specifies the field count, field type, and field length. This allows the collector to parse the data without knowing the data format in advance.

As you may have guessed, there are other tools that save us the problem of parsing NetFlow records one by one. Let's look at one such tool, called **ntop**, in the next section.

## ntop traffic monitoring

Just like SNMP in the previous chapter, we can use Python scripts and other tools to directly poll the device. However, there are tools such as Cacti, which is an all-in-one open source package, that includes data collection (poller), data storage (RRD), and a web frontend for visualization.

In the case of NetFlow, there are a number of open source and commercial NetFlow collectors you can choose from. If you do a quick search for "top N open source NetFlow analyzer," you will see a number of comparison studies for different tools. Each one of them has their own strong and weak points; which one to use is really a matter of preference, platform, and your appetite for customization. I would recommend choosing a tool that would support both v5 and v9, potentially sFlow as well. A secondary consideration would be if the tool is written in a language that you can understand; I imagine having Python extensibility would be a nice thing.

Two of the open source NetFlow tools that I like and have used before are NfSen (with NFDUMP as the backend collector) and ntop (or ntopng). Between the two of them, ntop is a better-known traffic analyzer; it runs on both Windows and Linux platforms and integrates well with Python. Therefore, let's use ntop as an example in this section.

The installation of our Ubuntu host is straightforward:

```
$ sudo apt-get install ntop
```

The installation process will prompt for the necessary interface for listening and the administrator password. By default, the ntop web interface listens on port `3000`, while the probe listens on UDP port `5556`. On the network device, we need to specify the location of the NetFlow exporter:

```
!
ip flow-export version 5
ip flow-export destination 172.16.1.173 5556 vrf Mgmt-intf
!
```

*By default, IOSv creates a VRF called `Mgmt-intf` and places `Gi0/0` under VRF.*

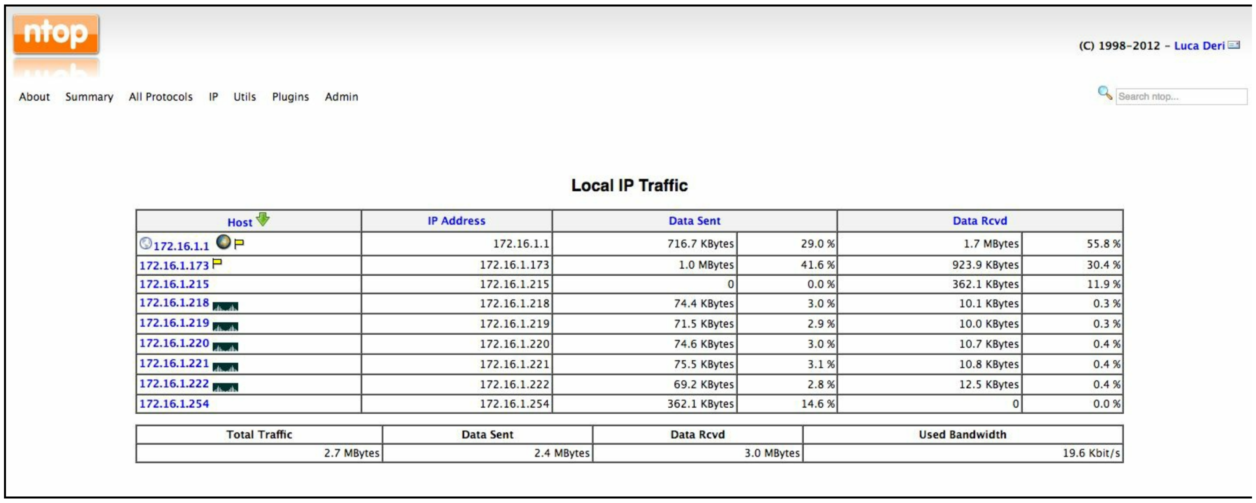
We will also need to specify which the direction of traffic exports, such as ingress or egress, under the interface configuration:

```
!
interface GigabitEthernet0/0
...
 ip flow ingress
 ip flow egress
...
```

For your reference, I have included the Ansible playbook, `cisco_config_netflow.yml`, to configure the lab device for the NetFlow export.

*Notice that `r5-tor` and `r6-edge` have two additional interfaces than `r1`, `r2`, and `r3`.*

Once everything is set up, you can check the ntop web interface for local IP traffic:

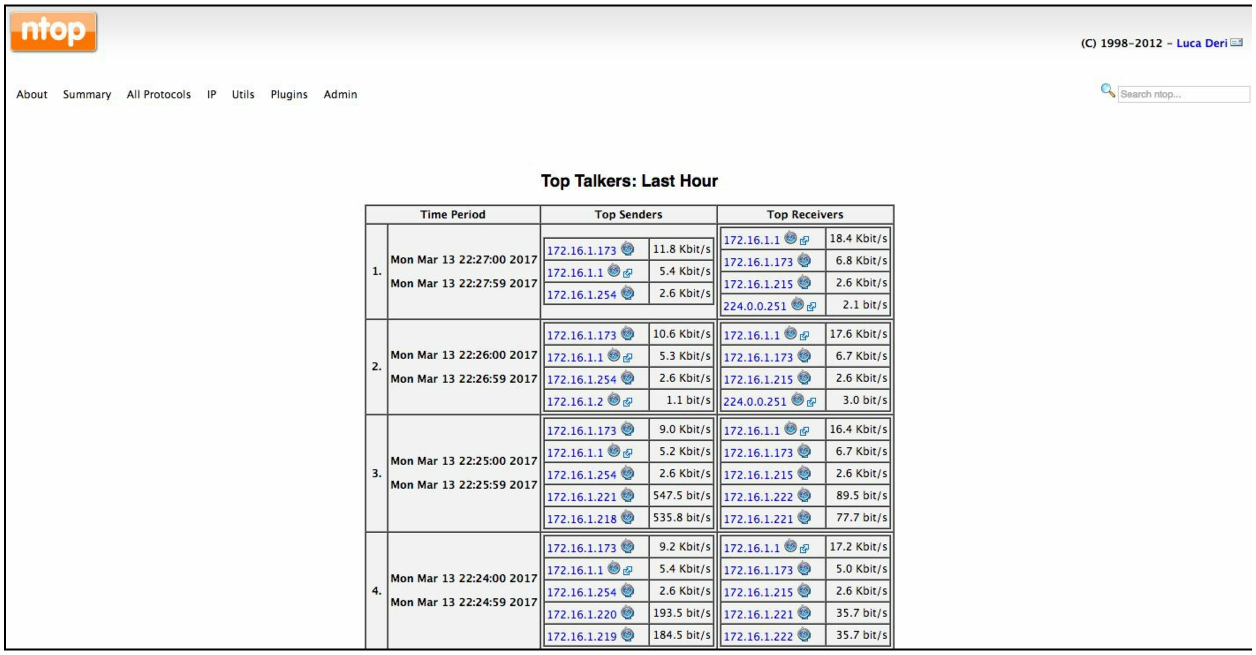


Local IP Traffic

| Host | IP Address | Data Sent | | Data Rcvd | |
|---|---|---|---|---|---|
| 172.16.1.1 | 172.16.1.1 | 716.7 KBytes | 29.0 % | 1.7 MBytes | 55.8 % |
| 172.16.1.173 | 172.16.1.173 | 1.0 MBytes | 41.6 % | 923.9 KBytes | 30.4 % |
| 172.16.1.215 | 172.16.1.215 | 0 | 0.0 % | 362.1 KBytes | 11.9 % |
| 172.16.1.218 | 172.16.1.218 | 74.4 KBytes | 3.0 % | 10.1 KBytes | 0.3 % |
| 172.16.1.219 | 172.16.1.219 | 71.5 KBytes | 2.9 % | 10.0 KBytes | 0.3 % |
| 172.16.1.220 | 172.16.1.220 | 74.6 KBytes | 3.0 % | 10.7 KBytes | 0.4 % |
| 172.16.1.221 | 172.16.1.221 | 75.5 KBytes | 3.1 % | 10.8 KBytes | 0.4 % |
| 172.16.1.222 | 172.16.1.222 | 69.2 KBytes | 2.8 % | 12.5 KBytes | 0.4 % |
| 172.16.1.254 | 172.16.1.254 | 362.1 KBytes | 14.6 % | 0 | 0.0 % |

| Total Traffic | Data Sent | Data Rcvd | Used Bandwidth |
|---|---|---|---|
| 2.7 MBytes | 2.4 MBytes | 3.0 MBytes | 19.6 Kbit/s |

One of the most often used features of ntop is using it to look at the top talker graph:



**Top Talkers: Last Hour**

| | Time Period | Top Senders | | Top Receivers | |
|---|---|---|---|---|---|
| 1. | Mon Mar 13 22:27:00 2017 Mon Mar 13 22:27:59 2017 | 172.16.1.173 | 11.8 Kbit/s | 172.16.1.1 | 18.4 Kbit/s |
| | | 172.16.1.1 | 5.4 Kbit/s | 172.16.1.173 | 6.8 Kbit/s |
| | | 172.16.1.254 | 2.6 Kbit/s | 172.16.1.215 | 2.6 Kbit/s |
| | | | | 224.0.0.251 | 2.1 bit/s |
| 2. | Mon Mar 13 22:26:00 2017 Mon Mar 13 22:26:59 2017 | 172.16.1.173 | 10.6 Kbit/s | 172.16.1.1 | 17.6 Kbit/s |
| | | 172.16.1.1 | 5.3 Kbit/s | 172.16.1.173 | 6.7 Kbit/s |
| | | 172.16.1.254 | 2.6 Kbit/s | 172.16.1.215 | 2.6 Kbit/s |
| | | 172.16.1.2 | 1.1 bit/s | 224.0.0.251 | 3.0 bit/s |
| 3. | Mon Mar 13 22:25:00 2017 Mon Mar 13 22:25:59 2017 | 172.16.1.173 | 9.0 Kbit/s | 172.16.1.1 | 16.4 Kbit/s |
| | | 172.16.1.1 | 5.2 Kbit/s | 172.16.1.173 | 6.7 Kbit/s |
| | | 172.16.1.254 | 2.6 Kbit/s | 172.16.1.215 | 2.6 Kbit/s |
| | | 172.16.1.221 | 547.5 bit/s | 172.16.1.222 | 89.5 bit/s |
| | | 172.16.1.218 | 535.8 bit/s | 172.16.1.221 | 77.7 bit/s |
| 4. | Mon Mar 13 22:24:00 2017 Mon Mar 13 22:24:59 2017 | 172.16.1.173 | 9.2 Kbit/s | 172.16.1.1 | 17.2 Kbit/s |
| | | 172.16.1.1 | 5.4 Kbit/s | 172.16.1.173 | 5.0 Kbit/s |
| | | 172.16.1.254 | 2.6 Kbit/s | 172.16.1.215 | 2.6 Kbit/s |
| | | 172.16.1.220 | 193.5 bit/s | 172.16.1.221 | 35.7 bit/s |
| | | 172.16.1.219 | 184.5 bit/s | 172.16.1.222 | 35.7 bit/s |

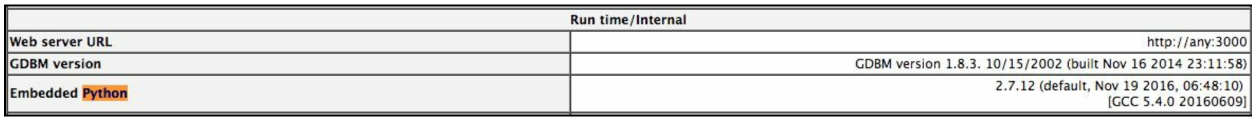
The ntop reporting engine is written in C; it is fast and efficient, but the need to have adequate knowledge of C in order to change the web frontend does not fit the modern agile development mindset. After a few false starts with Perl in the mid-2000s, the good folks at ntop finally settled on embedding Python as an extensible scripting engine. Let's take a look.

# Python extension for ntop

We can use Python to extend ntop through the ntop web server. The ntop web server can execute Python scripts. At a high level, the scripts will perform the following:

- Methods to access the state of ntop
- The Python CGI module to process forms and URL parameters
- Making templates that generate dynamic HTML pages
- Each Python script can read from `stdin` and print out `stdout/stderr`
- The `stdout` script is the returned HTTP page

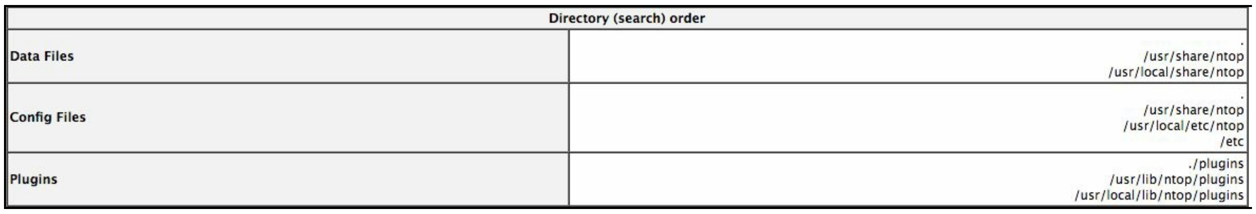
There are several resources that come in handy with the Python integration. Under the web interface, you can click on About | Show Configuration to see the Python interpreter version as well as the directory for your Python script:



| Run time/Internal | |
|---|---|
| Web server URL | http://any:3000 |
| GDBM version | GDBM version 1.8.3. 10/15/2002 (built Nov 16 2014 23:11:58) |
| Embedded Python | 2.7.12 (default, Nov 19 2016, 06:48:10) [GCC 5.4.0 20160609] |

Python Version

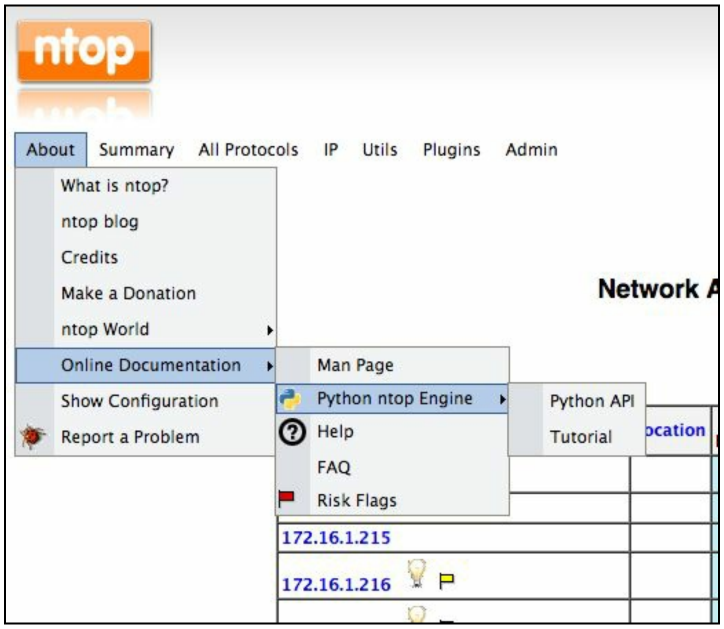
You can also check the various directories where the Python script should reside:



| Directory (search) order | |
|---|---|
| Data Files | . /usr/share/ntop /usr/local/share/ntop |
| Config Files | . /usr/share/ntop /usr/local/etc/ntop /etc |
| Plugins | ./plugins /usr/lib/ntop/plugins /usr/local/lib/ntop/plugins |

Plugin Directories

Under About | Online Documentation | Python ntop Engine, there are links for the Python API as well as the tutorial:



Python ntop Documentation

As mentioned, the ntop web server directly executes the Python script placed under the designated directory:

```
$ pwd
/usr/share/ntop/python
```

We will place our first script, namely `chapter8_ntop_1.py`, in the directory. The Python CGI module processes forms and parses URL parameters:

```
# Import modules for CGI handling
import cgi, cgitb
import ntop

# Parse URL
cgitb.enable();
```

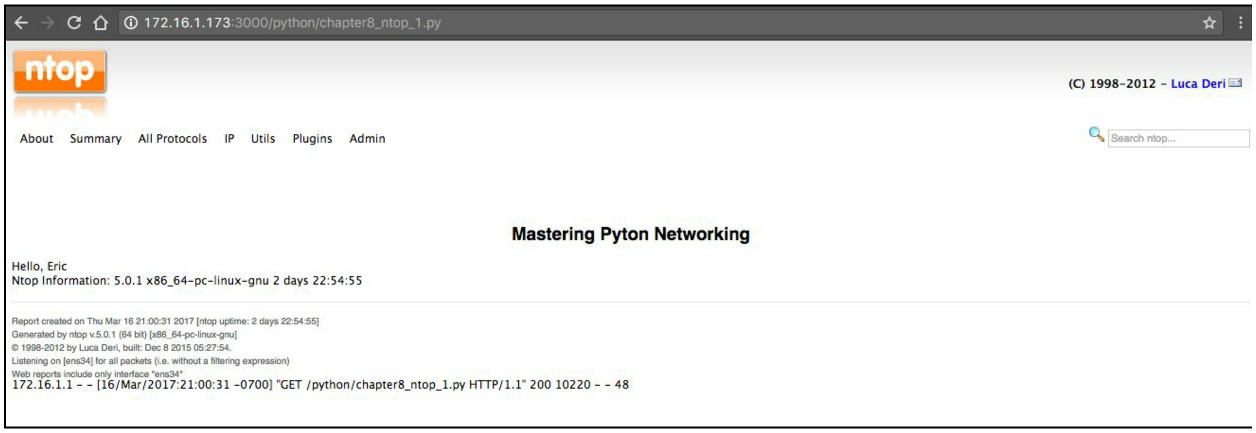
ntop implements three Python modules; each one of them has a specific purposes:

- **ntop**: This interacts with the ntop engine
- **Host**: This is used to drill down into specific hosts information
- **Interfaces**: This is the information on ntop instances

In our script, we will use the ntop module to retrieve the ntop engine information as well as use the `sendString()` method to send the HTML body text:

```
form = cgi.FieldStorage();
name = form.getvalue('Name', default="Eric")

version = ntop.version()
os = ntop.os()
uptime = ntop.uptime()

ntop.printHTMLHeader('Mastering Pyton Networking', 1, 0)
ntop.sendString("Hello, "+ name +"<br>")
ntop.sendString("Ntop Information: %s %s %s" % (version, os, uptime))
ntop.printHTMLFooter()
```

We will execute the Python script using `http://<ip>:3000/python/<script name>`. Here is the result of our `chapter8_ntop_1.py` script:


172.16.1.173:3000/python/chapter8_ntop_1.py
ntop
(C) 1998-2012 - Luca Deri
About
Summary
All Protocols
IP
Utils
Plugins
Admin
Search ntop...

Mastering Pyton Networking

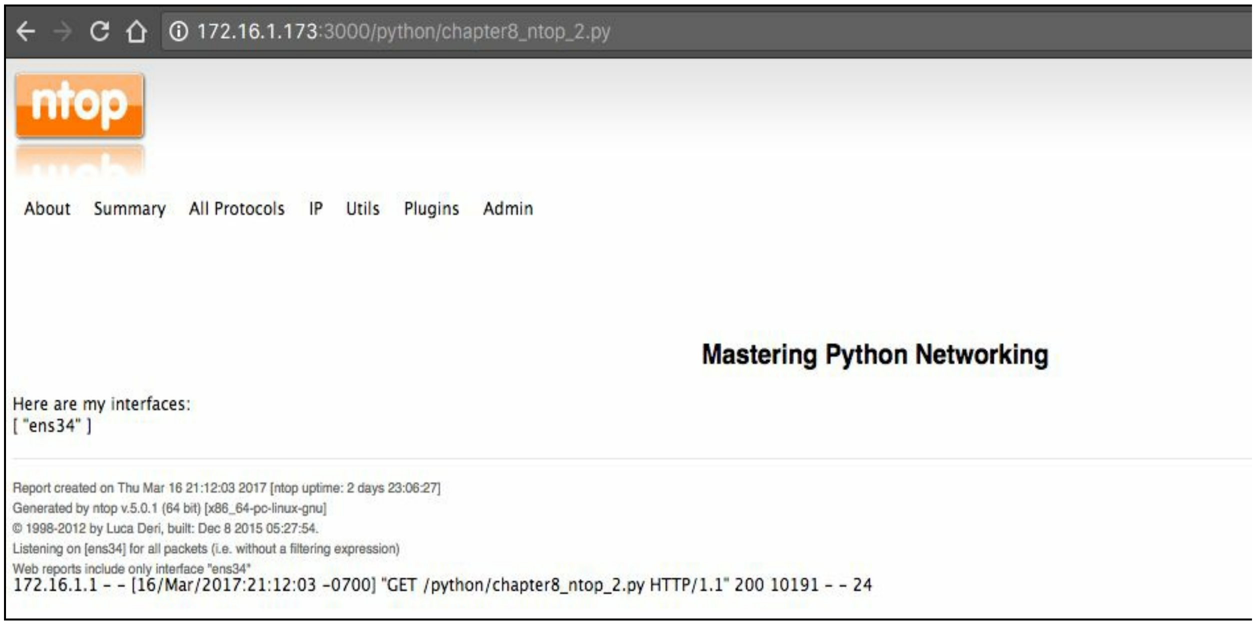
Hello, Eric
Ntop Information: 5.0.1 x86_64-pc-linux-gnu 2 days 22:54:55

```
Report created on Thu Mar 16 21:00:31 2017 [ntop uptime: 2 days 22:54:55]
Generated by ntop v.5.0.1 (64 bit) [x86_64-pc-linux-gnu]
© 1998-2012 by Luca Deri, built: Dec 8 2015 05:27:54.
Listening on [ens34] for all packets (i.e. without a filtering expression)
Web reports include only interface "ens34"
172.16.1.1 - - [16/Mar/2017:21:00:31 -0700] "GET /python/chapter8_ntop_1.py HTTP/1.1" 200 10220 - - 48
```

We can look at another example that interacts with the interface module `chapter8_ntop_2.py`. We will use the API to iterate through the interfaces:

```
import ntop, interface, json

ifnames = []
try:
    for i in range(interface.numInterfaces()):
        ifnames.append(interface.name(i))

except Exception as inst:
    print type(inst) # the exception instance
    print inst.args # arguments stored in .args
    print inst # __str__ allows args to printed directly
...
```

The resulting page will display the ntop interfaces:



Mastering Python Networking

Here are my interfaces:
[ "ens34" ]

Report created on Thu Mar 16 21:12:03 2017 [ntop uptime: 2 days 23:06:27]
Generated by ntop v.5.0.1 (64 bit) [x86_64-pc-linux-gnu]
© 1998-2012 by Luca Deri, built: Dec 8 2015 05:27:54.
Listening on [ens34] for all packets (i.e. without a filtering expression)
Web reports include only interface "ens34"

```
172.16.1.1 - - [16/Mar/2017:21:12:03 -0700] "GET /python/chapter8_ntop_2.py HTTP/1.1" 200 10191 - - 24
```

Besides the community version, they also offer commercial products if you need support. With the active open source community, commercial backing, and Python extensibility, ntop is a good choice for your NetFlow monitoring needs.

Next, let's take a look at NetFlow's cousin: sFlow.

## sFlow

sFlow, which stands for sampled flow, was originally developed by *InMon* (http://www.inmon.com) and later standardized by the way of RFC. The current version is v5. The primary advantage argued for sFlow is scalability. sFlow uses random one in `n` packets flow samples along with the polling interval of counter samples to derive an estimate of the traffic; this is less CPU-intensive than NetFlow. sFlow's statistical sampling is integrated with the hardware and provides real-time, raw export.

For scalability and competitive reasons, sFlow is generally preferred over NetFlow for newer vendors, such as Arista Networks, Vyatta, and A10 Networks. While Cisco supports sFlow on its Nexus 3000 platform, sFlow is generally *not* supported on Cisco platforms.

# SFlowtool and sFlow-RT with Python

Unfortunately, at this point, sFlow is something that our VIRL lab devices do not support. You can either use a Cisco Nexus 3000 switch or other vendor switches, such as Arista, that support sFlow. Another good option for the lab is to use an Arista vEOS virtual manager. I happen to have Cisco Nexus 3048 switches running 7.0 (3) that I will be using for this section.

The configuration of Cisco Nexus 3000 for sFlow is straightforward:

```
Nexus-2# sh run | i sflow
feature sflow
sflow max-sampled-size 256
sflow counter-poll-interval 10
sflow collector-ip 192.168.199.185 vrf management
sflow agent-ip 192.168.199.148
sflow data-source interface Ethernet1/48
```

The easiest way to utilize sFlow is to use sflowtool. Refer to http://blog.sflow.com/2011/12/sflowtool.html, which provides easy instructions for the installation:

```
$ wget http://www.inmon.com/bin/sflowtool-3.22.tar.gz
$ tar -xvzf sflowtool-3.22.tar.gz
$ cd sflowtool-3.22/
$ ./configure
$ make
$ sudo make install
```

After the installation, you can launch sFlow and look at the datagram Nexus 3048 is sending:

```
$ sflowtool
startDatagram =================================
datagramSourceIP 192.168.199.148
datagramSize 88
unixSecondsUTC 1489727283
datagramVersion 5
agentSubId 100
agent 192.168.199.148
packetSequenceNo 5250248
sysUpTime 4017060520
samplesInPacket 1
startSample ----------------------
```

```
sampleType_tag 0:4
sampleType COUNTERSSAMPLE
sampleSequenceNo 2503508
sourceId 2:1
counterBlock_tag 0:1001
5s_cpu 0.00
1m_cpu 21.00
5m_cpu 20.80
total_memory_bytes 3997478912
free_memory_bytes 1083838464
endSample ----------------------
endDatagram =================================
```

There are a number of good usage examples on the sflowtool GitHub repository (https://github.com/sflow/sflowtool); one of them is to use a script to receive the sflowtool input and parse the output. We can use a Python script for this purpose. In the `chapter8_sflowtool_1.py` example, we will use `sys.stdin.readline` to receive the input and regular expression search to only print out only the lines containing the word `agent` when we see the sFlow agent:

```
import sys, re
for line in iter(sys.stdin.readline, ''):
    if re.search('agent ', line):
        print(line.strip())
```

The output can be piped to the sflowtool:

```
$ sflowtool | python3 chapter8_sflowtool_1.py
agent 192.168.199.148
agent 192.168.199.148
```

There are a number of other useful output examples, such as `tcpdump`, output as NetFlow version 5 records, and a compact line-by-line output.

Ntop supports sFlow, which means you can directly export your sFlow destination to the ntop collector. If your collector is NetFlow, you can use the `-c` option for the output in the NetFlow version 5 format:

```
NetFlow output:
 -c hostname_or_IP - (netflow collector host)
 -d port - (netflow collector UDP port)
 -e - (netflow collector peer_as (default = origin_as))
 -s - (disable scaling of netflow output by sampling rate)
 -S - spoof source of netflow packets to input agent IP
```

Alternatively, you can also use InMon's sFlow-RT (http://www.sflow-rt.com/index.ph

p) as your sFlow analytics engine. What sets sFlow-RT apart from an operator perspective is its vast REST API that can be customized to support your use cases. You can also easily retrieve the metrics from the API. You can take a look at its extensive API reference on http://www.sflow-rt.com/reference.php.

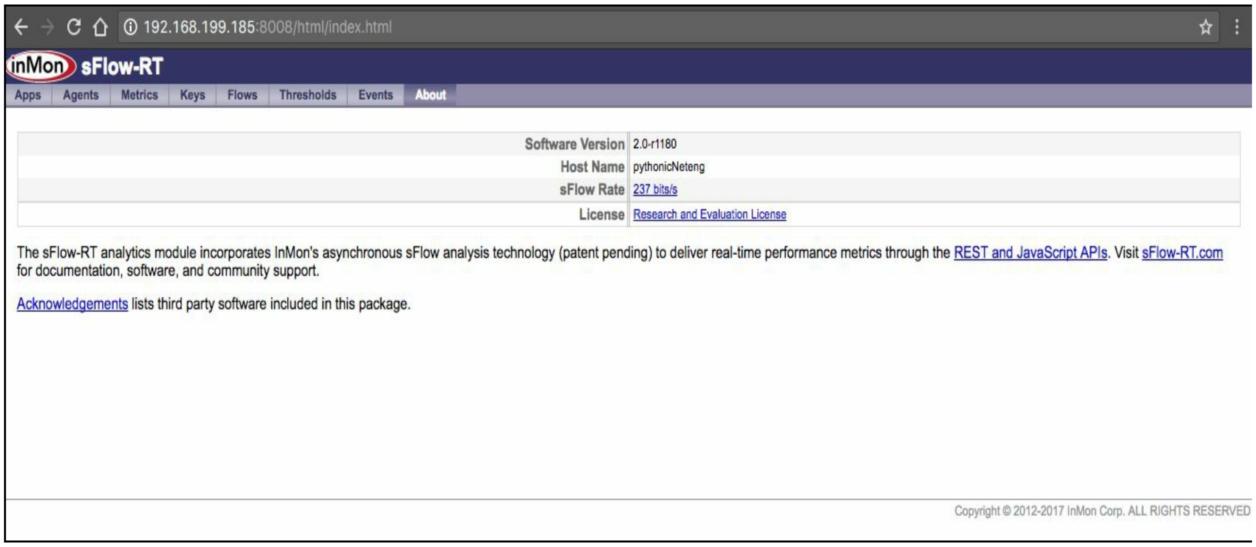
Note that sFlow-RT requires Java to run the following:

```
$ sudo apt-get install default-jre
$ java -version
openjdk version "1.8.0_121"
OpenJDK Runtime Environment (build 1.8.0_121-8u121-b13-0ubuntu1.16.04.2-b13)
OpenJDK 64-Bit Server VM (build 25.121-b13, mixed mode)
```

Once installed, downloading and running sFlow-RT is straightforward:

```
$ wget http://www.inmon.com/products/sFlow-RT/sflow-rt.tar.gz
$ tar -xvzf sflow-rt.tar.gz
$ cd sflow-rt/
$ ./start.sh
2017-03-17T09:35:01-0700 INFO: Listening, sFlow port 6343
2017-03-17T09:35:02-0700 INFO: Listening, HTTP port 8008
```

You can point the web browser to HTTP port `8008` and verify the installation:



|  |  |
|---|---|
| Software Version | 2.0-r1180 |
| Host Name | pythonicNeteng |
| sFlow Rate | 237 bits/s |
| License | Research and Evaluation License |

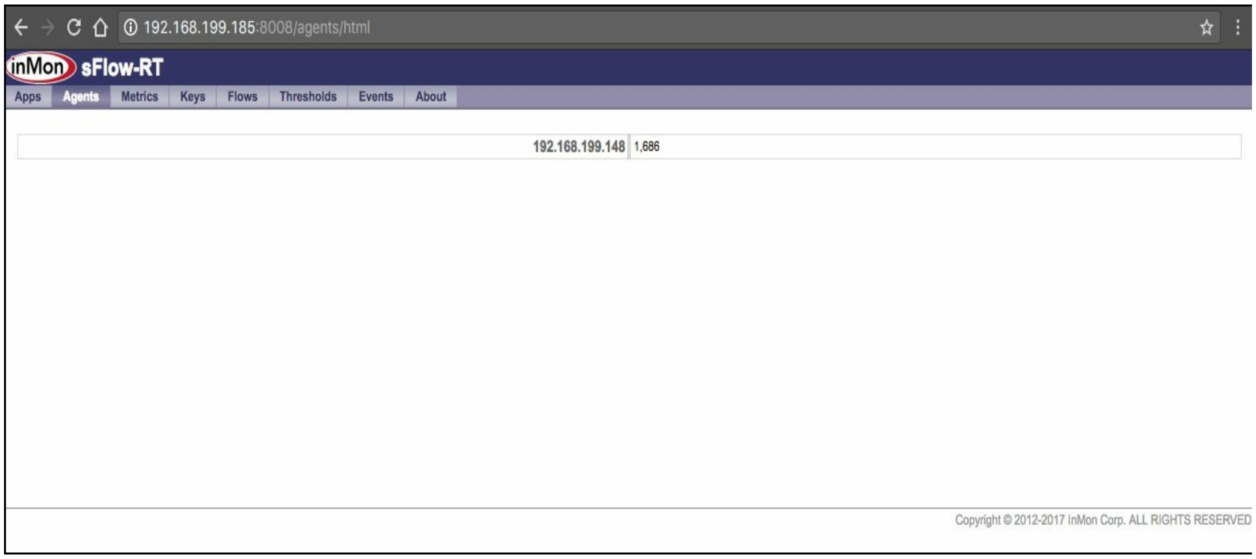
The sFlow-RT analytics module incorporates InMon's asynchronous sFlow analysis technology (patent pending) to deliver real-time performance metrics through the REST and JavaScript APIs. Visit sFlow-RT.com for documentation, software, and community support.

Acknowledgements lists third party software included in this package.

Copyright © 2012-2017 InMon Corp. ALL RIGHTS RESERVED

sFlow-RT About

As soon as sFlow receives the sFlow packets, the agents and other metrics will appear:


192.168.199.185:8008/agents/html
inMon sFlow-RT
Apps
Agents
Metrics
Keys
Flows
Thresholds
Events
About
Copyright © 2012-2017 InMon Corp. ALL RIGHTS RESERVED

| | |
|---|---|
| 192.168.199.148 | 1,686 |

sFlow-RT Agents

Here are two examples on using requests to retrieve information from sFlow-RT:

```
>>> import requests
>>> r = requests.get("http://192.168.199.185:8008/version")
>>> r.text
'2.0-r1180'
>>> r = requests.get("http://192.168.199.185:8008/agents/json")
>>> r.text
'{"192.168.199.148": {n "sFlowDatagramsLost": 0,n "sFlowDatagramSource":
["192.168.199.148"],n "firstSeen": 2195541,n "sFlowFlowDuplicateSamples": 0,n
"sFlowDatagramsReceived": 441,n "sFlowCounterDatasources": 2,n
"sFlowFlowOutOfOrderSamples": 0,n "sFlowFlowSamples": 0,n
"sFlowDatagramsOutOfOrder": 0,n "uptime": 4060470520,n
"sFlowCounterDuplicateSamples": 0,n "lastSeen": 3631,n "sFlowDatagramsDuplicates":
0,n "sFlowFlowDrops": 0,n "sFlowFlowLostSamples": 0,n "sFlowCounterSamples": 438,n
"sFlowCounterLostSamples": 0,n "sFlowFlowDatasources": 0,n
"sFlowCounterOutOfOrderSamples": 0n}}'
```

Consult the reference documentation for additional REST endpoints available for your needs. Next, we will take a look at another tool, namely Elasticsearch, that is becoming pretty popular for both syslog index and general network monitoring.

## Elasticsearch (ELK stack)

As we have seen so far in this chapter, use just the Python tools as we have done would adequately monitor your network with enough scalability for all types of networks, large and small alike. However, I would like to introduce one additional open source, general-purpose, distributed, search and analytics engine called **Elasticsearch** (https://www.elastic.co/). It is often referred to as the Elastic or ELK stack for combining with the frontend and input tools.

If you look at network monitoring in general, it is really about analyzing network data and making sense out of them. The ELK stack contains Elasticsearch, Logstash, and Kibina as a full stack to ingest information with Logstash, index and analyze data with Elasticsearch, and present the graphics output via Kibana. It is really three projects in one with the flexibility to substitute Logstash with another input, such as Beats. Alternatively, you can use other tools, such as Grafana, instead of Kibana for visualization. The ELK stack by *Elastic Co.* also provides many add-on tools,referred to as X-Pack, for additional security, alerting, monitoring, and such.

As you can probably tell by the description, ELK (or even Elasticsearch alone) is a deep topic to cover, and there are many books written on the subject. Even covering the basic usage would take up more space than we can spare in this book. I have at times considered leaving the subject out of the book simply for its depth. However, ELK has become a very important tool for many of the projects that I am working on, including network monitoring. I feel leaving it out would be a huge disservice to you.

Therefore, I am going to take a few pages to briefly introduce the tool and a few use cases along with information for you to dig deeper if needed. We will go through the following topics:

- Setting up a hosted ELK service
- The Logstash format
- Python's helper script for Logstash formatting

## Setting up a hosted ELK service

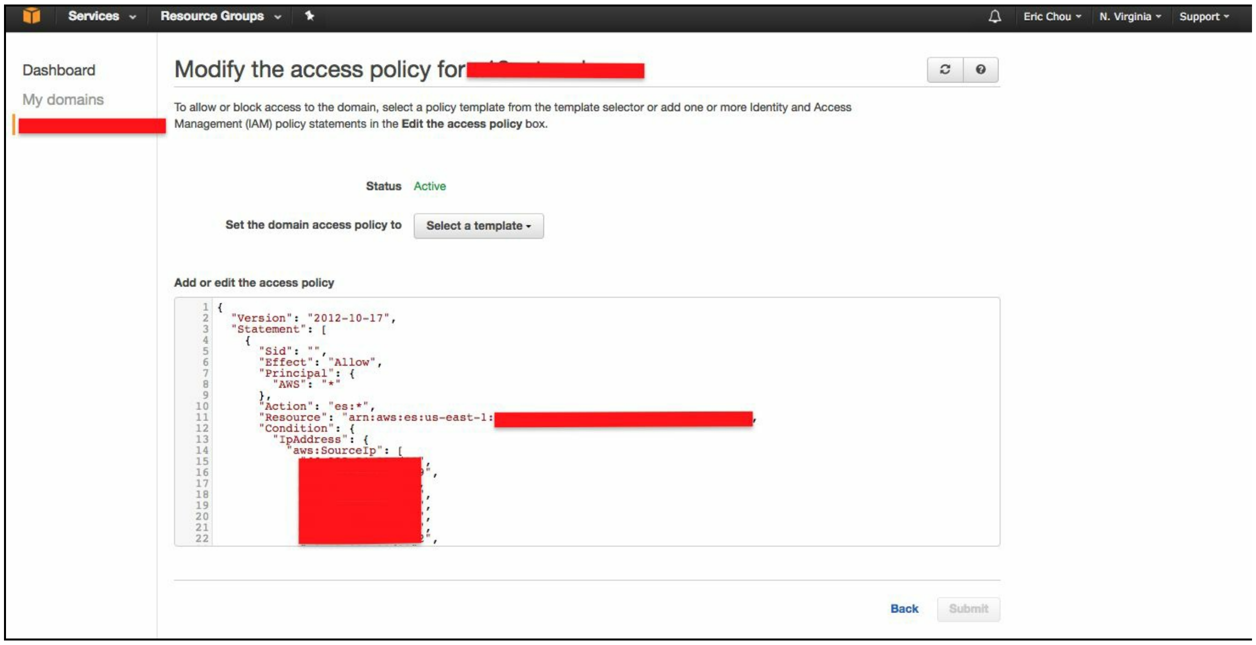
The entire ELK stack can be installed as a standalone server or distributed across multiple servers. The installation steps are available at https://www.elastic.co/guide/en/elastic-stack/current/installing-elastic-stack.html. In my experience, even with minimal amount of data, a single VM running ELK stack often stretches the resources. My first attempt at running ELK as a single VM lasted no more than a few days with barely two or three network devices sending log information toward it. After a few more unsuccessful attempts at running my own cluster as a beginner, I eventually settled on running the ELK stack as hosted service, and this is what I would recommend you start with.

As a hosted service, there are two providers that you can consider:

- **Amazon Elasticsearch Service** (https://aws.amazon.com/elasticsearch-service/)
- **Elastic Cloud** (https://cloud.elastic.co/)

Currently, AWS offers a free tier that is easy to get started with and is tightly integrated with the current suite of AWS set of tools, such as identity services (https://aws.amazon.com/iam/) and lambda functions (https://aws.amazon.com/lambda/). However, AWS's Elasticsearch Service does not have the latest features as compared to Elastic Cloud, nor does it have integration that is as tight as X-Pack. Because AWS offers a free tier, my recommendation would be that you start with the AWS Elasticsearch Service. If you find out later that you need more features than what AWS already provides, you can always move to Elastic Cloud.

Setting up the service is straightforward; you just need to choose your region and name your first domain. After setting it up, you can use the access policy to restrict input via an IP address; make sure this is the IP that AWS will see as the source IP (specify your corporate public IP if your host's IP address is translated behind NAT firewall):



Modify the access policy for [redacted]

To allow or block access to the domain, select a policy template from the template selector or add one or more Identity and Access Management (IAM) policy statements in the **Edit the access policy** box.

**Status** Active

**Set the domain access policy to** Select a template

**Add or edit the access policy**

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "AWS": "*"
      },
      "Action": "es:*",
      "Resource": "arn:aws:es:us-east-1:[redacted],
      "Condition": {
        "IpAddress": {
          "aws:SourceIp": [
            [redacted]
```

## The logstash format

Logstash can be installed where you are comfortable sending your network log to. The installation steps are available at https://www.elastic.co/guide/en/logstash/current/installing-logstash.html. By default, you can put the Logstash configuration file under `/etc/logstash/conf.d/`. The file is in the input-filter-output format (https://www.elastic.co/guide/en/logstash/current/advanced-pipeline.html). In the following example, we specified the input as a network log file, with a placeholder for filtering input, and the output as both printing out message to the console as well as having the output exported toward our AWS Elasticsearch Service instance:

```
input {
  file {
    type => "network_log"
    path => "path to your network log file"
 }
}
filter {
  if [type] == "network_log" {
  }
}
output {
  stdout { codec => rubydebug }
  elasticsearch {
  index => "logstash_network_log-%{+YYYY.MM.dd}"
  hosts => ["http://<instance>.<region>.es.amazonaws.com"]
  }
}
```

Now let's look at what more we can do with Python and Logstash.

## Python helper script for Logstash formatting

The preceding Logstash configuration will allow you to ingest network logs and create the index on Elasticsearch. What would happen if the text format we intend on putting into ELK is not a standard log format? This is where Python can help. In the next example, we will perform the following:

1. Use the Python script to retrieve a list of IPs that the Spamhaus project considers to be a drop list (https://www.spamhaus.org/drop/drop.txt).
2. Use the Python logging module to format the information in a way that Logstash can ingest it.
3. Modify the Logstash configuration file so any new input could be sent to the AWS Elasticsearch Service.

The `chapter8_logstash_1.py` script contains the code we will use. Besides the module imports, we will define the basic logging configuration:

```
#logging configuration
logging.basicConfig(filename='/home/echou/Master_Python_Networking/Chapter8/tmp/spa
 level=logging.INFO, format='%(asctime)s %(message)s', datefmt='%b %d %I:%M:%S')
```

We will define a few more variables and save the list of IP addresses from the requests in a variable:

```
host = 'python_networkign'
process = 'spamhause_drop_list'

r = requests.get('https://www.spamhaus.org/drop/drop.txt')
result = r.text.strip()

timeInUTC = datetime.datetime.utcnow().isoformat()
Item = OrderedDict()
Item["Time"] = timeInUTC
```

The final section of the script is a loop meant for parsing the output and writing it to the log:

```
for line in result.split('n'):
    if re.match('^;', line) or line == 'r': # comments
        next
    else:
       ip, record_number = line.split(";")
       logging.warning(host + ' ' + process + ': ' + 'src_ip=' + ip.split("/")[0] +
' record_number=' + record_number.strip())
```

Here's a sample of the log file entry:

```
...
Mar 17 01:06:08 python_networkign spamhause_drop_list: src_ip=5.8.37.0
record_number=SBL284078
Mar 17 01:06:08 python_networkign spamhause_drop_list: src_ip=5.34.242.0
record_number=SBL194796
...
```

We can then modify the Logstash configuration file accordingly, starting with adding the input file location:

```
input {
  file {
    type => "network_log"
    path => "path to your network log file"
 }
  file {
    type => "spamhaus_drop_list"
    path =>
"/home/echou/Master_Python_Networking/Chapter8/tmp/spamhaus_drop_list.log"
 }
}
```

We can add more filter configuration:

```
filter {
  if [type] == "spamhaus_drop_list" {
     grok {
       match => [ "message", "%{SYSLOGTIMESTAMP:timestamp} %{SYSLOGHOST:hostname} %
{NOTSPACE:process} src_ip=%{IP:src_ip} %{NOTSPACE:record_number}.*"]
       add_tag => ["spamhaus_drop_list"]
     }
  }
}
```

We can leave the output section unchanged, as the additional entries will be stored in the same index. We can now use the ELK stack to query, store, and view the network log as well as the Spamhaus IP information.

## Summary

In this chapter, we looked at additional ways in which we can utilize Python to enhance our network monitoring effort. We began by using Python's Graphviz package to create network topology graphs. This allows us to effortlessly show the current network topology as well as notice any link failures.

Next, we used Python to parse NetFlow version 5 packets to enhance our understanding and troubleshooting of NetFlow. We also looked at how to use ntop and Python to extend ntop for NetFlow monitoring. sFlow is an alternative packet sampling technology that we looked at where we use sflowtool and sFlow-RT to interpret the results. We ended the chapter with a general-purpose data analyzing tool, namely Elasticsearch or ELK stack.

In the next chapter, we will explore how to use the Python web framework Flask to build network web services.

# Building Network Web Services with Python

In the previous chapters, we were a consumer of the APIs provided by various tools. In chapter 3, *API and Intent-Driven Networking,* we saw we can use a `HTTP POST` method toÂ NX-API at the `http://<your router ip>/ins` URL with the `CLI` command embedded in the body to execute commands remotely on the Cisco Nexus device; the device then returns the command execution output in return. In chapter 8, *Network Monitoring with Python - Part 2,* we used the `GET` method for our sFlow-RT at `http://<your host ip>:8008/version`*Â* with an empty body to retrieve the version of the sFlow-RT software. These exchanges are examples of RESTful web services.Â

According to Wikipedia (https://en.wikipedia.org/wiki/Representational_state_transfer), *Representational state transfer (REST) or RESTful web services are one way of providing interoperability between computer systems on the Internet. REST-compliant web services allow requesting systems to access and manipulate textual representation of web resources using a uniform and predefined set of stateless operations.Â* As noted, REST web services using the HTTP protocol offerÂ the only method of information exchange on the web; other forms of web services just exist. But it is the most commonly used web service today, with the associated verbs `GET`, `POST`, `PUT`, and `DELETE` as a predefined way of information exchange.Â

One of the advantages of using RESTful services is the ability it provides for you to hide your internal operations from the user while still providing them with the service. In the case of the sFlow-RT example, if we were to log in to the device where our software is installed, we wouldn't really know where to check for the software version. By providing the resources in the form of a URL, the software abstracts the version-checking operations from the requester. The abstraction also provides a layer of security, as it can now open up the endpoints as needed.Â

As the master of theÂ network universe, RESTful web services provide many notableÂ benefits that we can enjoy, such as:Â

- You can abstract the requester from learning about the internals of the network operations. For example, we can provide a web service to query the switch version without the requester having to know the exact command.Â
- We can consolidate and customize operations that uniquely fit our network, such as a resource to upgrade all our top rack switches.Â
- We can provide better security by only exposing the operations as needed. For example, we can provide read-only URLs (`GET`) to core network devices and read-write URLs (`GET` / `POST` / `PUT` / `DELETE`) toÂ access switches.Â

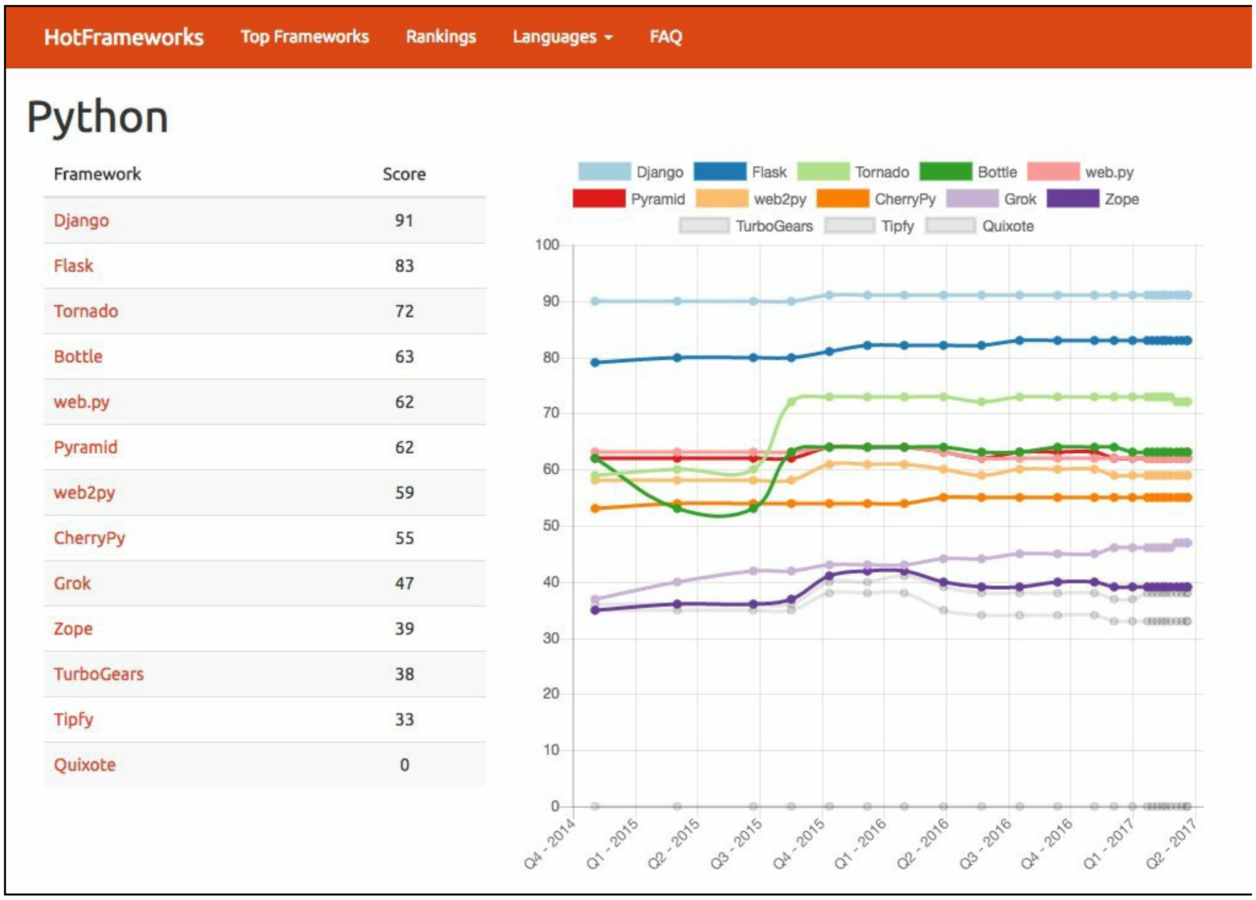
In this chapter, we will use one of the most popular Python web frameworks, namely Flask, to create our own REST web service for our network. In this chapter, we will learn about the following:Â

- Comparing Python web frameworksÂ
- Introduction to Flask
- Operations involving static network contents
- Operations involving dynamic network operations

Let's get started by looking at the available Python web frameworks and why we choose Flask.Â

# Comparing Python web frameworks

Python is known for its great web frameworks. There is a running joke at PyCon, which is that you can never work as a full-time Python developer without working on any of the Python web frameworks. There are annual conferences held for DjangoCon, one of the most popular Python frameworks. It attracts hundreds of attendees every year. If you sort the Python web frameworks on https://hotframeworks.com/languages/python, you can see there is no shortage of choices when it comes to Python and web frameworks.



Python

| Framework | Score |
|---|---|
| Django | 91 |
| Flask | 83 |
| Tornado | 72 |
| Bottle | 63 |
| web.py | 62 |
| Pyramid | 62 |
| web2py | 59 |
| CherryPy | 55 |
| Grok | 47 |
| Zope | 39 |
| TurboGears | 38 |
| Tipfy | 33 |
| Quixote | 0 |

Python Web Frameworks Ranking

With so many options to choose from, which one should we pick? Clearly, trying all the frameworks out yourself will be time consuming. The question about which web framework is better is also a passionate topic among web developers. If you ask this question on any of the forums, such as Quora, or search on Reddit, get ready for some highly opinionated answers and heated debates.

*Both Quora and Reddit were written in Python. Reddit uses Pylons (https://www.reddit.com/wiki/faq#wiki_so_what_python_framework_do_you_use.3F), while Quora started with Pylons then replaced some with their in-house code (https://www.quora.com/What-languages-and-frameworks-are-used-to-code-Quora).*

Of course, I have my own bias toward language and web frameworks. But in this section, I hope to convey to you my reasoning behind choosing one over the other. Let's pick the top two frameworks from the HotFrameworks list earlier and compare them:

- **Django**: The self-proclaimed web framework for perfectionists with deadlines is a high-level Python web framework that encourages rapid development and a clean, pragmatic design (https://www.djangoproject.com/). It is a large framework with prebuilt code that provides an administrative panel and built-in content management.
- **Flask**: This is a microframework for Python and is based on Werkzeug, Jinja2, and good intentions (http://flask.pocoo.org/). By micro, Flask intends on keeping the core small and the language easy to extend when needed; it certainly does not mean that Flask is lacking in functionality.

Personally, I find Django a bit difficult to extend, and most of the time, I only use a fraction of the prebuilt code. The idea of keeping the core code small and extending it when needed is very appealing to me. The initial example on the documentation to get Flask up and running consists of only eight lines of code and are easy to understand, even if you don't have any prior experience. Since Flask is built with extensions in mind, writing your own extensions, such as decorator, is pretty easy. Even though it is a microframework, the Flask core still includes the necessary components, such as a development server, debugger, integration with unit tests, RESTful request dispatching, and such, to get started out of the box. As you can see, besides Django, Flask is the second most popular Python framework by some measure. The popularity that comes with community contribution, support, and quick development helps it further.

For the preceding reasons, I feel Flask is an ideal choice for us when it comes to building network web services.

# Flask and lab setup

In this chapter, we will use virtualenv to isolate the environment we will work in. As the name indicates, virtualenv is a tool that creates a virtual environment. It can keep the dependencies required by different projects in separate places while keeping the global site-packages clean. In other words, when you install Flask in the virtual environment, it is only installed in the local virtualenv project directory, not the global site-packages.

The chances are you may have already come across virtualenv while working with Python before, so we will run through this process quickly. If you have not, feel free to pick up one of many excellent tutorials online, such as http://docs.python-guide.org/en/latest/dev/virtualenvs/. We will need to install virtualenv first:

```
# Python 3
$ sudo apt-get install python3-venv
$ python3 -m venv venv

# Python 2
$ sudo apt-get install python-virtualenv
$ virtualenv venv-python2
```

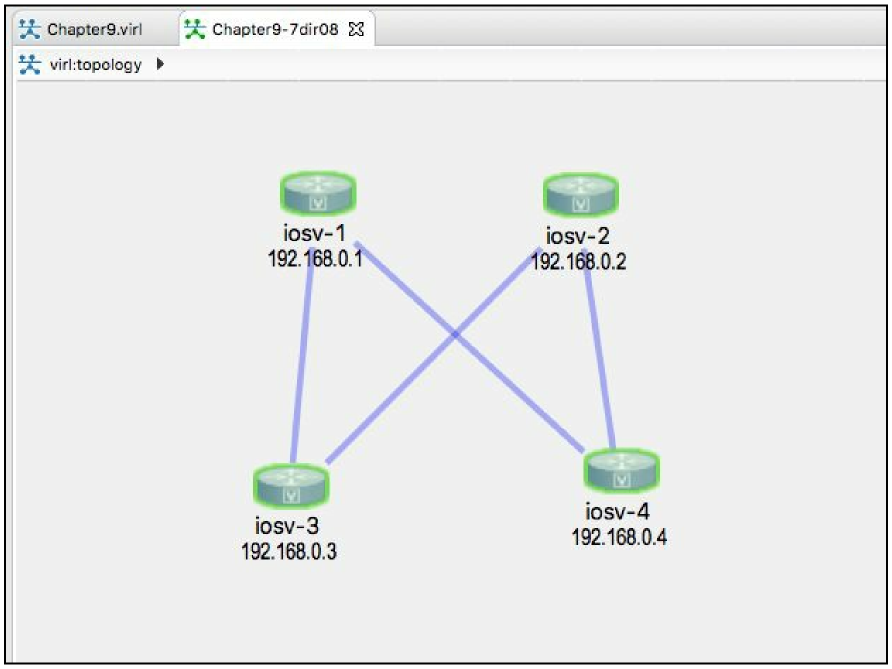
Then, activate and deactivate it in order to be in and out of the environment:

```
$ source venv/bin/activate
(venv) $ python
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
(venv) $ deactivate
```

In this chapter, we will install quite a few Python packages. To make life easier, I have included a `requirements.txt` file on the book's GitHub repository; we can use it to install all the necessary packages (remember to activate your virtualenv):

```
(venv) $ pip install -r requirements.txt
```

For our network topology, we will use a simple four-node network, as shown here:



Lab topology

Let's take a look at Flask in the next section.

*Please note that from here on out, I will assume that you would always execute from the virtual environment or you have installed the necessary packages in the* `requirements.txt` *file, which is in the global site-packages.*

# Introduction to Flask

Like most popular open source projects, Flask has very good documentation, available at http://flask.pocoo.org/docs/0.10/. If any of the examples are unclear, you can be sure to find the answer on the project documentation.

TIP *I would also highly recommend Miguel Grinberg's (https://blog.miguelgrinberg.com/) work related to Flask. His blog, book, and video training has taught me a lot about Flask. In fact, Miguel's class Building Web APIs with Flask inspired me to write this chapter. You can take a look at his published code on GitHub: https://github.com/miguelgrinberg/oreilly-flask-apis-video.*

Our first Flask application is contained in one single file, namely `chapter9_1.py`:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_networkers():
    return 'Hello Networkers!'

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

This will almost always be your design pattern for Flask initially. We create an instance of the Flask class with the first argument as the name of the application's module package. In this case, we used a single module; while doing this yourselves, type the name of your choice to indicate whether it is started as an application or imported as a module. We then use the route decorator to tell Flask which URL should be handled by the `hello_networkers()` function; in this case, we indicated the root path. We end the file with the usual name (https://docs.python.org/3.5/library/__main__.html). We only added the host and debug options, which allow more verbose output and also allow you to listen on all the interfaces of the host (by default, it only listens on loopback). We can run this application using the development server:

```
(venv) $ python chapter9_1.py
 * Running on http://0.0.0.0:5000/
 * Restarting with reloader
```

Now that we have a server running, let's test the server response with an HTTP client.

## The HTTPie client

We have already installed HTTPie (https://httpie.org/) as part of the `requirements.txt` step. Although it does not show in black and white text, HTTPie has better syntax highlighting. It also has more intuitive command-line interaction with the RESTful HTTP server. We can use it to test our first Flask application (more explanation on HTTPie to follow):

```
$ http GET http://172.16.1.173:5000/
HTTP/1.0 200 OK
Content-Length: 17
Content-Type: text/html; charset=utf-8
Date: Wed, 22 Mar 2017 17:37:12 GMT
Server: Werkzeug/0.9.6 Python/3.5.2

Hello Networkers!
```

*Alternatively, you can also use the -i switch with curl to see the HTTP headers:* `curl -i http://172.16.1.173:5000/`.

We will use HTTPie as our client for this chapter; it is worth a minute or two to take a look at its usage. We will use the HTTP-bin (https://httpbin.org/) service to show the use of HTTPie. The usage of HTTPie follows this simple pattern:

```
$ http [flags] [METHOD] URL [ITEM]
```

Following the preceding pattern, a `GET` request is very straightforward, as we have seen with our Flask development server:

```
$ http GET https://httpbin.org/user-agent
...
{
 "user-agent": "HTTPie/0.8.0"
}
```

JSON is the default implicit content type for HTTPie. If your body contains just strings, no other operation is needed. If you need to apply non-string JSON fields, use `:=` or other documented special characters:

```
$ http POST https://httpbin.org/post name=eric twitter=at_ericchou married:=true
HTTP/1.1 200 OK
...
Content-Type: application/json
...
{
    "headers": {
...
        "User-Agent": "HTTPie/0.8.0"
    },
    "json": {
        "married": true,
        "name": "eric",
        "twitter": "at_ericchou"
    },
    ...
    "url": "https://httpbin.org/post"
}
```

As you can see, HTTPie improves the traditional curl syntax and makes testing the REST API a breeze.

*More usage examples are available at https://httpie.org/doc#usage.*

Getting back to our Flask program, a large part of API building is based on the flow of URL routing. Let's take a deeper look at the `app.route()` decorator.

## URL routing

We added two additional functions and paired them up with the appropriate `app.route()` route in `chapter9_2.py`:

```
...
@app.route('/')
def index():
    return 'You are at index()'

@app.route('/routers/')
def routers():
    return 'You are at routers()'
...
```

The result is that different endpoints are passed to different functions:

```
$ http GET http://172.16.1.173:5000/
...

You are at index()

$ http GET http://172.16.1.173:5000/routers/
...

You are at routers()
```

Of course, the routing would be pretty limited if we have to keep it static all the time. There are ways to pass variables from the URL to Flask; we will look at an example in the next section.

# URL variables

As mentioned, we can also pass variables to the URL, as seen in the examples discussed in `chapter9_3.py`:

```
...
@app.route('/routers/<hostname>')
def router(hostname):
    return 'You are at %s' % hostname

@app.route('/routers/<hostname>/interface/<int:interface_number>')
def interface(hostname, interface_number):
    return 'You are at %s interface %d' % (hostname, interface_number)
...
```

Note that in the `/routers/<hostname>` URL, we pass the `<hostname>` variable as a string; `<int:interface_number>` will specify that the variable should only be an integer:

```
$ http GET http://172.16.1.173:5000/routers/host1
...
You are at host1

$ http GET http://172.16.1.173:5000/routers/host1/interface/1
...
You are at host1 interface 1

# Throws exception
$ http GET http://172.16.1.173:5000/routers/host1/interface/one
HTTP/1.0 404 NOT FOUND
...
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server. If you entered the URL manually
please check your spelling and try again.</p>
```

The converter includes integers, float, and path (accepts slashes).

Besides matching static routes, we can also generate URLs on the fly. This is very useful when we do not know the endpoint variable in advance or if the endpoint is based on other conditions, such as the values queried from a database. Let's take a look at an example of it.

## URL generation

In `chapter9_4.py`, we wanted to dynamically create a URL in the form of `'/<hostname>/list_interfaces'` in code:

```
from flask import Flask, url_for
...
@app.route('/<hostname>/list_interfaces')
def device(hostname):
    if hostname in routers:
        return 'Listing interfaces for %s' % hostname
    else:
        return 'Invalid hostname'

routers = ['r1', 'r2', 'r3']
for router in routers:
    with app.test_request_context():
        print(url_for('device', hostname=router))
...
```

Upon its execution, you will have a nice and logical URL created, as follows:

```
(venv) $ python chapter9_4.py
/r1/list_interfaces
/r2/list_interfaces
/r3/list_interfaces
```

For now, you can think of `app.text_request_context()` as a dummy request object that is necessary for demonstration purposes. If you are interested in the local context, feel free to take a look at http://flask.pocoo.org/docs/0.10/quickstart/#context-locals.

## The jsonify return

Another time saver in Flask is the `jsonify()` return, which wraps `json.dumps()` and turns the JSON output into a response object with `application/json` as the content type in the HTTP header. We can tweak the last script a bit as we will do in `chapter9_5.py`:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/routers/<hostname>/interface/<int:interface_number>')
def interface(hostname, interface_number):
    return jsonify(name=hostname, interface=interface_number)

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

We will see the result returned as a `JSON` object with the appropriate header:

```
$ http GET http://172.16.1.173:5000/routers/r1/interface/1
HTTP/1.0 200 OK
Content-Length: 36
Content-Type: application/json
...

{
     "interface": 1,
     "name": "r1"
}
```

Having looked at URL routing and the `jsonify()` return in Flask, we are now ready to build an API for our network.

## Network static content API

Often, your network consists of network devices that do not change a lot once put into production. For example, you would have core devices, distribution devices, spine, leaf, top of rack switches, and so on. Each of the devices would have certain characteristics and features that you would like to keep in a persistent location so you can easily retrieve them later on. This is often done in terms of storing data in a database. However, you would not normally want to give other users, who might want this information, direct access to the database; nor do they want to learn all the complex SQL query language. For this case, we can leverage Flask and the Flask-SQLAlchemy extension of Flask.

*You can learn more about Flask-SQLAlchemy at http://flask-sqlalchemy.pocoo.org/2.1/.*

# Flask-SQLAlchemy

Of course, SQLAlchemy and the Flask extension are a database abstraction layer and object relational mapper, respectively. It's a fancy way of saying *use the Python object for a database*. To make things simple, we will use SQLite as the database, which is a flat file that acts as a self-contained SQL database. We will look at the content of `chapter9_db_1.py` as an example of using Flask-SQLAlchemy to create a network database and insert a table entry into the database.

To begin with, we will create a Flask application and load the configuration for SQLAlchemy, such as the database path and name, then create the `SQLAlchemy` object by passing the application to it:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

# Create Flask application, load configuration, and create
# the SQLAlchemy object
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///network.db'
db = SQLAlchemy(app)
```

We can then create a database object and its associated primary key and various columns:

```
class Device(db.Model):
    __tablename__ = 'devices'
    id = db.Column(db.Integer, primary_key=True)
    hostname = db.Column(db.String(120), index=True)
    vendor = db.Column(db.String(40))

    def __init__(self, hostname, vendor):
        self.hostname = hostname
        self.vendor = vendor

    def __repr__(self):
        return '<Device %r>' % self.hostname
```

We can invoke the database object, create entries, and insert them into the database table. Keep in mind that anything we add to the session needs to be committed to the database in order to be permanent:

```
if __name__ == '__main__':
    db.create_all()
    r1 = Device('lax-dc1-core1', 'Juniper')
    r2 = Device('sfo-dc1-core1', 'Cisco')
    db.session.add(r1)
    db.session.add(r2)
    db.session.commit()
```

We can use the interactive prompt to check the database table entries:

```
>>> from flask import Flask
>>> from flask_sqlalchemy import SQLAlchemy
>>>
>>> app = Flask(__name__)
>>> app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///network.db'
>>> db = SQLAlchemy(app)
>>> from chapter9_db_1 import Device
>>> Device.query.all()
[<Device 'lax-dc1-core1'>, <Device 'sfo-dc1-core1'>]
>>> Device.query.filter_by(hostname='sfo-dc1-core1')
<flask_sqlalchemy.BaseQuery object at 0x7f1b4ae07eb8>
>>> Device.query.filter_by(hostname='sfo-dc1-core1').first()
<Device 'sfo-dc1-core1'>
```

We can also create new entries in the same manner:

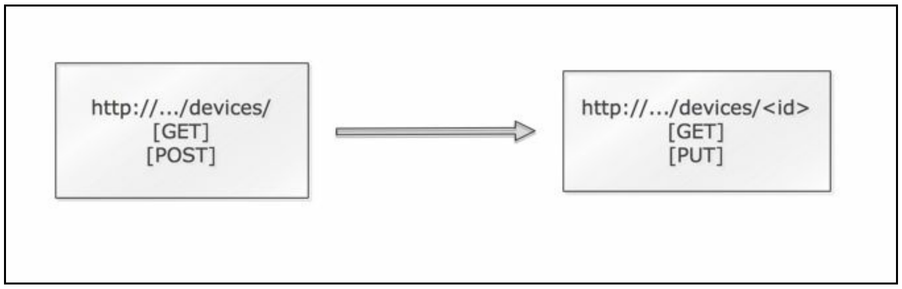
```
>>> r3 = Device('lax-dc1-core2', 'Juniper')
>>> db.session.add(r3)
>>> db.session.commit()
>>> Device.query.all()
[<Device 'lax-dc1-core1'>, <Device 'sfo-dc1-core1'>, <Device 'lax-dc1-core2'>]
```

# Network content API

Before we dive into code, let's take a moment to think about the API that we are trying to create. Planning for an API is usually more art than science; it really depends on your situation and preference. What I suggest next is, by no means, the right way, but for now, stay with me for the purpose of getting started.

Recall that in our diagram, we had four Cisco IOSv devices. Let's pretend that two of them, namely `iosv-1` and `iosv-2`, are of the network role of spine. The other two devices, `iosv-3` and `iosv-4`, are in our network service as leafs. These are obviously arbitrary choices and can be modified later on, but the point is that we want to keep some data about our network devices and expose them via an API.

To make things simple, we will create two APIs, namely devices group API and a single device API:



Network Content API

The first API will be our `http://172.16.1.173/devices/` endpoint that supports two methods: `GET` and `POST`. The `GET` request will return the current list of devices, while the `POST` request with the proper JSON body will create the device. Of course, you can choose to have different endpoints for creation and query, but in this design, we choose to differentiate the two by the HTTP method.

The second API will be specific to our device in the form of `http://172.16.1.173/devices/<device id>`. The API with the `GET` request will show the detail of the device that we have entered into the database. The `PUT` request will modify the entry with the update. Notice that we use `PUT` instead of `POST`. This is typical of HTTP API usage; when we need to modify an existing entry, we will use `PUT` instead of `POST`.

At this point, you would have a good idea about how your API would look. To better visualize the end result, I am going to jump ahead and see the end result quickly before we take a look at the code.

A `POST` request to the `/devices/` API will allow you to create an entry. In this case, I would like to create our network device with attributes such as hostname, loopback IP, management IP, role, vendor, and the operating system it runs on:

```
$ http POST http://172.16.1.173:5000/devices/ 'hostname'='iosv-1'
'loopback'='192.168.0.1' 'mgmt_ip'='172.16.1.225' 'role'='spine' 'vendor'='Cisco'
'os'='15.6'
HTTP/1.0 201 CREATED
Content-Length: 2
Content-Type: application/json
Date: Fri, 24 Mar 2017 01:45:15 GMT
Location: http://172.16.1.173:5000/devices/1
Server: Werkzeug/0.9.6 Python/3.5.2

{}
```

I can repeat the last step for the additional three devices:

```
$ http POST http://172.16.1.173:5000/devices/ 'hostname'='iosv-2'
'loopback'='192.168.0.2' 'mgmt_ip'='172.16.1.226' 'role'='spine' 'vendor'='Cisco'
'os'='15.6'
...
$ http POST http://172.16.1.173:5000/devices/ 'hostname'='iosv-3',
'loopback'='192.168.0.3' 'mgmt_ip'='172.16.1.227' 'role'='leaf' 'vendor'='Cisco'
'os'='15.6'
...
$ http POST http://172.16.1.173:5000/devices/ 'hostname'='iosv-4',
'loopback'='192.168.0.4' 'mgmt_ip'='172.16.1.228' 'role'='leaf' 'vendor'='Cisco'
'os'='15.6'
```

If we can use the same API with the `GET` request, we will be able to see the list of network devices that we created:

```
$ http GET http://172.16.1.173:5000/devices/
HTTP/1.0 200 OK
Content-Length: 188
Content-Type: application/json
Date: Fri, 24 Mar 2017 01:53:15 GMT
```

```
Server: Werkzeug/0.9.6 Python/3.5.2

{
    "device": [
        "http://172.16.1.173:5000/devices/1",
        "http://172.16.1.173:5000/devices/2",
        "http://172.16.1.173:5000/devices/3",
        "http://172.16.1.173:5000/devices/4"
    ]
}
```

Similarly, using the `GET` request for `/devices/<id>` will return specific information related to the device:

```
$ http GET http://172.16.1.173:5000/devices/1
HTTP/1.0 200 OK
Content-Length: 188
Content-Type: application/json
...
{
    "hostname": "iosv-1",
    "loopback": "192.168.0.1",
    "mgmt_ip": "172.16.1.225",
    "os": "15.6",
    "role": "spine",
    "self_url": "http://172.16.1.173:5000/devices/1",
    "vendor": "Cisco"
}
```

Let's pretend we have to downgrade the r1 operating system from 15.6 to 14.6; we can use the `PUT` request to update the device record:

```
$ http PUT http://172.16.1.173:5000/devices/1 'hostname'='iosv-1'
'loopback'='192.168.0.1' 'mgmt_ip'='172.16.1.225' 'role'='spine' 'vendor'='Cisco'
'os'='14.6'
HTTP/1.0 200 OK

# Verification
$ http GET http://172.16.1.173:5000/devices/1
...
{
    "hostname": "r1",
    "loopback": "192.168.0.1",
    "mgmt_ip": "172.16.1.225",
    "os": "14.6",
    "role": "spine",
    "self_url": "http://172.16.1.173:5000/devices/1",
    "vendor": "Cisco"
}
```

Now let's take a look at the code in `chapter9_6.py` that helped create the preceding APIs. What's cool, in my opinion, is that all of these APIs were done in a single file, including the database interaction. Later on, when we outgrow the APIs at hand, we can always separate the components out, such as having a separate file for the database class.

## Devices API

The `chapter9_6.py` file starts with the necessary imports. Note that the following request import is the request object from the client and not the requests package that we have been using in the previous chapters:

```
from flask import Flask, url_for, jsonify, request
from flask_sqlalchemy import SQLAlchemy
# The following is deprecated but still used in some examples
# from flask.ext.sqlalchemy import SQLAlchemy
```

We declared a database object with its ID as the primary key and string fields for hostname, loopback, management IP, role, vendor, and OS:

```
class Device(db.Model):
    __tablename__ = 'devices'
    id = db.Column(db.Integer, primary_key=True)
    hostname = db.Column(db.String(64), unique=True)
    loopback = db.Column(db.String(120), unique=True)
    mgmt_ip = db.Column(db.String(120), unique=True)
    role = db.Column(db.String(64))
    vendor = db.Column(db.String(64))
    os = db.Column(db.String(64))
```

The `get_url()` function returns a URL from the `url_for()` function. Note that the `get_device()` function called is not defined just yet under the `'/devices/<int:id>'` route:

```
def get_url(self):
    return url_for('get_device', id=self.id, _external=True)
```

The `export_data()` and `import_data()` functions are mirror images of each other. One is used to get the information from the database to the user (`export_data()`), such as when we use the `GET` method. The other is to put information from the user to the database (`import_data()`), such as when we use the `POST` or `PUT` method:

```
def export_data(self):
    return {
        'self_url': self.get_url(),
        'hostname': self.hostname,
        'loopback': self.loopback,
        'mgmt_ip': self.mgmt_ip,
        'role': self.role,
        'vendor': self.vendor,
        'os': self.os
    }

def import_data(self, data):
    try:
        self.hostname = data['hostname']
        self.loopback = data['loopback']
        self.mgmt_ip = data['mgmt_ip']
        self.role = data['role']
        self.vendor = data['vendor']
        self.os = data['os']
    except KeyError as e:
        raise ValidationError('Invalid device: missing ' + e.args[0])
    return self
```

With the database object in place as well as import and export functions created, the URL dispatch is straightforward for devices operations. The `GET` request will return a list of devices by querying all the entries in the devices table and also return the URL of each entry. The `POST` method will use the `import_data()` function with the global request object as the input. It will then add the device and commit to the database:

```
@app.route('/devices/', methods=['GET'])
def get_devices():
    return jsonify({'device': [device.get_url()
                              for device in Device.query.all()]})

@app.route('/devices/', methods=['POST'])
def new_device():
    device = Device()
    device.import_data(request.json)
    db.session.add(device)
    db.session.commit()
    return jsonify({}), 201, {'Location': device.get_url()}
```

If you look at the `POST` method, the returned body is an empty JSON body, with the status code `201` (created) as well as extra headers:

```
HTTP/1.0 201 CREATED
Content-Length: 2
Content-Type: application/json
Date: ...
Location: http://172.16.1.173:5000/devices/4
Server: Werkzeug/0.9.6 Python/3.5.2
```

Let's look at the API that queries and returns information pertaining to individual devices.

## The device ID API

The route for individual devices specifies that the ID should be an integer, which can act as our first line of defense against a bad request. The two endpoints follow the same design pattern as our `/devices/` endpoint, where we use the same `import` and `export` functions:

```
@app.route('/devices/<int:id>', methods=['GET'])
def get_device(id):
    return jsonify(Device.query.get_or_404(id).export_data())

@app.route('/devices/<int:id>', methods=['PUT'])
def edit_device(id):
    device = Device.query.get_or_404(id)
    device.import_data(request.json)
    db.session.add(device)
    db.session.commit()
    return jsonify({})
```

Notice the `query_or_404()` method; it provides a convenient way for returning `404 (not found)` if the database query returns negative for the ID passed in. This is a pretty elegant way of providing a quick check on the database query.

Finally, the last part of the code creates the database table and starts the Flask development server:

```
if __name__ == '__main__':
    db.create_all()
    app.run(host='0.0.0.0', debug=True)
```

This is one of the longer Python scripts in the book; therefore, we will take more time to explain it in detail. This provides a way to illustrate ways we can utilize the database in the backend to keep track of the network devices and only expose what we need to the external world as APIs, using Flask.

In the next section, we will take a look at how to use API to perform asynchronous tasks to either individual devices or a group of devices.

## Network dynamic operations

Our API can now provide static information about the network; anything that we can store in the database can be returned to the user. It would be great if we can interact with our network directly, such as query the device for information or push configuration changes to the device.

We will start this process by leveraging the script we have already seen in Chapter 2, *Low-Level Network Device Interactions* for interacting with a device via Pexpect. We will modify the script slightly into a function we can repeatedly use in `chapter9_pexpect_1.py`:

```
# We need to install pexpect for our virtual env
$ pip install pexpect

$ cat chapter9_pexpect_1.py
import pexpect

def show_version(device, prompt, ip, username, password):
    device_prompt = prompt
    child = pexpect.spawn('telnet ' + ip)
    child.expect('Username:')
    child.sendline(username)
    child.expect('Password:')
    child.sendline(password)
    child.expect(device_prompt)
    child.sendline('show version | i V')
    child.expect(device_prompt)
    result = child.before
    child.sendline('exit')
    return device, result
```

We can test the new function via the interactive prompt:

```
>>> from chapter9_pexpect_1 import show_version
>>> print(show_version('iosv-1', 'iosv-1#', '172.16.1.225', 'cisco', 'cisco'))
('iosv-1', b'show version | i V\r\nCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE (fc2)\r\n')
>>>
```

*Make sure your Pexpect script works before you proceed.*

We can add a new API for querying the device version in `chapter9_7.py`:

```
from chapter9_pexpect_1 import show_version
...
@app.route('/devices/<int:id>/version', methods=['GET'])
def get_device_version(id):
    device = Device.query.get_or_404(id)
    hostname = device.hostname
    ip = device.mgmt_ip
    prompt = hostname+"#"
    result = show_version(hostname, prompt, ip, 'cisco', 'cisco')
    return jsonify({"version": str(result)})
```

The result will be returned to the requester:

```
$ http GET http://172.16.1.173:5000/devices/4/version
HTTP/1.0 200 OK
Content-Length: 210
Content-Type: application/json
Date: Fri, 24 Mar 2017 17:05:13 GMT
Server: Werkzeug/0.9.6 Python/3.5.2

{
 "version": "('iosv-4', b'show version | i V\\r\\nCisco IOS Software, IOSv Software
(VIOS-ADVENTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE (fc2)\\r\\nProcessor
board ID 9U96V39A4Z12PCG4O6Y0Q\\r\\n')"
}
```

We can also add another endpoint that would allow us to perform bulk action on multiple devices, based on their common fields. In the following example, the endpoint will take the `device_role` attribute in the URL and match it up with the appropriate device(s):

```
@app.route('/devices/<device_role>/version', methods=['GET'])
def get_role_version(device_role):
    device_id_list = [device.id for device in Device.query.all() if device.role ==
device_role]
    result = {}
    for id in device_id_list:
        device = Device.query.get_or_404(id)
        hostname = device.hostname
        ip = device.mgmt_ip
        prompt = hostname + "#"
        device_result = show_version(hostname, prompt, ip, 'cisco', 'cisco')
        result[hostname] = str(device_result)
    return jsonify(result)
```

*Of course, looping through all the devices in `Device.query.all()` is not efficient, as in the preceding code. In production, we will use a SQL query that specifically targets the role of the device.*

When we use the REST API, we can see that all the spine as well as leaf devices can be queried at the same time:

```
$ http GET http://172.16.1.173:5000/devices/spine/version
HTTP/1.0 200 OK
...
{
 "iosv-1": "('iosv-1', b'show version | i V\\r\\nCisco IOS Software, IOSv Software
(VIOS-ADVENTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE (fc2)\\r\\n')",
 "iosv-2": "('iosv-2', b'show version | i V\\r\\nCisco IOS Software, IOSv Software
(VIOS-ADVENTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE (fc2)\\r\\nProcessor
board ID 9T7CB2J2V6F0DLWK7V48E\\r\\n')"
}


$ http GET http://172.16.1.173:5000/devices/leaf/version
HTTP/1.0 200 OK
...
{
 "iosv-3": "('iosv-3', b'show version | i V\\r\\nCisco IOS Software, IOSv Software
(VIOS-ADVENTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE (fc2)\\r\\nProcessor
board ID 9MGG8EA1E0V2PE2D8KDD7\\r\\n')",
 "iosv-4": "('iosv-4', b'show version | i V\\r\\nCisco IOS Software, IOSv Software
(VIOS-ADVENTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE (fc2)\\r\\nProcessor
board ID 9U96V39A4Z12PCG4O6Y0Q\\r\\n')"
}
```

As illustrated, the new API endpoints query the device(s) in real time and return the result to the requester. This works relatively well when you can guarantee a response from the operation within the timeout value of the transaction (30 seconds by default) or if you are OK with the HTTP session timing out before the operation is completed. One way to deal with the timeout issue is to perform the tasks asynchronously. We will look at an example in the next section.

# Asynchronous operations

Asynchronous operations are, in my opinion, an advanced topic of Flask. Luckily, Miguel Grinberg (https://blog.miguelgrinberg.com/), whose Flask work I am a big fan of, provides many posts and examples on his blog and GitHub. For asynchronous operations, the example code in `chapter9_8.py` referenced Miguel's GitHub code on the `Raspberry Pi` file (https://github.com/miguelgrinberg/oreilly-flask-apis-video/blob/master/camera/camera.py) for the background decorator. We start by importing a few more modules:

```
from flask import Flask, url_for, jsonify, request,\
    make_response, copy_current_request_context
...
import uuid
import functools
from threading import Thread
```

The background decorator takes in a function and runs it as a background task using Thread and UUID for the task ID. It returns the status code `202` accepted and the location of the new resources for the requester to check. We will make a new URL for status checking:

```
@app.route('/status/<id>', methods=['GET'])
def get_task_status(id):
    global background_tasks
    rv = background_tasks.get(id)
    if rv is None:
        return not_found(None)

    if isinstance(rv, Thread):
        return jsonify({}), 202, {'Location': url_for('get_task_status', id=id)}

    if app.config['AUTO_DELETE_BG_TASKS']:
        del background_tasks[id]
    return rv
```

Once we retrieve the resource, it is deleted. This was done via setting `app.config['AUTO_DELETE_BG_TASKS']` to true at the top of the app. We will add this decorator to our version endpoints without changing the other part of the code because all of the complexity is hidden in the decorator (how cool is that!):

```
@app.route('/devices/<int:id>/version', methods=['GET'])
@background
def get_device_version(id):
    device = Device.query.get_or_404(id)
...

@app.route('/devices/<device_role>/version', methods=['GET'])
@background
def get_role_version(device_role):
    device_id_list = [device.id for device in Device.query.all() if device.role ==
device_role]
...
```

The end result is a two-part process. We will perform the `GET` request for the endpoint and receive the location header:

```
$ http GET http://172.16.1.173:5000/devices/spine/version
HTTP/1.0 202 ACCEPTED
Content-Length: 2
Content-Type: application/json
Date: <skip>
Location: http://172.16.1.173:5000/status/d02c3f58f4014e96a5dca075e1bb65d4
Server: Werkzeug/0.9.6 Python/3.5.2

{}
```

We can then make a second request to the location to retrieve the result:

```
$ http GET http://172.16.1.173:5000/status/d02c3f58f4014e96a5dca075e1bb65d4
HTTP/1.0 200 OK
Content-Length: 370
Content-Type: application/json
Date: <skip>
Server: Werkzeug/0.9.6 Python/3.5.2

{
 "iosv-1": "('iosv-1', b'show version | i V\\r\\nCisco IOS Software, IOSv Software
(VIOS-ADVENTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE (fc2)\\r\\n')",
 "iosv-2": "('iosv-2', b'show version | i V\\r\\nCisco IOS Software, IOSv Software
(VIOS-ADVENTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE (fc2)\\r\\nProcessor
board ID 9T7CB2J2V6F0DLWK7V48E\\r\\n')"
}
```

To verify that the status code `202` is returned when the resource is not ready, we will use the following script to immediately make a request to the new resource:

```
import requests, time

server = 'http://172.16.1.173:5000'
endpoint = '/devices/1/version'

# First request to get the new resource
r = requests.get(server+endpoint)
```

```
resource = r.headers['location']
print("Status: {} Resource: {}".format(r.status_code, resource))

# Second request to get the resource status
r = requests.get(resource)
print("Immediate Status Query to Resource: " + str(r.status_code))

print("Sleep for 2 seconds")
time.sleep(2)
# Third request to get the resource status
r = requests.get(resource)
print("Status after 2 seconds: " + str(r.status_code))
```

As you can see in the result, the status code is returned while the resource is still being run in the background as `202`:

```
$ python chapter9_request_1.py
Status: 202 Resource:
http://172.16.1.173:5000/status/1de21f5235c94236a38abd5606680b92
Immediate Status Query to Resource: 202
Sleep for 2 seconds
Status after 2 seconds: 200
```

Our APIs are coming along nicely. Because our network resource is valuable to us, we should secure API access to only authorized personnel. We will add basic security measures to our API in the next section.

## Security

For user authentication security, we will use Flask's extension httpauth, written by Miguel Grinberg, as well as the password functions in Werkzeug. The httpauth extension should have been installed as part of the `requirements.txt` installation in the beginning of the chapter. The file is named `chapter9_9.py`; we will start with a few more module imports:

```
...
from werkzeug.security import generate_password_hash, check_password_hash
from flask.ext.httpauth import HTTPBasicAuth
...
```

We will create an `HTTPBasicAuth` object as well as the user database object. Note that during the user creation process, we will pass the password value; however, we are only storing `password_hash` instead of the `password` itself:

```
auth = HTTPBasicAuth()

class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), index=True)
    password_hash = db.Column(db.String(128))

    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

    def verify_password(self, password):
        return check_password_hash(self.password_hash, password)
```

The `auth` object has a `verify_password` decorator that we can use, along with Flash's g global context object that was created when the request started for password verification. Because `g` is global, if we save the user to the `g` variable, it will live through the entire transaction:

```
@auth.verify_password
def verify_password(username, password):
    g.user = User.query.filter_by(username=username).first()
    if g.user is None:
        return False
    return g.user.verify_password(password)
```

There is a handy `before_request` handler that can be used before any API endpoint is called. We will combine the `auth.login_required` decorator with the `before_request` handler that will be applied to all the API routes:

```
@app.before_request
@auth.login_required
def before_request():
    pass
```

Lastly, we will use the unauthorized error handler to return a response object for the `401` unauthorized error:

```
@auth.error_handler
def unathorized():
    response = jsonify({'status': 401, 'error': 'unahtorized',
                        'message': 'please authenticate'})

    response.status_code = 401
    return response
```

Before we can test user authentication, we will need to create users in our database:

```
>>> from chapter9_9 import db, User
>>> db.create_all()
>>> u = User(username='eric')
>>> u.set_password('secret')
>>> db.session.add(u)
>>> db.session.commit()
>>> exit()
```

Once you start your Flask development server, try to make a request as before; however, this time it will be rejected and presented with a `401` unauthorized error:

```
$ http GET http://172.16.1.173:5000/devices/
HTTP/1.0 401 UNAUTHORIZED
Content-Length: 81
Content-Type: application/json
Date: <skip>
Server: Werkzeug/0.9.6 Python/3.5.2
WWW-Authenticate: Basic realm="Authentication Required"

{
 "error": "unauthorized",
 "message": "please authenticate",
 "status": 401
}
```

We will now need to provide the authentication header for our requests:

```
$ http --auth eric:secret GET http://172.16.1.173:5000/devices/
HTTP/1.0 200 OK
Content-Length: 188
Content-Type: application/json
Date: <skip>
Server: Werkzeug/0.9.6 Python/3.5.2

{
    "device": [
        "http://172.16.1.173:5000/devices/1",
        "http://172.16.1.173:5000/devices/2",
        "http://172.16.1.173:5000/devices/3",
        "http://172.16.1.173:5000/devices/4"
    ]
}
```

We now have a decent RESTful API setup for our network. The user will be able to interact with the APIs now instead of the network devices. They can query for static contents of the network as well as perform tasks for individual devices or a group of devices. We also added basic security measures to ensure only the users we created are able to retrieve the information from our API.

We have now abstracted the underlying vendor API away from our network and replaced them with our own RESTful API. We are free to use what is required, such as Pexpect, while still providing a uniform frontend to our requester.

Let's take a look at additional resources for Flask so we can continue to build on our API framework.

## Additional resources

Flask is no doubt a feature-rich framework that is growing in features and community. We have covered a lot of topics in this chapter but still have only scraped the surface of the framework. Besides APIs, you can use Flask for web applications as well as your websites. There are a few improvements that I think we can still make to our network API framework:

- Separate out the database and each endpoint in its own file so the code is clean and easier to troubleshoot.
- Migrate from SQLite to other production-ready databases.
- Use token-based authentication instead of passing the username and password for every transaction. In essence, we will receive a token with finite expiration time upon initial authentication and use the token for further transactions until the expiration.
- Deploy your Flask API app behind a web server, such as Nginx, along with the Python WSGI server for production use.

Obviously, the preceding improvements will vary greatly from company to company. For example, the choice of database and web server may have implications from the company's technical preference as well as the other team's input. The use of token-based authentication might not be necessary if the API is only used internally and other forms of security have been put into place. For these reasons, I would like to provide you with additional links as extra resources should you choose to move forward with any of the preceding items.

Here are some of the links I find useful when thinking about design patterns, database options, and general Flask features:

- Best practices on Flask design patterns, http://flask.pocoo.org/docs/0.10/patterns/
- Flask API, http://flask.pocoo.org/docs/0.12/api/
- Deployment options, http://flask.pocoo.org/docs/0.12/deploying/

Due to the nature of Flask and the fact that it relies on the extension outside of its small core, sometimes you might find yourself jumping from one documentation to another. This can be frustrating, but the upside is that you only need to know the extension you are using, which I feel saves time in the long run.

## Summary

In this chapter, we started moving down the path of building REST APIs for our network. We looked at different popular Python web frameworks, namely Django and Flask, and compared and contrasted between the two. By choosing Flask, we are able to start small and expand on features by adding Flask extensions.

In our lab, we used the virtual environment to separate the Flask installation base from our global site-packages. The lab network consist of four nodes, two of which we have designated as spine routers while the other two as leafs. We took a tour of the basics of Flask and used the simple HTTPie client for testing our API setup.

Among the different setups of Flask, we placed special emphasis on URL dispatch as well as the URL variables because they are the initial logic between the requesters and our API system. We took a look at using Flask-SQLAlchemy and sqlite to store and return network elements that are static in nature. For operation tasks, we also created API endpoints while calling other programs, such as Pexpect, to accomplish our tasks. We improved the setup by adding asynchronous handling as well user authentication to our API. Toward the end of the chapter, we look at some of the additional resource links we can follow to add even more security and other features.

In the next chapter, we will shift our gear to take a look at **Software-Defined Networking** (**SDN**), starting with the original technology that some considered started the SDN movement: OpenFlow.

# OpenFlow Basics

Up to this point in the book, we have been working with an existing networks and automating various tasks using Python. We started with basic pseudo-human automation using SSH, interaction with the API, higher-level abstraction with Ansible, network security, and network monitoring. In the previousÂ chapter, we also looked at how to build our own network API with the Flask framework. The book is structured this way by design to focus on working with the network engineering field as it is today with the dominating vendors such as Cisco, Juniper, and Arista. The skills introduced so far in the book will help you fill in the gap as you scale and grow your network.Â

One point of this consistent theme is how we are working within the realm of vendor walls. For example, if aÂ Cisco IOS device does not provide an API interface, we have to use pexpect and paramiko for SSH. If the device only supports **Simple Network Management Protocol** (**SNMP**) for network monitoring, we have to work with SNMP. There are ways we can get around these limitations, but, for the most part, they feel ad hoc and sometimes hard to implement without feeling like you are voiding your device warranty.

Perhaps sparked by these frustrations and the need to rapidly develop new network features, the **Software Defined Networking** (**SDN**) movement was born. If you work in the industry and have not lived under a rock for the last few years, no doubt you have heard of terms such as SDN, OpenFlow, OpenDaylight, and **Network Function Virtualization** (**NFV**). Perhaps you even picked up this book because you have heard about *software eating the networking world*Â and wanted to prepare yourself for the future. SDN no doubt offers real value and clearly drives innovation at a pace that has not happened in the network-engineering world for a long time.Â

*As an example of SDN-driven innovation, one of the hottest areas of networking, is software-defined WAN (https://en.wikipedia.org/wiki/Software-defined_networking), which many predict will completely replace routers in the edge of the WAN in a multi-*

*billion US dollar industry.Â*

Understanding SDN, however, is not as straightforward as it may seem. In any new technology such as SDN, there is much marketing and hype in the beginning. When there is no common standard, the technology is open for interpretation, especially when the incumbent feels the need to maintain their lead in the space by injecting their own *flavor*Â to the mix. If you were looking for an example, look no further than Cisco's initial support for OpenFlow in the Nexus 5500 and 6000 line of switches, then the introduction of the OpenFlow-competing OpFlex as well as the Cisco **Open Network Environment** (**ONE**) software controller and Cisco **One Platform Kit** (**onePK**). I get tired just byÂ keeping track all the Cisco SDN-related acronyms.Â

*I lay emphasis on Cisco since I follow them more closely because they are the long-term market leaders.Â*

In this chapter, I would like to focus on the technology that many consider started the SDN evolution, OpenFlow. In a nutshell, OpenFlow is a technology that's layered on top of theÂ **Transmission Control Protocol** (**TCP**)Â that separates the control and forwarding paths between the physical device and the controller. By separating the two functions and communication methods, OpenFlow allows the determination of the network path outside of the physical device. This enables the controller to use the software on the centralized controller to have a holistic view and simplify the amount of intelligence needed on the individual network equipment.

In this chapter, we will cover the following topics:

- Setting up an OpenFlow lab using virtual machinesÂ
- IntroducingÂ the OpenFlow protocol
- Using Mininet for network simulation
- Taking a look at a Python-based OpenFlow controller, Ryu, to construct a Layer 2 OpenFlow switch
- Using the same Ryu controller to build a Layer 3 firewall application

- IntroducingÂ an alternative Python OpenFlow controller, POX

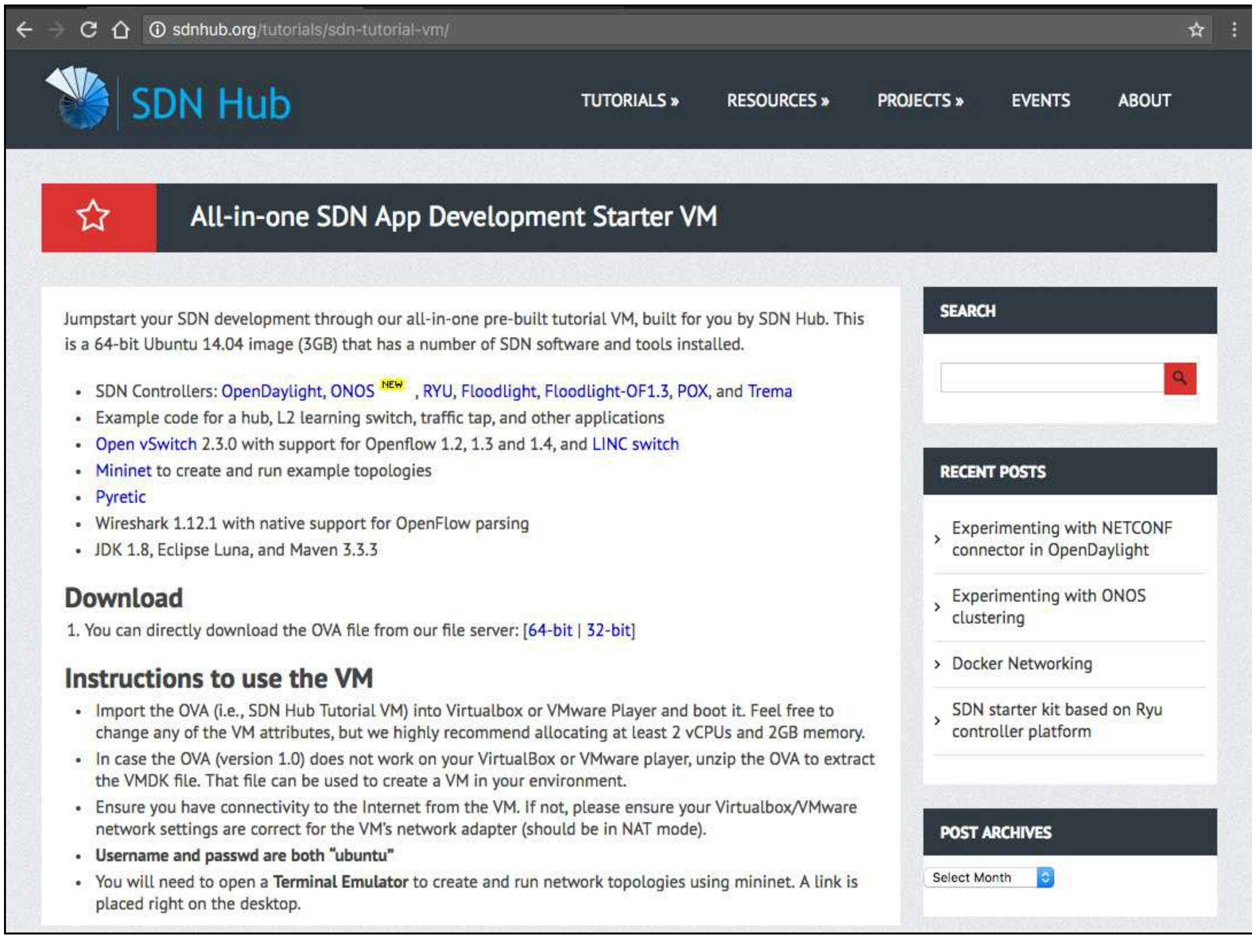
Let's get started.Â

## Lab setup

We will be using a number of tools, namely Mininet, Open vSwitch, the OpenFlow Ryu controller, and Wireshark with the OpenFlow dissector. We can install each of these tools separately; however, it will take a while before we get everything installed and configured correctly. Luckily, SDN Hub (http://sdnhub.org) provides an all-in-one SDN app development starter VM (http://sdnhub.org/tutorials/sdn-tutorial-vm/) that includes all of the tools mentioned here and more. The base operating system is Ubuntu 14.04, which is similar to the VM that we have been using up to this point, and it also contains all the popular SDN controllers in one setting, such as OpenDaylight, ONOS, Ryu, Floodlight, POX, and Trema.

*SDN Hub also provides several useful tutorials under the Tutorials header, including another Python-based OpenFlow controller, POX (http://sdnhub.org/tutorials/pox/).*

For this chapter and the next, we will be using this all-in-one image to have a self-contained OpenFlow network as a starting point to experiment and learn about OpenFlow and SDN. It is a pretty big image to be downloaded, at about 3 GB, but for the amount of work it saves, I feel it is worth the effort. Start the download, grab a cup of coffee, and off we go!


sdnhub.org/tutorials/sdn-tutorial-vm/
SDN Hub
TUTORIALS »
RESOURCES »
PROJECTS »
EVENTS
ABOUT
SEARCH
RECENT POSTS
Experimenting with NETCONF connector in OpenDaylight
Experimenting with ONOS clustering
Docker Networking
SDN starter kit based on Ryu controller platform
POST ARCHIVES
Select Month

All-in-one SDN App Development Starter VM

Jumpstart your SDN development through our all-in-one pre-built tutorial VM, built for you by SDN Hub. This is a 64-bit Ubuntu 14.04 image (3GB) that has a number of SDN software and tools installed.

- SDN Controllers: OpenDaylight, ONOS NEW, RYU, Floodlight, Floodlight-OF1.3, POX, and Trema
- Example code for a hub, L2 learning switch, traffic tap, and other applications
- Open vSwitch 2.3.0 with support for Openflow 1.2, 1.3 and 1.4, and LINC switch
- Mininet to create and run example topologies
- Pyretic
- Wireshark 1.12.1 with native support for OpenFlow parsing
- JDK 1.8, Eclipse Luna, and Maven 3.3.3

Download

1. You can directly download the OVA file from our file server: [64-bit | 32-bit]

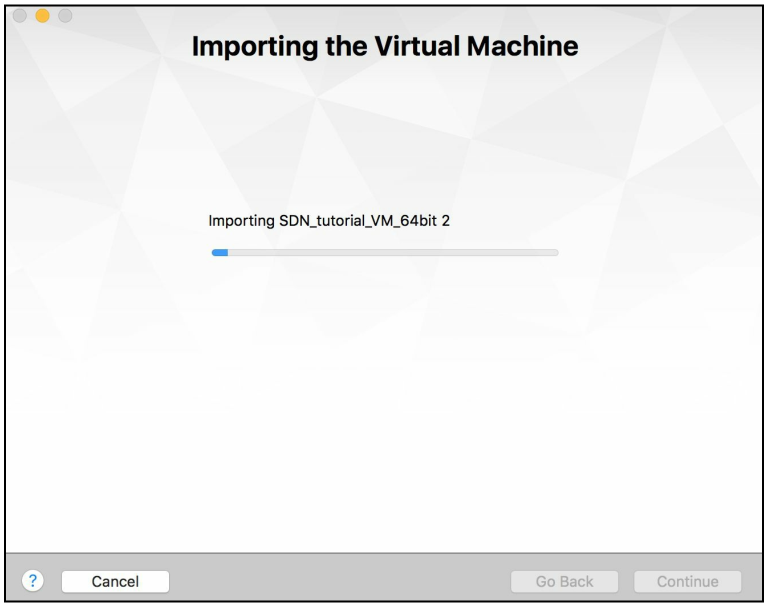
Instructions to use the VM

- Import the OVA (i.e., SDN Hub Tutorial VM) into Virtualbox or VMware Player and boot it. Feel free to change any of the VM attributes, but we highly recommend allocating at least 2 vCPUs and 2GB memory.
- In case the OVA (version 1.0) does not work on your VirtualBox or VMware player, unzip the OVA to extract the VMDK file. That file can be used to create a VM in your environment.
- Ensure you have connectivity to the Internet from the VM. If not, please ensure your Virtualbox/VMware network settings are correct for the VM's network adapter (should be in NAT mode).
- **Username and passwd are both "ubuntu"**
- You will need to open a **Terminal Emulator** to create and run network topologies using mininet. A link is placed right on the desktop.

SDN Hub all-in-one image download

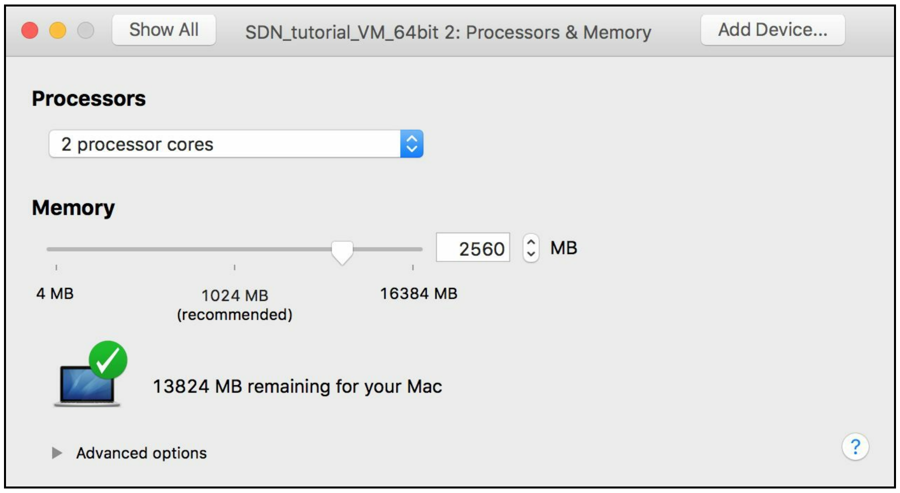
After the OVA image download, you can run the image on a number of hypervisors, including VirtualBox and VMWare. I would recommend using the same hypervisor as your VIRL setup, since in the following chapters, we will try to use the OpenFlow environment to interact with the VIRL VM. You can also use a different hypervisor, such as Hyper-V or VirtualBox, as long as you bridge the interfaces to the same virtual network as the VIRL VM for future chapters.

In my personal setup, I imported the image to VMware Fusion:


Importing the Virtual Machine
Importing SDN_tutorial_VM_64bit 2
?
Cancel
Go Back
Continue

VMWare Fusion import

The recommended resource allocation is 2 vCPU and 2 GB of memory:


SDN_tutorial_VM_64bit 2: Processors & Memory

**Processors**

2 processor cores

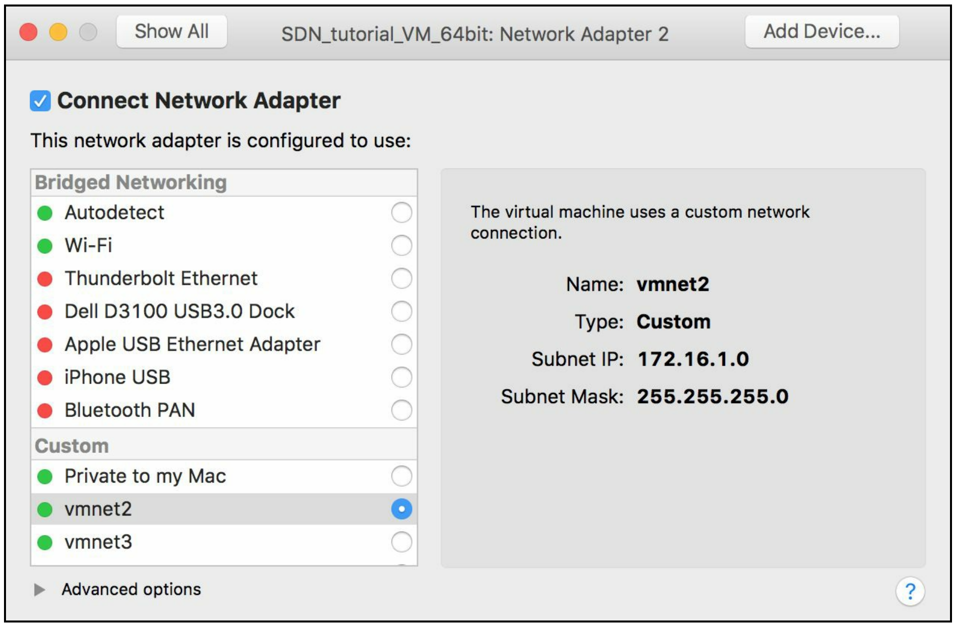
**Memory**

2560 MB

13824 MB remaining for your Mac

Image resource allocation

For consistency, I have also added additional network interfaces to the VMNet, similar to the setup for the virtual machine we have been using:



Connect Network Adapter [✓]

This network adapter is configured to use:

Bridged Networking
- ● Autodetect
- ● Wi-Fi
- ● Thunderbolt Ethernet
- ● Dell D3100 USB3.0 Dock
- ● Apple USB Ethernet Adapter
- ● iPhone USB
- ● Bluetooth PAN

Custom
- ● Private to my Mac
- ● vmnet2 [✓]
- ● vmnet3

The virtual machine uses a custom network connection.

| | |
|---|---|
| Name: | vmnet2 |
| Type: | Custom |
| Subnet IP: | 172.16.1.0 |
| Subnet Mask: | 255.255.255.0 |

Advanced options

VMNet setup

We will add the IP address to `/etc/network/interfaces/` in order to preserve settings after reboot:

*The default username and password combination is* `ubuntu`/`ubuntu`.

```
ubuntu@sdnhubvm:~[06:56]$ cat /etc/network/interfaces
...
auto eth1
iface eth1 inet static
 address 172.16.1.174
 netmask 255.255.255.0
```

We are now ready to work with OpenFlow, so let's take a quick overview of the OpenFlow protocol.

# Introducing OpenFlow

Let's start with an honest statement: OpenFlow is not SDN, and SDN is not OpenFlow. People often interchanged the two terms in the early days, but they are certainly not the same. OpenFlow is a very important building block of SDN, and many credit it to be the origin of the SDN movement. OpenFlow originally started at Stanford University as a research project, which eventually jump-started the startups of Nicira and Big Switch Networks. Nicira was acquired by VMWare and is now part of the company's network-virtualization product line. Big Switch Networks leads a number of OpenFlow open source projects such as Floodlight Controller, Switch Light OS for bar-metal Ethernet switches, and Switch Light vSwitch for virtual switches.

The original version of OpenFlow 1.1 was released in February 2011, and soon after the release, the OpenFlow effort was overseen by the Open Network Foundation, which retains control for the 1.2 release and beyond. In 2012, Google made a big wave by describing how the company's internal network had been redesigned to run under OpenFlow at the Open Networking Summit. The current version of OpenFlow is 1.4, with most switches supporting version 1.3.

*You can refer to the OpenFlow 1.4 specification at https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.*

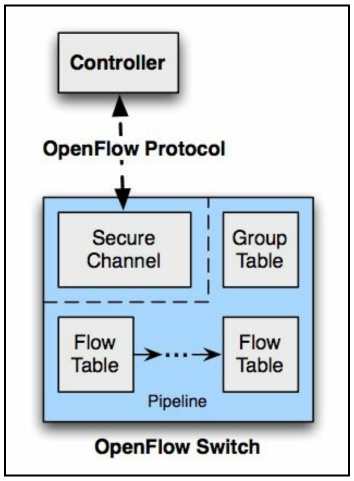
*You can refer to the OpenFlow 1.3 specification here https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf.*

In most cases, the controller is a software-based package that supports a range of OpenFlow versions from 1.1 and up. The switches, however, vary in their support for the version of OpenFlow specification. To help with the adaptation and increase confidence, the Open Network Foundation provides certification labs for switches (https://www.opennetworking.org/openflow-conformance-certification) and publishes the list of certified switches. In addition, the controller manufacturer, such as Ryu, might also perform their own compatibility tests (https://osrg.github.io/ryu/certification.html) and publish the results.

Let's look at the basic OpenFlow operations in the next section.

# Basic operations

At its core, the OpenFlow protocol specifies how the switch and the controller should communicate with each other. When the switch initially boots up, it registers itself with the controller. In most cases, the controller IP is specified to the switch either during the bootup process or configured on the switch out of band via management. The switch will then initiate the channel to communicate with the controller:



OpenFlow switch to controller communication
(source: http://archive.openflow.org/documents/openflow-spec-v1.1.0.pdf)

Following the bootup, the switch will communicate with the controller via several types of messages; the main messages are as follows:

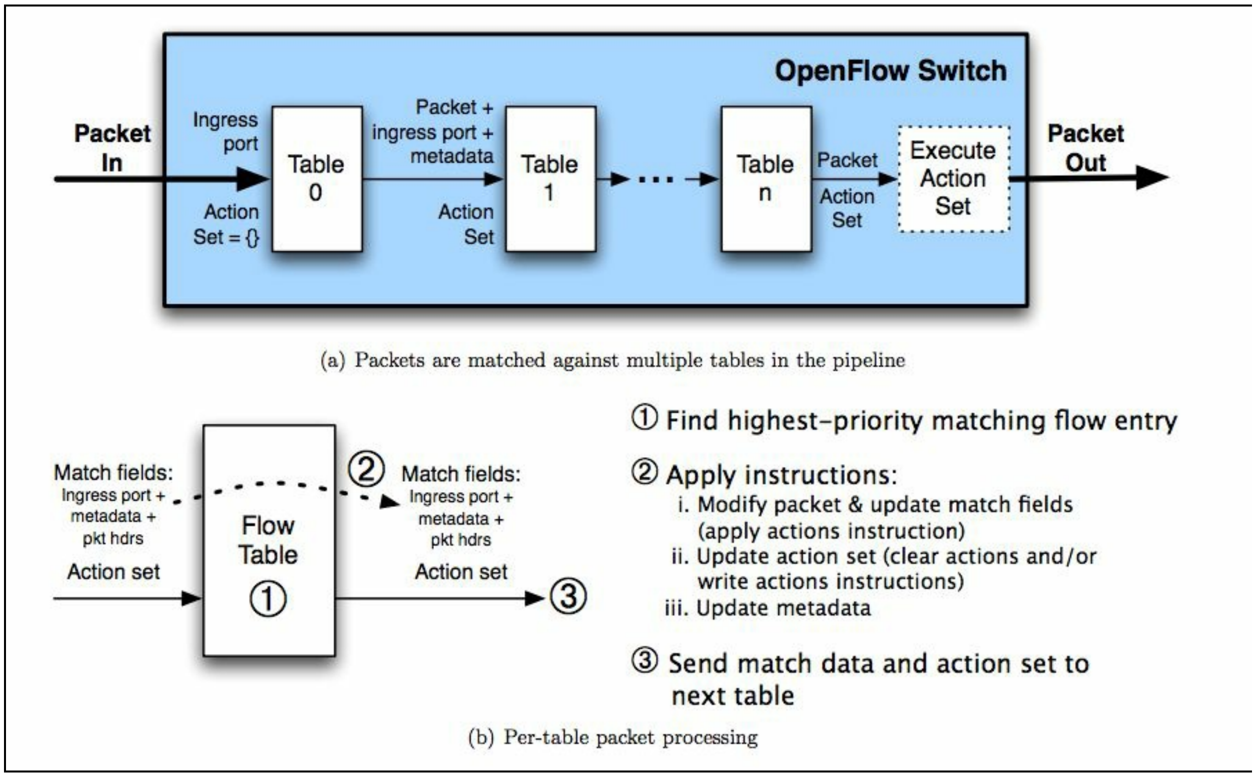
- **Symmetric**: Symmetric messages are initiated by either the switch or controller. Typical symmetric messages are hello messages, echo request/reply messages, and other experimental messages for future features.
- **Controller-to-switch**: Controller-to-switch messages are initiated by the controller to manage the switch or query the switch for its state. They include feature and configuration queries from controller to switch, statistics collection, and barrier messages for operation-completion notifications. Also, the controller can initiate modify-state messages to

change flows and groups on the switch as well as packet-out in response to switch PacketIn messaged to forward a packet out of a switch port.
- **Asynchronous**: The switches send the controller asynchronous messages for packet arrival, state change, or error. When the packet arrives and the switch does not have an existing matching flow entry, the event will be sent to the controller for action as a `PacketIn` message. The controller will respond with a packet-out message. The switch will also send any error and port-status change as well as flow removal as asynchronous messages to the controller.

Obviously, there is no need to memorize the different messages: the common messages will become natural to you the more you use and see them. The more obscure messages can always been looked up when needed. What is important is the fact that the switch and the controller are both in constant communication, and they need to synchronize to a certain degree in order for the network to function properly.

The point of the channel establishment and all the messages exchanged is to allow the network device to take proper forwarding action based on what it knows about the packet. We will look at the available matching fields in a bit, but the forwarding decisions are generally done by matching with the flow entries. Many flow entries can exist in a flow table, and each OpenFlow switch contains multiple flow tables at different priorities. When a packet arrives at the device, it is matched against the highest-priority flow table and the flow entries in the table. When a match is found, an action such as forwarding out an interface or going to another table is executed. If there is no match, for example a table miss, you can configure the action to be as follows:

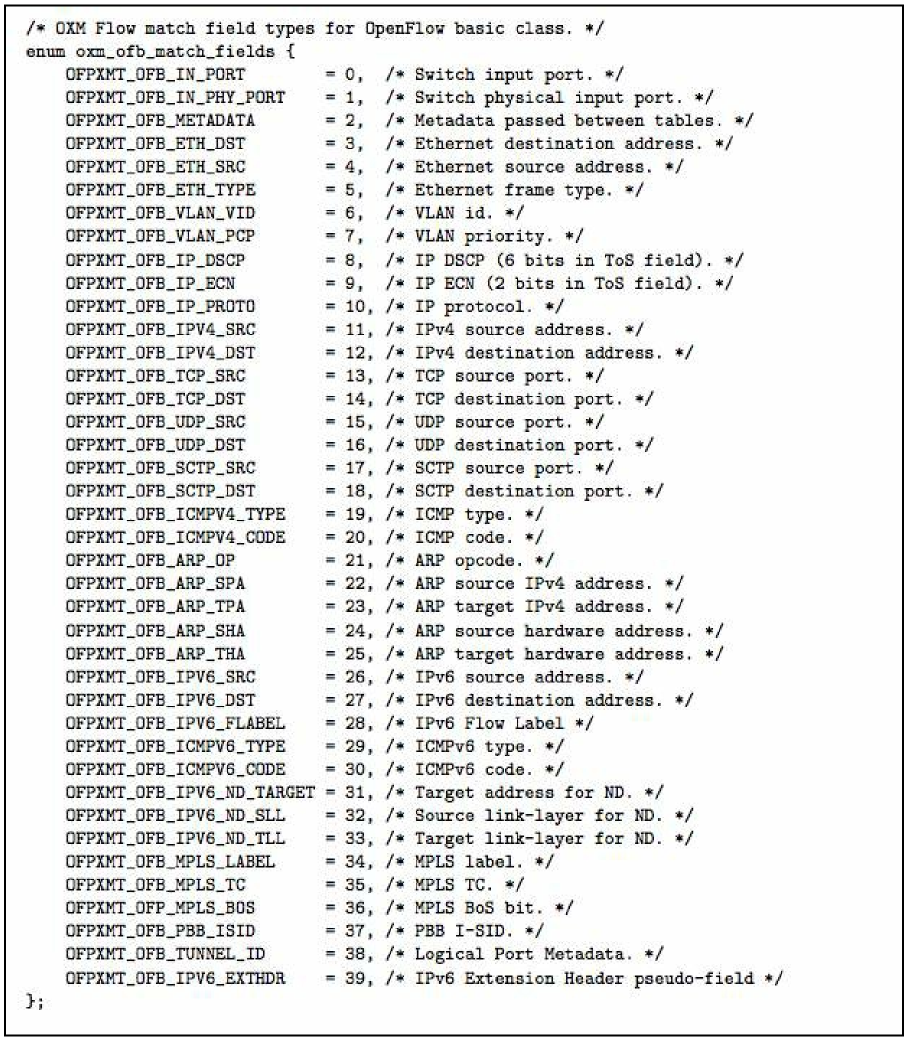
- Drop the packet
- Pass the packet to another table
- Send the packet to the controller via a `PacketIn` message and wait for further instructions


OpenFlow Switch
Packet In
Ingress port
Action Set = {}
Table 0
Packet + ingress port + metadata
Action Set
Table 1
Table n
Packet
Action Set
Execute Action Set
Packet Out
(a) Packets are matched against multiple tables in the pipeline
Match fields: Ingress port + metadata + pkt hdrs
Action set
Flow Table ①
② Match fields: Ingress port + metadata + pkt hdrs
Action set
③
① Find highest-priority matching flow entry
② Apply instructions:
i. Modify packet & update match fields (apply actions instruction)
ii. Update action set (clear actions and/or write actions instructions)
iii. Update metadata
③ Send match data and action set to next table
(b) Per-table packet processing

The OpenFlow packet processing pipeline
(source: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf)

As you can see, the OpenFlow operation is relatively simple and straightforward. Besides the separation of control and data planes, the uniqueness of OpenFlow is that it allows packets to be granularly matched, thus enabling a finer control over your packet flow.

As of OpenFlow 1.3, the following set of match fields are specified:



```
/* OXM Flow match field types for OpenFlow basic class. */
enum oxm_ofb_match_fields {
    OFPXMT_OFB_IN_PORT        = 0,  /* Switch input port. */
    OFPXMT_OFB_IN_PHY_PORT    = 1,  /* Switch physical input port. */
    OFPXMT_OFB_METADATA       = 2,  /* Metadata passed between tables. */
    OFPXMT_OFB_ETH_DST        = 3,  /* Ethernet destination address. */
    OFPXMT_OFB_ETH_SRC        = 4,  /* Ethernet source address. */
    OFPXMT_OFB_ETH_TYPE       = 5,  /* Ethernet frame type. */
    OFPXMT_OFB_VLAN_VID       = 6,  /* VLAN id. */
    OFPXMT_OFB_VLAN_PCP       = 7,  /* VLAN priority. */
    OFPXMT_OFB_IP_DSCP        = 8,  /* IP DSCP (6 bits in ToS field). */
    OFPXMT_OFB_IP_ECN         = 9,  /* IP ECN (2 bits in ToS field). */
    OFPXMT_OFB_IP_PROTO       = 10, /* IP protocol. */
    OFPXMT_OFB_IPV4_SRC       = 11, /* IPv4 source address. */
    OFPXMT_OFB_IPV4_DST       = 12, /* IPv4 destination address. */
    OFPXMT_OFB_TCP_SRC        = 13, /* TCP source port. */
    OFPXMT_OFB_TCP_DST        = 14, /* TCP destination port. */
    OFPXMT_OFB_UDP_SRC        = 15, /* UDP source port. */
    OFPXMT_OFB_UDP_DST        = 16, /* UDP destination port. */
    OFPXMT_OFB_SCTP_SRC       = 17, /* SCTP source port. */
    OFPXMT_OFB_SCTP_DST       = 18, /* SCTP destination port. */
    OFPXMT_OFB_ICMPV4_TYPE    = 19, /* ICMP type. */
    OFPXMT_OFB_ICMPV4_CODE    = 20, /* ICMP code. */
    OFPXMT_OFB_ARP_OP         = 21, /* ARP opcode. */
    OFPXMT_OFB_ARP_SPA        = 22, /* ARP source IPv4 address. */
    OFPXMT_OFB_ARP_TPA        = 23, /* ARP target IPv4 address. */
    OFPXMT_OFB_ARP_SHA        = 24, /* ARP source hardware address. */
    OFPXMT_OFB_ARP_THA        = 25, /* ARP target hardware address. */
    OFPXMT_OFB_IPV6_SRC       = 26, /* IPv6 source address. */
    OFPXMT_OFB_IPV6_DST       = 27, /* IPv6 destination address. */
    OFPXMT_OFB_IPV6_FLABEL    = 28, /* IPv6 Flow Label */
    OFPXMT_OFB_ICMPV6_TYPE    = 29, /* ICMPv6 type. */
    OFPXMT_OFB_ICMPV6_CODE    = 30, /* ICMPv6 code. */
    OFPXMT_OFB_IPV6_ND_TARGET = 31, /* Target address for ND. */
    OFPXMT_OFB_IPV6_ND_SLL    = 32, /* Source link-layer for ND. */
    OFPXMT_OFB_IPV6_ND_TLL    = 33, /* Target link-layer for ND. */
    OFPXMT_OFB_MPLS_LABEL     = 34, /* MPLS label. */
    OFPXMT_OFB_MPLS_TC        = 35, /* MPLS TC. */
    OFPXMT_OFP_MPLS_BOS       = 36, /* MPLS BoS bit. */
    OFPXMT_OFB_PBB_ISID       = 37, /* PBB I-SID. */
    OFPXMT_OFB_TUNNEL_ID      = 38, /* Logical Port Metadata. */
    OFPXMT_OFB_IPV6_EXTHDR    = 39, /* IPv6 Extension Header pseudo-field */
};
```

OpenFlow 1.3 flow match fields
(source: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf)

From the preceding chart, many of the fields OpenFlow can match against are no different than than the existing layer 2 to 4 headers. So what is the big deal? If we look back at traditional network devices, a switch that is examining VLAN information might not be looking at IP headers, a router making forwarding decisions based on IP headers is not digging into TCP ports, and an MPLS-capable device might only be dealing with label headers. OpenFlow allows network devices to be controlled by a software controller that looks at all of the available information; thus, it can make a so-called *white box* to be any network function as it desires. It also allows the software controller to have a total view of the network instead of each device making forwarding decisions to the best of its knowledge. This makes OpenFlow or other software controllers that provide the same function very desirable for network operators.

## OpenFlow 1.0 vs 1.3

You might be wondering about the difference between OpenFlow 1.0 vs. 1.3, and what happened to OpenFlow 1.2. OpenFlow 1.0 represents the very first effort to introduce the concept of *programmable flow* in switches to the world. It includes a limited subset of switch-matching and action functions, but many in the industry do not consider OpenFlow 1.0 to include sufficient features for production networks. It is really about the introduction of the concept and showing the potential of the technology to the industry.

The concept was well received by the industry, and the standard makers were quick to start working on iterations of the specification, such as version 1.1 and 1.2. However, hardware implementation of a technology takes much longer than the speed at which the specifications are written. Many of the vendors cannot keep up with the speed of specification releases. It became obvious that the specification needs to slow down a bit and work with equipment makers so there can be actual hardware to run our networks when the specification comes out. OpenFlow 1.3 was written in conjunction with equipment makers' timing to have hardware switch support. It is the first version of OpenFlow considered by many to be production ready.

Now that we have learned about the basics of OpenFlow, we are ready to take a look at the protocol in action. But before we can do that, we will need a switch that understands OpenFlow as well as a few hosts connected to that switch. Luckily, Mininet provides both. In the next section, we will look at Mininet, a network emulation tool that we can use to experiment with OpenFlow.

# Mininet

Mininet, from http://mininet.org/, creates a virtual network working with the OpenFlow controller on a single Linux kernel. It uses lightweight virtualization to make a single system look like a complete network, consisting of end-hosts, switches, routers, and links. This is similar to VIRL, or perhaps GNS3, which we have been using, except it is much lighter in weight than either, so you can run more nodes much faster.

All Mininet commands starts with `mn`, and you can get a help menu with the

`-h` option. Among the options, pay special attention to the switch, controller, and topology options, which we will use often. They are used to specify the type of switch, controller type, location, and the topology. The MAC option automatically sets the host MAC address to easily identify the hosts when layer 2 is involved, such as when the host is sending ARP requests:

```
$ sudo mn -h
...
Options:
 ...
 --switch=SWITCH default|ivs|lxbr|ovs|ovsbr|ovsk|user[,param=value...]
 --controller=CONTROLLER
 --topo=TOPO linear|minimal|reversed|single|torus|tree[,param=value
 --mac automatically set host MACs
...
```

Mininet provides several defaults; to launch the basic topology with default settings, you can simply type in the `sudo mn` command:

```
$ sudo mn
*** No default OpenFlow controller found for default switch!
*** Falling back to OVS Bridge
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
```

```
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
```

As you can see, the command creates two hosts and one switch host with a basic OpenFlow controller, and then drops the user into the Mininet CLI. From the CLI, you can interact with the host and gather information about the network, much like the GNS3 console:

```
mininet> nodes
available nodes are:
h1 h2 s1
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
s1 lo: s1-eth1:h1-eth0 s1-eth2:h2-eth0
mininet> links
h1-eth0<->s1-eth1 (OK OK)
h2-eth0<->s1-eth2 (OK OK)
mininet> help

Documented commands (type help <topic>):
...
```

A very useful feature of the Mininet CLI is that you can interact with the hosts directly, as if you were at their console prompts:

```
mininet> h1 ifconfig
h1-eth0 Link encap:Ethernet HWaddr 16:e5:72:7b:53:f8
 inet addr:10.0.0.1 Bcast:10.255.255.255 Mask:255.0.0.0
...

lo Link encap:Local Loopback
 inet addr:127.0.0.1 Mask:255.0.0.0
...

mininet> h2 ps
 PID TTY TIME CMD
 9492 pts/7 00:00:00 bash
10012 pts/7 00:00:00 ps
mininet>
```

There are many more useful options in Mininet, such as changing topology, running iperf tests, introduce delay and bandwidth limitation on links, and running custom topologies. In this chapter, we will mostly use Mininet with the following options:

```
$ sudo mn --topo single,3 --mac --controller remote --switch ovsk
```

The preceding command will start a network emulation with one switch, three hosts, automatically assigned MAC addresses for the hosts, a remote controller, and an Open vSwitch as the switch type.

*The Mininet walk-through provides an excellent tutorial of Mininet at* http://mininet.org/walkthrough/.

We are now ready to look at the OpenFlow controller that we will be using, Ryu.

# The Ryu controller with Python

Ryu is a component-based SDN controller fully written in Python. It is a project backed by **Nippon Telegraph and Telephone** (**NTT**) Labs. The project has Japanese roots; Ryu means "flow" in Japanese and is pronounced "ree-yooh" in English, which matches well with the OpenFlow objective of programming flow in network devices. By being component based, the Ryu framework breaks the software it uses into individual portions that can be taken in part or as a whole. For example, on your virtual machine, under `/home/ubuntu/ryu/ryu`, you can see several folders, including `app`, `base`, and `ofproto`:

```
ubuntu@sdnhubvm:~/ryu/ryu[00:04] (master)$ pwd
 /home/ubuntu/ryu/ryu
ubuntu@sdnhubvm:~/ryu/ryu[00:05] (master)$ ls
 app/ cmd/ exception.pyc __init__.py log.pyc topology/
 base/ contrib/ flags.py __init__.pyc ofproto/ utils.py
 cfg.py controller/ flags.pyc lib/ services/ utils.pyc
 cfg.pyc exception.py hooks.py log.py tests/
```

Each of these folders contains different software components. The base folder contains the `app_manager.py` file, which loads the Ryu applications, provides contexts, and routes messages among Ryu applications. The `app` folder contains various applications that you can load using the `app_manager`, such as layer 2 switches, routers, and firewalls. The `ofproto` folder contains encoders/decoders as per OpenFlow standards: version 1.0, version 1.3, and so on. By focusing on the components, it is easy to only pick the portion of the package that you need. Let's look at an example. First, let's fire up our Mininet network:

```
$ sudo mn --topo single,3 --mac --controller remote --switch ovsk
 ...
 *** Starting CLI:
 mininet>
```

Then, in a separate terminal window, we can run the `simple_switch_13.py` application by placing it as the first argument after `ryu-manager`.
Ryu applications are simply Python scripts:

```
$ cd ryu/
 $ ./bin/ryu-manager ryu/app/simple_switch_13.py
 loading app ryu/app/simple_switch_13.py
 loading app ryu.controller.ofp_handler
 instantiating app ryu/app/simple_switch_13.py of SimpleSwitch13
 instantiating app ryu.controller.ofp_handler of OFPHandler
```

You can run a quick check between the two hosts in Mininet to verify reachability:

```
mininet> h1 ping -c 2 h2
 PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
 64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=8.64 ms
 64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.218 ms

 --- 10.0.0.2 ping statistics ---
 2 packets transmitted, 2 received, 0% packet loss, time 1003ms
 rtt min/avg/max/mdev = 0.218/4.430/8.643/4.213 ms
 mininet>
```

You can also see a few events happening on your screen where you launched the switch application:

```
packet in 1 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
packet in 1 00:00:00:00:00:02 00:00:00:00:00:01 2
packet in 1 00:00:00:00:00:01 00:00:00:00:00:02 1
```

As you would expect, host 1 broadcasts the request, to which host 2 answers. Then host 1 will proceed with sending the ping packet out to host 2. Notice that we have successfully sent two ICMP packets from host 1 to host 2; however, the controller is only seeing 1 packet being sent across. Also notice that the first ping packet took about 8.64 ms, while the second ping packet took about 0.218 ms, almost a 40x improvement! Why is that? What was the cause of the speed improvement?

You may have already guessed it if you have read the OpenFlow specification or have programmed a switch before. If you put yourself in the switch's position--*be the switch*, if you will--when you received the first broadcast packet, you wouldn't know what to do with the packet. This is considered a table miss because there is no flow in any of the tables. If you recall, we have three options for table misses; the two obvious options would be to either drop the packet or to send it to the controller and wait for further instructions. As it turns out, `simple_switch_13.py` installed a flow action for flow misses to be sent to the controller, so the controller would see any table miss as a `PacketIn` event:

```
actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
ofproto.OFPCML_NO_BUFFER)]
self.add_flow(datapath, 0, match, actions)
```

That explains the first three packets, since they were unknown to the switch. The second question we had was why did the controller not see the second ICMP packet from `h1` to `h2`? As you would have already guessed, by the time we saw the MAC address from `h2` to `h1`, we would have had enough information to install the flow entries. Therefore, when the second ICMP from `h1` to `h2` arrives at the switch, the switch already has a flow match and no longer needs to send the packet to the controller. This is why the second ICMP had a dramatic time improvement: most of the delay in the first packet was caused by the controller processing and communication.

This brings us to a good point to to discuss some of the Open vSwitch commands that we can use to interact with the switch directly.

# Open vSwitch commands

There are several command-line options for Open vSwitch as it has become a very popular software-based switch for projects such as OpenStack networking and OpenFlow. There are three popular command-line tools for Open vSwitch administration:

1. `ovs-vsctl`: This configures Open vSwitch parameters such as ports, adding or deleting bridges, and VLAN tagging
2. `ovs-dpctl`: This configures the data path of Open vSwitch, such as showing the currently installed flows
3. `ovs-ofctl`: This combines the functions of `ovs-vsctl` as well as

   `ovs-dpctl` but is meant for OpenFlow switches, thus the `of` in the name

Let's continue to use the Layer 2 switch example as before and look at some examples of the command-line tools. First up, let's look at `ovs-vsctl` and `ovs-dpctl` for configuring and querying Open vSwitch:

```
$ ovs-vsctl -V
ovs-vsctl (Open vSwitch) 2.3.90
Compiled Jan 8 2015 11:52:49
DB Schema 7.11.1

$ sudo ovs-vsctl show
873c293e-912d-4067-82ad-d1116d2ad39f
    Bridge "s1"
        Controller "ptcp:6634"
        Controller "tcp:127.0.0.1:6633"
            is_connected: true
        fail_mode: secure
        Port "s1"
            Interface "s1"
                type: internal
 ...
```

You can also query a specific switch for more information:

```
ubuntu@sdnhubvm:~[12:43]$ sudo ovs-vsctl list-br
 s1

ubuntu@sdnhubvm:~[12:43]$ sudo ovs-vsctl list-ports s1
 s1-eth1
 s1-eth2
 s1-eth3

ubuntu@sdnhubvm:~[12:43]$ sudo ovs-vsctl list interface
 _uuid : 399a3edc-8f93-4984-a646-b54cb35cdc4c
 admin_state : up
 bfd : {}
 bfd_status : {}
 cfm_fault : []
 cfm_fault_status : []
 ...

ubuntu@sdnhubvm:~[12:44]$ sudo ovs-dpctl dump-flows
 flow-dump from pmd on cpu core: 32767
 recirc_id(0),in_port(2),eth(dst=00:00:00:00:00:02),eth_type(0x0800),ipv4(frag=no),
packets:0, bytes:0, used:never, actions:1
 recirc_id(0),in_port(1),eth(dst=00:00:00:00:00:01),eth_type(0x0800),ipv4(frag=no),
packets:1, bytes:98, used:2.316s, actions:2
```

By default, OVS supports OpenFlow 1.0 and 1.3. One of the useful things you can use `ovs-vsctl` to do, for example, is to force Open vSwitch to support OpenFlow 1.3 only (we will look at `ovs-ofctl` in a bit; here we are just using it to verify the OpenFlow version):

```
 $ sudo ovs-ofctl show s1
 OFPT_FEATURES_REPLY (xid=0x2): dpid:0000000000000001
 $ sudo ovs-vsctl set bridge s1 protocols=OpenFlow13
 $ sudo ovs-ofctl show s1
 2017-03-30T21:55:20Z|00001|vconn|WARN|unix:/var/run/openvswitch/s1.mgmt: version
negotiation failed (we support version 0x01, peer supports version 0x04)
```

As an alternative, you can also use the `ovs-ofctl` command-line tools for OpenFlow switches:

```
$ sudo ovs-ofctl show s1
OFPT_FEATURES_REPLY (xid=0x2): dpid:0000000000000001
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP
actions: output enqueue set_vlan_vid set_vlan_pcp strip_vlan mod_dl_src mod_dl_dst
mod_nw_src mod_nw_dst mod_nw_tos mod_tp_src
...

$ sudo ovs-ofctl dump-flows s1
 NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=790.374s, table=0, n_packets=4, n_bytes=280, idle_age=785,
priority=1,in_port=2,dl_dst=00:00:00:00:00:01 actions=output:1
 cookie=0x0, duration=790.372s, table=0, n_packets=3, n_bytes=238, idle_age=785,
priority=1,in_port=1,dl_dst=00:00:00:00:00:02 actions=output:2
 cookie=0x0, duration=1037.897s, table=0, n_packets=3, n_bytes=182, idle_age=790,
priority=0 actions=CONTROLLER:65535
```

As you can see from the output, the flows are unidirectional. Even though the communication happens bidirectionally (`h1` to `h2` and from `h2` back to `h1`), the flows are installed individually. There is one flow from h1 to h2, and another flow from `h2` to `h1`; together, they form a bidirectional communication. As network engineers, who sometimes take for granted that a forward path from a source to destination does not necessarily guarantee a return path in the case of OpenFlow or any other network application, bidirectional communication needs to be kept in mind.

Two other points to note about the flow output are as follows:

- `dpid` represents the identification of the switch. If we have multiple switches, they will be differentiated by `dpid`.
- Also note the mac address of the hosts. Because we use the `--mac` option, host 1 has mac address `00:00:00:00:00:01` and host 2 has mac address of `00:00:00:00:00:02`, which makes it very easy to identify the hosts at the layer 2 level:

```
 OFPT_FEATURES_REPLY (xid=0x2): dpid:0000000000000001
 recirc_id(0),in_port(2),eth(dst=00:00:00:00:00:02),eth_type(0x0800),ipv4(frag=no),
packets:0, bytes:0, used:never, actions:1
 recirc_id(0),in_port(1),eth(dst=00:00:00:00:00:01),eth_type(0x0800),ipv4(frag=no),
packets:1, bytes:98, used:2.316s, actions:2
```

The command-line tools are very useful when you need to configure the switch manually for both switch parameters as well as datapath, say, adding or deleting a flow manually. The analogy I would like to make is they are similar to out-of-band management of a physical switch. They are helpful when you need to either isolate the controller or the switch for troubleshooting. They are also useful when you are writing your own network application, as we will do later in this chapter, to verify that the controller is instructing the switch to install flows as expected.

We have seen the process of launching a typical Mininet network with a Layer 2 learning switch as well as command-line tools to verify switch operations. Now let's take a look at launching different applications via `ryu-manager`.

# The Ryu firewall application

As discussed earlier in the chapter, part of the attraction of OpenFlow is that the technology allows the controller to decide the functions of the switch. In other words, while the switch itself houses the flow, the controller decides the logic of matching fields and the flows to program the switch with. Let's see examples of how we can use the same Mininet topology, but now make the switch behave like a firewall. We will use one of the ready-made sample applications, called `rest_firewall.py`, as an example.

First, we will launch the Mininet network and set the switch version to OpenFlow 1.3:

```
$ sudo mn --topo single,3 --mac --controller remote --switch ovsk
$ sudo ovs-vsctl set bridge s1 protocols=OpenFlow13
```

Then under the `/home/ubuntu/ryu` directory, we will launch the firewall application with the `verbose` option. The `rest_firewall.py` file was part of the example applications that came with the Ryu source code. We will see the switch register to the controller:

```
$ ./bin/ryu-manager --verbose ryu/app/rest_firewall.py
...
creating context dpset
creating context wsgi
instantiating app ryu/app/rest_firewall.py of RestFirewallAPI
...
(3829) wsgi starting up on http://0.0.0.0:8080/
...
EVENT dpset->RestFirewallAPI EventDP
[FW][INFO] dpid=0000000000000001: Join as firewall.
```

By default, the switch is disabled. So we will enable the switch first:

```
$ curl -X GET http://localhost:8080/firewall/module/status
 [{"status": "disable", "switch_id": "0000000000000001"}]

$ curl -X PUT http://localhost:8080/firewall/module/enable/0000000000000001
 [{"switch_id": "0000000000000001", "command_result": {"result": "success",
"details": "firewall running."}}]
```

By default, when you try to ping from `h1` to `h2`, the packet is blocked:

```
mininet> h1 ping -c 2 h2
 PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
 --- 10.0.0.2 ping statistics ---
 2 packets transmitted, 0 received, 100% packet loss, time 1004ms
```

We can see this in the `verbose` output of the Ryu terminal window:

```
[FW][INFO] dpid=0000000000000001: Blocked packet =
ethernet(dst='00:00:00:00:00:02',ethertype=2048,src='00:00:00:00:00:01'),
ipv4(csum=56630,dst='10.0.0.2',flags=2,header_length=5,identification=18800,offset=

icmp(code=0,csum=58805,data=echo(data='xddx84xddXx00x00x00x00x84xc6x02x00x00x00x00x
 !"#$%&'()*+,-./01234567',id=4562,seq=1),type=8)
```

We can now add a rule to permit ICMP packets:

```
$ curl -X POST -d '{"nw_src": "10.0.0.1", "nw_proto": "ICMP"}'
http://localhost:8080/firewall/rules/0000000000000001

 [{"switch_id": "0000000000000001", "command_result": [{"result": "success",
"details": "Rule added. : rule_id=1"}]}]

$ curl -X POST -d '{"nw_src": "10.0.0.2", "nw_proto": "ICMP"}'
http://localhost:8080/firewall/rules/0000000000000001

 [{"switch_id": "0000000000000001", "command_result": [{"result": "success",
"details": "Rule added. : rule_id=2"}]}]
```

We can verify the rule set via the API:

```
$ curl -X GET http://localhost:8080/firewall/rules/0000000000000001
 [{"access_control_list": [{"rules": [{"priority": 1, "dl_type": "IPv4",
"nw_proto": "ICMP", "nw_src": "10.0.0.1", "rule_id": 1, "actions": "ALLOW"},
{"priority": 1, "dl_type": "IPv4", "nw_proto": "ICMP", "nw_src": "10.0.0.2",
"rule_id": 2, "actions": "ALLOW"}]}], "switch_id": "0000000000000001"}]
```

Now the Mininet `h1` host can ping `h2`:

```
mininet> h1 ping -c 2 h2
 PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
 64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.561 ms
 64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.319 ms

--- 10.0.0.2 ping statistics ---
 2 packets transmitted, 2 received, 0% packet loss, time 1000ms
 rtt min/avg/max/mdev = 0.319/0.440/0.561/0.121 ms
 mininet>
```

*For more Ryu firewall application APIs, consult the Ryu manual at https://github.com/osrg/ryu-book.*

I don't know about you, but the first time I saw this reference application, it really blew me away!  The reason it is impressive to me is because of the agility it provides. Note that in our example, we did not change the topology or the virtual switch. We were able to alter the virtual switch function from layer 3 switching to a firewall simply by changing the Ryu application. Imagine the amount of work it will take on vendor-based equipment: it would have taken a lot longer than we just did. If we have some brand new idea about a firewall, this is would be a great way to shorten the develop-test-trial cycle. To me, this example illustrates the flexibility and the power of OpenFlow and SDN.

We have seen how OpenFlow and SDN enable quick prototyping of ideas and features via software. In the next section, we will start to learn how we can write our own network application for the Ryu controller. We will start by looking closely at the Layer 2 switch application that we used before.

## Layer 2 OpenFlow switch

While we have lots of choices when it comes to learning and dissecting Ryu OpenFlow applications, we want to pick something simple that we have already worked with. By simple, I mean we want to pick something in which we know how the technology works so we can focus on just the new information, such as how Ryu uses Python to implement OpenFlow technologies. To that end, the Layer 2 OpenFlow switch application that we saw earlier would be a good candidate. Most of us network engineers know how switching works, especially in a single-switch setup (without spanning tree, which is not so simple).

As a reminder, if you are using the SDNHub virtual machine and are logged in as the default user Ubuntu, the application file is located under `/home/ubuntu/ryu/ryu/app/simple_switch_13.py`:

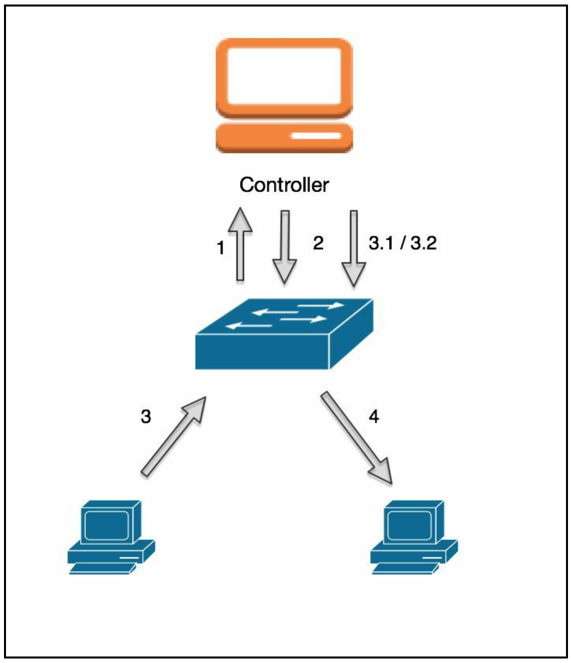
```
ubuntu@sdnhubvm:~[21:28]$ ls ryu/ryu/app/simple_switch_13.py
 ryu/ryu/app/simple_switch_13.py
```

The file is surprisingly short: fewer than 100 lines of code minus the comments and import statements. This is mainly due to the fact that we are able to abstract and reuse common Python codes such as `packet decode`, `controller`, and `ofprotocol`. Let's start by thinking about what a controller and switch should do in this scenario.

## Planning your application

Before you can write your application, you should at least have an outline of what a successful application would look like. It does not need to be very detailed; in fact, most people don't have all the details. But you should have at least a high-level outline of what you envision it to be. It is like writing an essay: you would need to know where to start, then illustrate your points, and know what the ending would be like. In our scenario, we know our application should perform the same as a switch, so we would envision it to do the following:

1. Upon initialization, the switch will negotiate its capabilities with the controller so that the controller will know which version of OpenFlow operations the switch can perform.
2. The controller will install a table miss flow action so that the switch will send any table miss flows to the controller and a `PacketIn` event for further instructions from the controller.
3. Upon receiving the `PacketIn` event, the controller will perform the following:
    1. Learn the MAC address connected to the port. Record the MAC address and port information.
    2. Look at the destination MAC address and see whether the host is already known. If it is a known host, use the `PacketOut` function on the port and install the flow for future packets. If the host is unknown, use the `PacketOut` function to flood to all ports.
4. This step is optional during initial application construction, but when in production, you want to set a timeout value associated with both the MAC address table as well as the flow. This is to conserve your resources as well as to avoid stale entries.



OpenFlow packet-processing pipeline

Now that we have a good idea of what is required of us, let's look at the stock application and its different components.

## Application components

In a typical development cycle, we will start to develop the application after we have planned out the requirements. However, since we are in learning mode and already know of a reference application, it would be smart of us to examine the `simple_switch_13.py` application as a starting point. Based on what we've seen, we can also tweak or make our own code based on this reference to help us understand the Ryu framework better.

Let's make a separate folder called `mastering_python_networking` under `/home/ubuntu/ryu` and store our file in there:

```
$ mkdir mastering_python_networking
```

In the folder, you can copy and paste the following code, from `chapter10_1.py`:

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3, ofproto_v1_0

class SimpleSwitch(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch, self).__init__(*args, **kwargs)
        self.mac_to_port = {}

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        print("ev message: ", ev.msg)
        datapath = ev.msg.datapath
        print("datapath: ", datapath)
        ofproto = datapath.ofproto
        print("ofprotocol: ", ofproto)
        parser = datapath.ofproto_parser
        print("parser: ", parser)
```

This file is very similar to the top portion of the reference application, with a few exceptions:

- Since Open vSwitch supports both OpenFlow 1.0 and 1.3, we purposely want to register the switch with OpenFlow 1.0

- We are using the `print` function to get the output of `ev`, `datapath`, `ofprotocol`, and `parser`

We can see from the Ryu API document, http://ryu.readthedocs.io/en/latest/, that a Ryu application is a Python module that defines a subclass of `ryu.base.app_manager.RyuApp`. Our `switch` class is a subclass of this base class. We also see the decorator of `ryu.controller.handler.set_ev_cls`, which is a decorator for the Ryu application to declare an event handler. The decorator takes two arguments: the first one indicates which event will invoke the function, and the second argument indicates the state of the switch. In our scenario, we are calling the function when there is an OpenFlow `Switch Features` event; `CONFI_DISPATCHER` means the switch is in the negotiation phase of *version is negotiated and the features-request message has been sent*. If you are curious about the different switch states, you can check out the documentation at http://ryu.readthedocs.io/en/latest/ryu_app_api.html#ryu-base-app-manager-ryuapp. Although our application does not do much at this time, it is nonetheless a complete application that we can run:

```
$ bin/ryu-manager --verbose mastering_python_networking/chapter10_1.py
```

When you run the application, you will notice that the switch version is now 1.0 (`0x1`) instead of version 1.3 (`0x4`) as well as the different objects that we printed out. This is helpful because it gives us the full path to the object if we want to look up the API documentation:

```
 ...
 switch features ev version: 0x1
 ...
 ('ev: ',
OFPSwitchFeatures(actions=4095,capabilities=199,datapath_id=1,n_buffers=256,n_table
{1: OFPPhyPort(port_no=1,hw_addr='92:27:5b:6f:97:88',name='s1-
eth1',config=0,state=0,curr=192,advertised=0,supported=0,peer=0), 2:
OFPPhyPort(port_no=2,hw_addr='f2:8d:a9:8d:49:be',name='s1-
eth2',config=0,state=0,curr=192,advertised=0,supported=0,peer=0), 3:
OFPPhyPort(port_no=3,hw_addr='ce:34:9a:7d:90:7f',name='s1-
eth3',config=0,state=0,curr=192,advertised=0,supported=0,peer=0), 65534:
OFPPhyPort(port_no=65534,hw_addr='ea:99:57:fc:56:4f',name='s1',config=1,state=1,cur
 ('datapath: ', <ryu.controller.controller.Datapath object at 0x7feebca5e290>)
 ('ofprotocol: ', <module 'ryu.ofproto.ofproto_v1_0' from
'/home/ubuntu/ryu/ryu/ofproto/ofproto_v1_0.pyc'>)
 ('parser: ', <module 'ryu.ofproto.ofproto_v1_0_parser' from
'/home/ubuntu/ryu/ryu/ofproto/ofproto_v1_0_parser.pyc'>)
```

*Refer to* http://ryu.readthedocs.io/en/latest/ *for more Ryu APIs.*

TIP

We can now take a look at handling `PacketIn` events to make a simple hub. The file can be copied from `chapter10_2.py` to the same directory. The top portion of the file is identical to the previous version without the print statements outside of the event message:

```
class SimpleHub(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION]
    #OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleHub, self).__init__(*args, **kwargs)

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        message = ev.msg
        print("message: ", message)
```

What is now different is the new function that is called when there is a `PacketIn` event:

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packet_in_handler(self, ev):
    msg = ev.msg
    print("ev message: ", ev.msg)
    datapath = msg.datapath
    ofproto = datapath.ofproto
    ofp_parser = datapath.ofproto_parser

    actions = [ofp_parser.OFPActionOutput(ofproto.OFPP_FLOOD)]
    out = ofp_parser.OFPPacketOut(
        datapath=datapath, buffer_id=msg.buffer_id, in_port=msg.in_port,
        actions=actions)
    datapath.send_msg(out)
```

We use the same `set_ev_cls`, but to specify by calling the function for a `PacketIn` event. We also specified the state of the switch after the feature negotiation is completed via `MAIN_DISPATCHER`. As we can see from the previous example, the event message typically contains the information that we are interested in; therefore, we are printing it out in this function as well. What we added is an OpenFlow version 1.0 parser, `OFPActionOutput` (http://ryu.readthedocs.io/en/latest/ofproto_v1_0_ref.html#action-structures), for flooding all the ports as well as a `PacketOut` specifying the flooding action.

If we launch the application, we will see the same switch feature message.

We can also ping from `h1` to `h2`:

```
mininet> h1 ping -c 2 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=4.89 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=3.84 ms

--- 10.0.0.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 3.842/4.366/4.891/0.528 ms
mininet>
```

In this case, we will see the broadcast packet coming in on port `1`:

```
EVENT ofp_event->SimpleHub EventOFPPacketIn
 ('ev message: ',
OFPPacketIn(buffer_id=256,data='xffxffxffxffxffxffx00x00x00x00x00x01x08x06x00x01x08
```

The broadcast packet will reach h2, and h2 will respond to the broadcast from port `2`:

```
EVENT ofp_event->SimpleHub EventOFPPacketIn
 ('ev message: ',
OFPPacketIn(buffer_id=257,data='x00x00x00x00x00x01x00x00x00x00x00x02x08x06x00x01x08
```

Note that unlike the simple switch application, we did not install any flows into the switch, so all the packets are flooded out to all ports, just like a hub. You may have also noticed a difference between our application and the `simple_switch_13.py` reference module. In the reference module, upon feature negotiation, the application installed a flow-miss flow to send all packets to the controller:

```
match = parser.OFPMatch()
actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                  ofproto.OFPCML_NO_BUFFER)]
self.add_flow(datapath, 0, match, actions)
```

However, we did not have to do that in our module in order to see future `PacketIn` events. This is due to the default behavior change between OpenFlow 1.0 to OpenFlow 1.3. In 1.0, the default table-miss action is to send the packet to the controller, whereas in 1.3, the default action is to drop the packet. Feel free to uncomment the `OFP_VERSIONS` variable form 1.0 to 1.3 and relaunch the application; you will see that the `PacketIn` event will not be sent to the controller:

```
class SimpleHub(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION]
    #OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
```

Let's look at the `add_flow()` function of the reference module:

```
def add_flow(self, datapath, priority, match, actions, buffer_id=None):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                         actions)]
    if buffer_id:
        mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
                                priority=priority, match=match,
                                instructions=inst)
    else:
        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                match=match, instructions=inst)
    datapath.send_msg(mod)
```

We have seen most of these fields; the only thing we have not seen is the `buffer_id`. This is part of the OpenFlow standard, in which the switch has the option of sending the whole packet including payload or choosing to park the packet at a buffer and only send a fraction of the packet header to the controller. In this case, the controller will send the flow insertion, including the buffer ID.

The reference module contains a more sophisticated `PacketIn` handler than our hub in order to make it a switch. There were two extra imports in order to decode the packet:

```
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
```

Within the `_packet_in_hander()`, we took stock of the `in_port`, and populated the `mac_to_port` dictionary by using the dpid of the switch and the MAC address:

```
in_port = msg.match['in_port']

pkt = packet.Packet(msg.data)
eth = pkt.get_protocols(ethernet.ethernet)[0]

dst = eth.dst
src = eth.src
```

The rest of the code takes care of the `PacketOut` event as well as inserting the flow using the `add_flow()` function.

```
# learn a mac address to avoid FLOOD next time.
self.mac_to_port[dpid][src] = in_port

if dst in self.mac_to_port[dpid]:
    out_port = self.mac_to_port[dpid][dst]
else:
    out_port = ofproto.OFPP_FLOOD

actions = [parser.OFPActionOutput(out_port)]

# install a flow to avoid packet_in next time
if out_port != ofproto.OFPP_FLOOD:
    match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
    # verify if we have a valid buffer_id, if yes avoid to send both
    # flow_mod & packet_out
    if msg.buffer_id != ofproto.OFP_NO_BUFFER:
        self.add_flow(datapath, 1, match, actions, msg.buffer_id)
        return
    else:
        self.add_flow(datapath, 1, match, actions)
data = None
if msg.buffer_id == ofproto.OFP_NO_BUFFER:
    data = msg.data

out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                          in_port=in_port, actions=actions, data=data)
datapath.send_msg(out)
```

You might be thinking at this point, "Wow, that is a lot of work just to get a simple switch to work!" and you would be correct in that regard. If you just want a switch, using OpenFlow and Ryu is absolutely overkill. You are better off spending a few bucks at the local electronics store than studying all the API documents and code. However, keep in mind that the modules and design patterns we have learned in this section are reused in almost any Ryu application. By learning these techniques, you are setting yourself up for bigger and better OpenFlow applications using the Ryu controller to enable rapid development and prototyping. I would love to hear about your next million-dollar network feature using OpenFlow and Ryu!

# The POX controller

The final section of Chapter 1, *Review of TCP/IP Protocol Suite and Python Language*, introduce to you another Python-based OpenFlow controller, POX. When OpenFlow was originally developed at Stanford, the original controller was NOX, written in Java. As an easier-to-learn alternative to NOX, POX was created. I personally used POX to learn OpenFlow a few years ago and thought it was an excellent learning tool. However, as time has passed, the development of POX seems to have slowed down a bit. For example, currently there is no active effort for Python 3. More importantly, officially, POX supports OpenFlow 1.0 and a number of Nicira extensions and has partial support for OpenFlow 1.1. Therefore, I have picked Ryu as the OpenFlow controller for this book. POX remains a viable alternative, however, for Python-based controllers if you are only working with OpenFlow 1.0.

TIP *You can learn more about POX on https://openflow.stanford.edu/display/ONL/POX+Wiki.*

Launching any of the applications in POX is similar to Ryu. On the SDNHub VM, POX is already installed under `/home/ubuntu/pox`. The `pox.py` module is equivalent to Ryu-manager. The controller will automatically look under the `pox/` folder for applications.

Change to the `/home/ubuntu/pox` directory, and launch the application under `pox/forwarding/tutorial_l2_hub`. It is really easy:

```
$ ./pox.py log.level --DEBUG forwarding.tutorial_l2_hub
POX 0.5.0 (eel) / Copyright 2011-2014 James McCauley, et al.
DEBUG:core:POX 0.5.0 (eel) going up...
DEBUG:core:Running on CPython (2.7.6/Oct 26 2016 20:30:19)
DEBUG:core:Platform is Linux-3.13.0-24-generic-x86_64-with-Ubuntu-14.04-trusty
INFO:core:POX 0.5.0 (eel) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
```

Testing with Mininet of course does not change it:

```
mininet> h1 ping -c 2 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=39.9 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=24.5 ms

--- 10.0.0.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 24.503/32.231/39.960/7.730 ms
```

We will not spend a lot of time learning about POX since our focus is on Ryu. POX has similar structures in regard to listening for OpenFlow events and launching callback functions as well as a packet parsing library.

## Summary

In this chapter, we covered a lot of topics related to OpenFlow. We learned about the basics and the history of the OpenFlow protocol. We were able to set up a lab with Mininet, which is a lightweight network-emulation tool that can create a full network with virtual Open vSwitch and Linux hosts. We also learned about the command-line tools that can interact directly with Open vSwitch for information gathering and troubleshooting.

We continued on to look at two Python-based OpenFlow controllers, Ryu and POX. The POX controller was a clone of the original NOX controller reference design written in Python. Due to its academic nature, among other reasons, the project has lagged behind on the latest OpenFlow features, such as OpenFlow 1.3. Ryu, on the other hand, was sponsored by NTT and benefited from the production usage and experience of the service provider. We focused on the Ryu controller components and looked at how we can easily switch between Ryu applications by pointing Ryu-manager to different Python files, such as L2 switches and L3 firewalls.

We looked at a full-featured firewall, written as a Ryu-based OpenFlow application. The firewall application is able to implement firewall rules via REST API using  the HTTP protocol, making it more automation-friendly than traditional vendor-based firewalls because we are able to make the change programmatically without the human-based command-line interface.

The switch application is a simple yet valuable application that we dissected and looked at in detail in order to understand the Ryu framework and design patterns.

In the next chapter, we will build on the knowledge we gained in this chapter and look at more advanced topics of OpenFlow.

# Advanced OpenFlow Topics

In the previous chapter, we looked at some of the basic concepts of OpenFlow, Mininet, and the Ryu controller. We proceeded to dissect and construct an OpenFlow switch in order to understand Ryu's framework as well as looking at one of Ryu's firewall applications as a reference for the API interface. For the same Mininet network, we were able to switch between a simple switch and a firewall by just replacing one software application with another. Ryu provides Python modules to abstract basic operations such as switch feature negotiation, packet parsing, OpenFlow message constructs, and many more. This allows us to focus on the networking logic of our application. Since Ryu was written in Python, the application code can be written in our familiar Python language as well.Â

Of course, our goal is to write our own applications. In this chapter, we will take what we have learned so far and build on top of that knowledge. In particular, we will look at how we can build several network applications, such as a **Border Gateway ProtocolÂ** (**BGP**)Â router. We will start by examining how we can decode OpenFlow events and packets via the Ryu framework. Then we will build the following network applications gradually from one to the next:Â

- OpenFlow operations with Ryu
- Packet inspection
- Static flow router
- Router with REST API
- BGP router
- Firewall

Let's get started!

# Setup

In this chapter, we will continue to use the same http://sdnhub.org/ all-in-one virtual machine from the last chapter. The Ryu install on the VM was a few releases behind the current one, so it was fine for a start, but we want to use the latest BGP speaker library and examples for this chapter. The latest version also supports OpenFlow 1.5 if you would like to experiment with it. Therefore, let's download the latest version and install from source:

```
$ mkdir ryu_latest
$ cd ryu_latest/
$ git clone https://github.com/osrg/ryu.git
$ cd ryu/
$ sudo python setup.py install

# Verification:

$ ls /usr/local/bin/ryu*
/usr/local/bin/ryu* /usr/local/bin/ryu-manager*
$ ryu-manager --version
ryu-manager 4.13
```

*Officially, Ryu supports Python 3.4 and beyond. It is my opinion that most of the existing applications and user communities still have been using Python 2.7. This directly translates to supportability and knowledge base; therefore, I have decided to stick with Python 2.7 when it comes to Ryu applications.*

As we have seen so far, the Ryu framework basically breaks down OpenFlow events and dispatches them to the right functions. The framework takes care of the basic operations, such as handling the communication with the switch and decoding the right OpenFlow protocol based on the version as well as providing the libraries to build and parse various protocol packets (such as BGP). In the next section, we will see how we can look at the specific event message contents so that we can further process them.

# OpenFlow operations with Ryu

To begin with, let's look at two basic components of Ryu code that we have seen in the switch application:

- `ryu.base.app_manager`: The component that centrally manages Ryu applications. `ryu.base.app_manager.RyuApp` is the base class that all Ryu applications inherit. It provides contexts to Ryu applications and routes messages among Ryu applications (contexts are ordinary Python objects shared among Ryu applications). It also contains methods such as `send_event` or `send_event_to_observers` to raise OpenFlow events.
- `ryu.controller`: The `ryu.controller.controller` handles connections from switches and routes events to appropriate entities:
    - `ryu.controller.ofp_event`: This is the OpenFlow event definition.
    - `ryu.controller.ofp_handler`: This is the handler of an OpenFlow event. A Ryu application registers itself to listen for specific events using `ryu.controller.handler.set_ev_cls` decorator. We have seen the decorator `set_ev_cls` as well as `CONFIG_DISPATCHER` and `MAIN_DISPATCHER` from the handler.

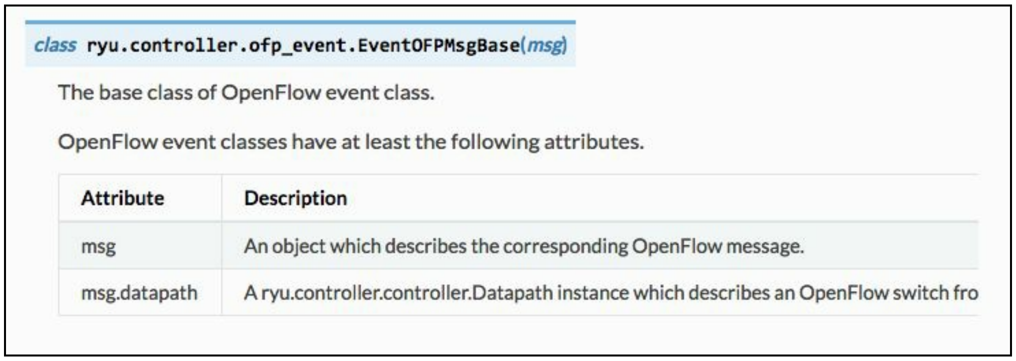
Each Ryu application has an event-received queue that is first-in, first-out with the event order preserved. Ryu applications are single-threaded, and the thread keeps draining the received queue by dequeueing and calling the appropriate event handler for different event types. We have seen two examples of it so far in the `simple_switch_13.py` application. The first was the handling of `ryu.controller.ofp_event.EventOFSwitchFeatures` for switch feature negotiation from the switch:

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
    datapath = ev.msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
...
```

The second example was `ryu.controller.ofp_event.EventOFPacketIn`. A `PacketIn` event is what the switch sends to the controller when there is a flow miss:

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    if ev.msg.msg_len < ev.msg.total_len:
        self.logger.debug("packet truncated: only %s of %s bytes",
                          ev.msg.msg_len, ev.msg.total_len)
...
```

The two examples show the conventions used in Ryu for OpenFlow events; they are named `ryu.controller.ofp_event.EventOF<name>`, where `<name>` is the OpenFlow message. The Ryu controller automatically decodes the message and sends it to the registered function. There are two common attributes for the `ryu.controller.opf_event.EVentOFMsgBase` class:


*class* **ryu.controller.ofp_event.EventOFPMsgBase**(*msg*)

The base class of OpenFlow event class.

OpenFlow event classes have at least the following attributes.

| Attribute | Description |
|---|---|
| msg | An object which describes the corresponding OpenFlow message. |
| msg.datapath | A ryu.controller.controller.Datapath instance which describes an OpenFlow switch fro |

OpenFlow event message base (source: http://ryu-ippouzumi.readthedocs.io/en/latest/ryu_app_api.html#openflow-event-classes)

The function of the event will take the event object as input and examine the `msg` and `msg.datapath` attributes. For example, in `chapter11_1.py`, we stripped out all the code for `ryu.controller.ofp_event.EventOFPSwitchFeatures` and decoded it from the event object:

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3

class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}

    # Catching Switch Features Events
    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):

        print("EventOFPSwitchFeatures: datapath_id {}, n_buffers {} capabilities
{}".format(ev.msg.datapath_id,
                        ev.msg.n_buffers,
                        ev.msg.capabilities))
```

This will print out the following message for the switch feature received:

```
EventOFPSwitchFeatures: datapath_id 1, n_buffers 256 capabilities 79
```

As you become more familiar with the Ryu framework, the API document will become a much-read reference: http://ryu-ippouzumi.readthedocs.io/en/latest/ofproto_ref.html#ofproto-ref.

Because switch registration is such as common operation, the Ryu controller typically takes care of the handshake of feature request and reply. It is probably more relevant for us to take a look at the parsing packet in a PacketIn event. This is what we will be doing in the next section.

## Packet inspection

In this section, we will take a closer look at the PacketIn message and parse the packet so we can further process the event and determine the corresponding action. In `chapter11_2.py`, we can use the `ryu.lib.packet.packet` method to decode the packet:

```
from ryu.lib.packet import packet
import array
...
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    print("msg.data: {}".format(array.array('B', ev.msg.data)))
    pkt = packet.Packet(ev.msg.data)
    for p in pkt.protocols:
        print(p.protocol_name, p)
```

Note that `ev.msg.data` is in bytearray format, which is why use the `array.array()` function to print it out. We can then initiate a ping, `h1 ping -c 2 h2`, and observe in the output the arp from `h1` to `h2`:

```
EVENT ofp_event->SimpleSwitch13 EventOFPPacketIn
 msg.data: array('B', [255, 255, 255, 255, 255, 255, 0, 0, 0, 0, 0, 1, 8, 6, 0, 1,
8, 0, 6, 4, 0, 1, 0, 0, 0, 0, 0, 1, 10, 0, 0, 1, 0, 0, 0, 0, 0, 0, 10, 0, 0, 2])
 ('ethernet',
ethernet(dst='ff:ff:ff:ff:ff:ff',ethertype=2054,src='00:00:00:00:00:01'))
 ('arp',
arp(dst_ip='10.0.0.2',dst_mac='00:00:00:00:00:00',hlen=6,hwtype=1,opcode=1,plen=4,p
```

If we add in the rest of the switch code, you can see the rest of the ARP response from `h2` and the subsequent successful pings from `h1` to `h2`:

```
EVENT ofp_event->SimpleSwitch13 EventOFPPacketIn
 msg.data: array('B', [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 2, 8, 6, 0, 1, 8, 0, 6, 4,
0, 2, 0, 0, 0, 0, 0, 2, 10, 0, 0, 2, 0, 0, 0, 0, 0, 1, 10, 0, 0, 1])
 ('ethernet',
ethernet(dst='00:00:00:00:00:01',ethertype=2054,src='00:00:00:00:00:02'))
 ('arp',
arp(dst_ip='10.0.0.1',dst_mac='00:00:00:00:00:01',hlen=6,hwtype=1,opcode=2,plen=4,p
('ethernet',
ethernet(dst='00:00:00:00:00:02',ethertype=2048,src='00:00:00:00:00:01'))
 ('ipv4',
ipv4(csum=56624,dst='10.0.0.2',flags=2,header_length=5,identification=18806,offset=
 ('icmp', icmp(code=0,csum=21806,data=echo(data=array('B', [30, 216, 230, 88, 0, 0,
0, 0, 125, 173, 9, 0, 0, 0, 0, 0, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27,
28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,
49, 50, 51, 52, 53, 54, 55]),id=22559,seq=1),type=8))
```

If we were to restart Mininet (or simply delete the installed flows), we could test out an HTTP flow with a Python HTTP server from the standard library. Simply start the HTTP server on `h2`, and use `curl` to get the webpage. By default, the Python HTTP server listens on port `8000`:

```
mininet> h2 python -m SimpleHTTPServer &
mininet> h1 curl http://10.0.0.2:8000
```

The observed result is as expected:

```
('ipv4',
ipv4(csum=48411,dst='10.0.0.2',flags=2,header_length=5,identification=27038,offset=
 ('tcp', tcp(ack=0,bits=2,csum=19098,dst_port=8000,offset=10,option=
[TCPOptionMaximumSegmentSize(kind=2,length=4,max_seg_size=1460),
TCPOptionSACKPermitted(kind=4,length=2),
TCPOptionTimestamps(kind=8,length=10,ts_ecr=0,ts_val=8596478),
TCPOptionNoOperation(kind=1,length=1),
TCPOptionWindowScale(kind=3,length=3,shift_cnt=9)],seq=1316962912,src_port=39600,ur
```

You could further decode the packet, for example, the IPv4 source and destination addresses:

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):

    print("msg.data: {}".format(array.array('B', ev.msg.data)))
    pkt = packet.Packet(ev.msg.data)
    for p in pkt.protocols:
        print(p.protocol_name, p)
        if p.protocol_name == 'ipv4':
            print('IP: src {} dst {}'.format(p.src, p.dst))
```

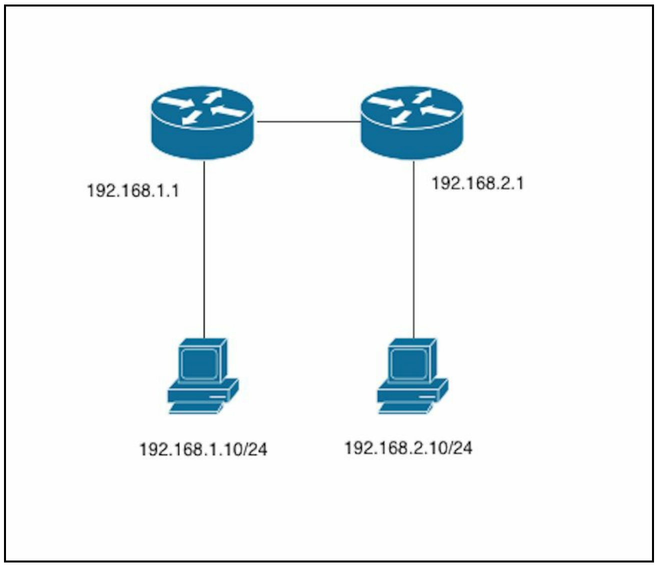
You will see the print-out of the HTTP curl from `h1` to `h2` as follows:

```
('ethernet',
ethernet(dst='00:00:00:00:00:02',ethertype=2048,src='00:00:00:00:00:01'))('ipv4',
ipv4(csum=56064,dst='10.0.0.2',flags=2,header_length=5,identification=19385,offset=
IP: src 10.0.0.1 dst 10.0.0.2
```

By looking closely at how the Ryu framework decodes/encodes OpenFlow messages and events, we are closer to building our own applications.

# Static router

Let us start by building a multi-device network statically. By static, I mean we will program static flows in the routers to make our topology work, which is similar to a statically routed device. Obviously, building something by hand is not very scalable, just like we do not use static routes in a production environment, but it is a great way to learn how to program a Ryu-based OpenFlow router. Note that I will use the terms *switch* and *router* interchangeably. In Mininet, the devices are by default switches with `s1`, `s2`, and so on; however, since we are performing routing functions via software and flow, they are also routers in my opinion.
Here is the simple topology that we will build:



Static router topology

Let's see how we can build this topology with Mininet.

# Mininet topology

Mininet provides a convenient way for building multi-device linear topologies with the `--topo linear` option:

```
$ sudo mn --topo linear,2 --mac --controller remote --switch ovsk
...
mininet> links
h1-eth0<->s1-eth1 (OK OK)
h2-eth0<->s2-eth1 (OK OK)
s2-eth2<->s1-eth2 (OK OK)
```

By default, the hosts were assigned the same IP addresses sequentially, similar to a single-switch topology:

```
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=29367>
<Host h2: h2-eth0:10.0.0.2 pid=29375>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None pid=29383>
<OVSSwitch s2: lo:127.0.0.1,s2-eth1:None,s2-eth2:None pid=29386>
<RemoteController c0: 127.0.0.1:6633 pid=29361>
```

We can easily change the IP addresses of the hosts according to our diagram:

```
mininet> h1 ip addr del 10.0.0.1/8 dev h1-eth0
mininet> h1 ip addr add 192.168.1.10/24 dev h1-eth0
mininet> h2 ip addr del 10.0.0.2/8 dev h2-eth0
mininet> h2 ip addr add 192.168.2.10/24 dev h2-eth0
mininet> h1 ip route add default via 192.168.1.1
mininet> h2 ip route add default via 192.168.2.1
```

We can verify the configuration as follows:

```
 mininet> h1 ifconfig | grep inet
 inet addr:192.168.1.10 Bcast:0.0.0.0 Mask:255.255.255.0
 inet6 addr: fe80::200:ff:fe00:1/64 Scope:Link
 inet addr:127.0.0.1 Mask:255.0.0.0
 inet6 addr: ::1/128 Scope:Host
...
 mininet> h2 ifconfig | grep inet
 inet addr:192.168.2.10 Bcast:0.0.0.0 Mask:255.255.255.0
 inet6 addr: fe80::200:ff:fe00:2/64 Scope:Link
 inet addr:127.0.0.1 Mask:255.0.0.0
 inet6 addr: ::1/128 Scope:Host
...
 mininet> h1 route -n
 Kernel IP routing table
```

```
Destination Gateway Genmask Flags Metric Ref Use Iface
0.0.0.0 192.168.1.1 0.0.0.0 UG 0 0 0 h1-eth0
192.168.1.0 0.0.0.0 255.255.255.0 U 0 0 0 h1-eth0
...
mininet> h2 route -n
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
0.0.0.0 192.168.2.1 0.0.0.0 UG 0 0 0 h2-eth0
192.168.2.0 0.0.0.0 255.255.255.0 U 0 0 0 h2-eth0
mininet>
```

Great! We've built the topology we needed in less than 5 minutes. Keep these steps handy, as we will start and restart Mininet a few times during our build. Let's take a look at the code that we will be using.

## Ryu controller code

Most of us network engineers take routing for granted after a while. After all, it is easy to just assign the default gateway for devices, turn on the routing protocol on routers, advertise IP blocks, and call it a day. However, as we have seen in the switch configuration, OpenFlow gives you very granular control over network operations. With this control, you have to tell the devices what to do with your code: very little is handled on your behalf. So let's take a step back and think about the network operations in our two-router topology when `h1` is trying to ping `h2`, such as when issuing the command `h1 ping -c 2 h2`:

- The `h1` host with IP address `192.168.1.10/24` knows that the destination `192.168.2.10` is not on its own subnet. Therefore, it will need to send the packet to its default gateway, `192.168.1.1`.
- The `h1` host initially does not know how to reach `192.168.1.1` on the same subnet, so it will send out an ARP request for `192.168.1.1`.
- After receiving the ARP response, `h1` will encapsulate the `ICMP` packet with IPv4 source `192.168.1.10`, IPv4 destination `192.168.2.10`, source mac `00:00:00:00:00:01`, and the destination mac from the ARP response.
- If router 1 does not have a flow for the receiving packet, assuming we have installed a flow miss entry for the controller, the packet will be sent to the controller for further action.
- The controller will instruct router 1 to send the packet out to the correct port, in this case port 2, and optionally install a flow entry for future flow table match.
- The steps are repeated until the packet reaches `h2`, and then the process is reversed, that is, `h2` will ARP for its default gateway, then send out the packet, and so on.

The code is provided for you in `Chapter11_3.py`. As with the rest of the code in the book, the code is written to demonstrate concepts and features, so some implementation might not make sense in real-world production. This is especially true when it comes to OpenFlow and Ryu because of the freedom it offers you. First, let's take a look at the static flow installs from the controller.

# Ryu flow installation

At the top of the code, you will notice we have a few more imports that we need further down the file:

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types

#new import
from ryu.ofproto import ether
from ryu.lib.packet import ipv4, arp
```

During the class initialization, we will determine what MAC addresses we will use for `s1` and `s2`:

```
class MySimpleStaticRouter(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(MySimpleStaticRouter, self).__init__(*args, **kwargs)
        self.s1_gateway_mac = '00:00:00:00:00:02'
        self.s2_gateway_mac = '00:00:00:00:00:01'
```

We can pick any valid formatted MAC address we want for the ARP, but you will notice that we have used the same MAC address for the gateway to spoof the host mac. In other words, the `s1` gateway has the `h2` host's MAC address, while the `s2` gateway has the `h1` host's MAC address. This is done intentionally; we will look at the reason later on. In the next section, we see the same `EventOFPSwitchFeatures` event for the decorator and the corresponding function.

After installing the flow miss table entry, we see that we have pushed out a number of static flows to the two devices:

```
if datapath.id == 1:
    # flow from h1 to h2
    match = parser.OFPMatch(in_port=1,
                              eth_type=ether.ETH_TYPE_IP,
                              ipv4_src=('192.168.1.0', '255.255.255.0'),
                              ipv4_dst=('192.168.2.0', '255.255.255.0'))
    out_port = 2
    actions = [parser.OFPActionOutput(out_port)]
    self.add_flow(datapath, 1, match, actions)
...
```

A few points to note in this section:

- We need to match the device for the flow to be pushed out by identifying the `datapath.id`, which corresponds to the DPID of each switch.
- We use several match types, such as `in_port`, `ether_type`, `ipv4_src`, and `ipv4_dst`, to make the flow unique. You can match as many fields as makes sense. Each OpenFlow specification adds more and more match fields. The module `ryu.lib.packet.ether_types` offers a convenient way to match Ethernet types, such as **Link Layer Discovery Protocol (LLDP),** ARP, Dot1Q trunks, IP, and much more.
- The flow priority is pushed out as priority 1. Recall that the flow miss entry is priority 0: since these entries will be pushed out after the flow miss entry, we need to put them at a higher priority to be matched. Otherwise, at the same priority, the flow entries are just matched sequentially.
- Optionally, you can comment out the `add_flow` entries in order to see the packet at the controller level. For example, you can comment out the `datapath.id 1` for the `h1` to `h2` flow in order to see the ARP entry as well as the first `ICMP` packet from `h1` to `h2` at the controller. Note that since these flows are installed upon switch feature handshake, if you comment out any flow install, you will need to restart Mininet and repeat the process in the Mininet section.

We have installed four flows; each corresponds to when the packet arrives from the hosts to the switches and between the switches. Remember that flows are unidirectional, so a flow is needed for both forward and return paths. In the `add_flow()` function, we also added the `idle_timeout` and `hard_timeout` options. These are commonly used options to avoid stale flow entries in the switch. The values are in seconds, so after 600 seconds, if the switch has not seen any flow miss or matching flow entries, the entries will be removed. A more realistic option would be to keep the flow miss entry permanent and dynamically install the other entries upon the corresponding `PacketIn` event. But again, we are making things simple for now:

```
def add_flow(self, datapath, priority, match, actions, buffer_id=None):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
...
    if buffer_id:
        mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
                                priority=priority, match=match,
                                instructions=inst, idle_timeout=600,
                                hard_timeout=600)
    else:
        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                match=match, instructions=inst,
                                idle_timeout=600, hard_timeout=600)
    datapath.send_msg(mod)
```

We can verify the flows installed on `s1` and `s2` using `ovs-ofctl`:

```
ubuntu@sdnhubvm:~[08:35]$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=1.694s, table=0, n_packets=0, n_bytes=0, idle_timeout=600,
hard_timeout=600, idle_age=20,
priority=1,ip,in_port=1,nw_src=192.168.1.0/24,nw_dst=192.168.2.0/24
actions=output:2
 cookie=0x0, duration=1.694s, table=0, n_packets=0, n_bytes=0, idle_timeout=600,
hard_timeout=600, idle_age=20,
priority=1,ip,in_port=2,nw_src=192.168.2.0/24,nw_dst=192.168.1.0/24
actions=output:1
 cookie=0x0, duration=1.694s, table=0, n_packets=13, n_bytes=1026,
idle_timeout=600, hard_timeout=600, idle_age=10, priority=0
actions=CONTROLLER:65535
 ...
ubuntu@sdnhubvm:~[08:36]$ sudo ovs-ofctl dump-flows s2
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=5.770s, table=0, n_packets=0, n_bytes=0,  idle_timeout=600,
hard_timeout=600, idle_age=24,
priority=1,ip,in_port=2,nw_src=192.168.1.0/24,nw_dst=192.168.2.0/24
actions=output:1
 cookie=0x0, duration=5.770s, table=0, n_packets=0, n_bytes=0, idle_timeout=600,
hard_timeout=600, idle_age=24,
priority=1,ip,in_port=1,nw_src=192.168.2.0/24,nw_dst=192.168.1.0/24
actions=output:2
 cookie=0x0, duration=5.770s, table=0, n_packets=13, n_bytes=1038,
idle_timeout=600, hard_timeout=600, idle_age=14, priority=0
actions=CONTROLLER:65535
```

Now that we have taken care of the static flows, we need to respond to the ARP requests from the hosts. We will complete that process in the next section of the code.

# Ryu packet generation

In order to respond to the ARP requests, we need to do the following:

- Catch the ARP request based on the right switch, or datapath, and know the IP it is sending an ARP for
- Build the ARP response packet
- Send the response back out the port it was originally received from

In initial part of the `PacketIn` function, most of the code is identical to the `switch` function, which corresponds to decoding the OpenFlow event as well as the packet itself. For our purposes, we do not need `ofproto` and `parser` for OpenFlow events, but for consistency, we are leaving them in:

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']


    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]
```

The next section catches ARP packets by identifying the right `ether_type`:

```
if eth.ethertype == ether_types.ETH_TYPE_ARP:
    arp_packet = pkt.get_protocols(arp.arp)[0]
    ethernet_src = eth.src
```

We can then identify the device and the IP it is sending an ARP for and continue on to build the packet. The packet-building process is similar to the tools we have seen before, such as Scapy, in that it took a layered approach where you can build the Ethernet header and then the ARP packet itself. We will construct a packet via `ryu.lib.packet.Packet()`, add the contents in, and serialize the packet:

```
if arp_packet.dst_ip == '192.168.2.1' and datapath.id == 2:
    print('Received ARP for 192.168.2.1')

    e = ethernet.ethernet(dst=eth.src, src=self.s2_gateway_mac,
ethertype=ether.ETH_TYPE_ARP)
    a = arp.arp(hwtype=1, proto=0x0800, hlen=6, plen=4, opcode=2,
                src_mac=self.s2_gateway_mac, src_ip='192.168.2.1',
                dst_mac=ethernet_src, dst_ip=arp_packet.src_ip)

    p = packet.Packet()
    p.add_protocol(e)
    p.add_protocol(a)
    p.serialize()
...
```

The last step would be to send the packet out to the port it was received from:

```
outPort = in_port
actions = [datapath.ofproto_parser.OFPActionOutput(outPort, 0)]
out = datapath.ofproto_parser.OFPPacketOut(
    datapath=datapath,
    buffer_id=0xffffffff,
    in_port=datapath.ofproto.OFPP_CONTROLLER,
    actions=actions,
    data=p.data)
datapath.send_msg(out)
```

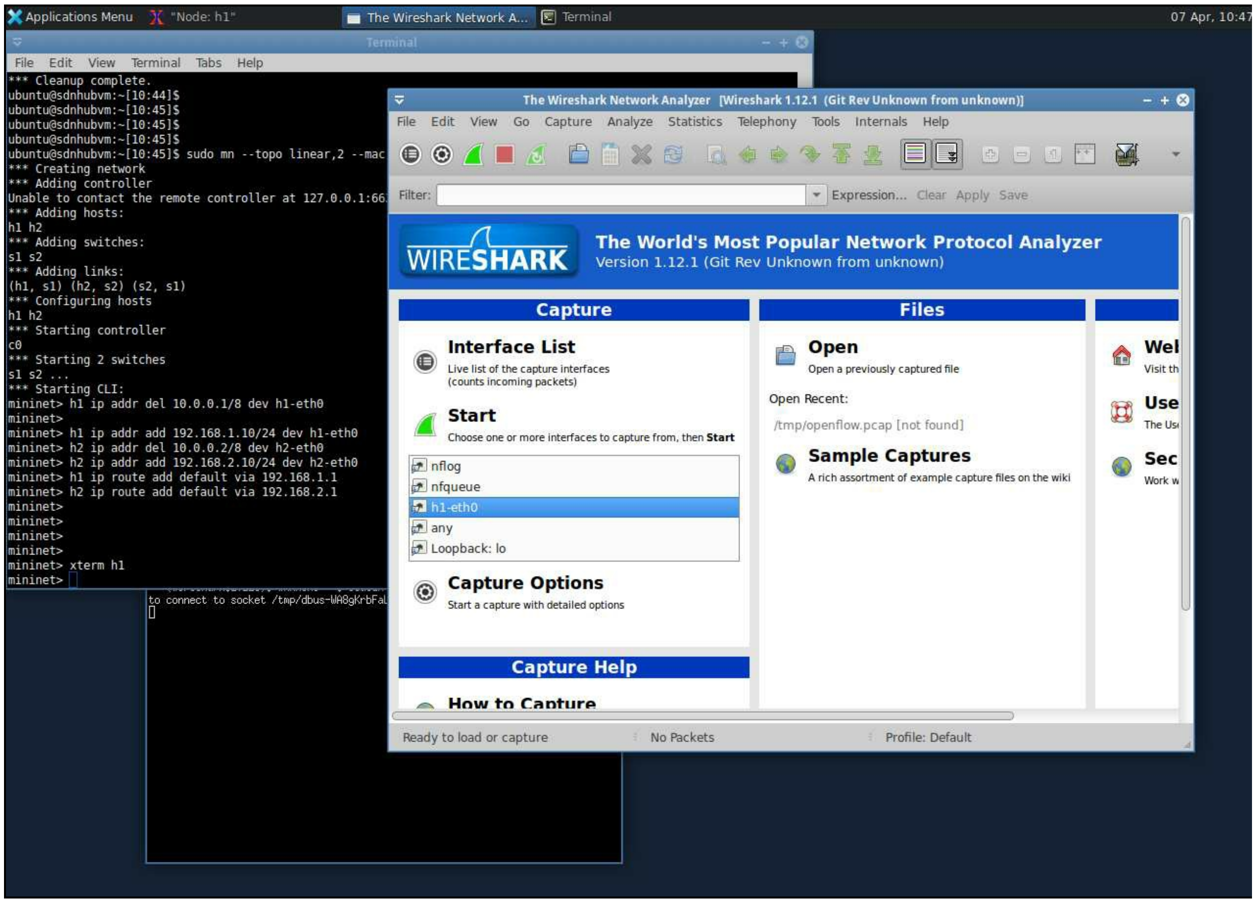
Notice that for the `in_port`, we use the special `OFPP_CONTROLLER` since we don't technically have an inbound port. We will repeat the ARP process for both gateways. The only additional code we have is to verbosely print out the packet we have received and its meta information:

```
# verbose iteration of packets
try:
    for p in pkt.protocols:
        print(p.protocol_name, p)
    print("datapath: {} in_port: {}".format(datapath.id, in_port))
except:
    pass
```

We are now ready to check on the final result of our code.

## Final result

To verify, we will launch Mininet on the virtual machine desktop so we can use xterm on `h1` and launch Wireshark within the host to see the arp reply. After launching Mininet, simply type in `xterm h1` to launch a terminal window for `h1`, and type in `wireshark` at the `h1` terminal window to launch Wireshark. Choose h1-eth0 as the interface to capture packets on:



```
*** Cleanup complete.
ubuntu@sdnhubvm:~[10:44]$
ubuntu@sdnhubvm:~[10:45]$
ubuntu@sdnhubvm:~[10:45]$
ubuntu@sdnhubvm:~[10:45]$
ubuntu@sdnhubvm:~[10:45]$ sudo mn --topo linear,2 --mac
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:66
*** Adding hosts:
h1 h2
*** Adding switches:
s1 s2
*** Adding links:
(h1, s1) (h2, s2) (s2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 2 switches
s1 s2 ...
*** Starting CLI:
mininet> h1 ip addr del 10.0.0.1/8 dev h1-eth0
mininet>
mininet> h1 ip addr add 192.168.1.10/24 dev h1-eth0
mininet> h2 ip addr del 10.0.0.2/8 dev h2-eth0
mininet> h2 ip addr add 192.168.2.10/24 dev h2-eth0
mininet> h1 ip route add default via 192.168.1.1
mininet> h2 ip route add default via 192.168.2.1
mininet>
mininet>
mininet>
mininet>
mininet> xterm h1
mininet>
```

```
to connect to socket /tmp/dbus-WA8gKrbFaL
```

Applications Menu
"Node: h1"
The Wireshark Network A...
Terminal
07 Apr, 10:47
The Wireshark Network Analyzer [Wireshark 1.12.1 (Git Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save
WIRESHARK
The World's Most Popular Network Protocol Analyzer
Version 1.12.1 (Git Rev Unknown from unknown)
Capture
Interface List
Live list of the capture interfaces (counts incoming packets)
Start
Choose one or more interfaces to capture from, then Start
nflog
nfqueue
h1-eth0
any
Loopback: lo
Capture Options
Start a capture with detailed options
Capture Help
How to Capture
Files
Open
Open a previously captured file
Open Recent:
/tmp/openflow.pcap [not found]
Sample Captures
A rich assortment of example capture files on the wiki
Ready to load or capture
No Packets
Profile: Default

Mininet and h1 Wireshark

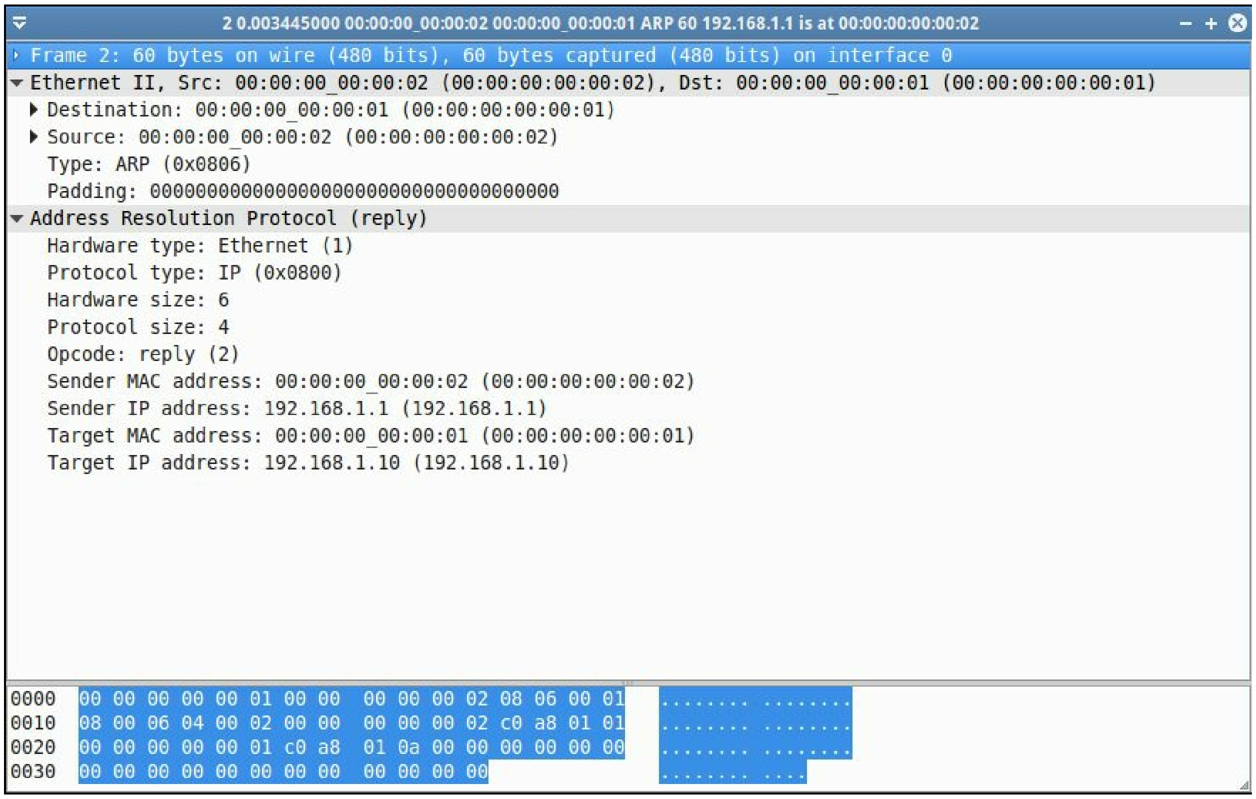
We can try to ping from `h1` to `h2` with two packets:

```
 mininet> h1 ping -c 2 h2
 PING 192.168.2.10 (192.168.2.10) 56(84) bytes of data.
 From 192.168.1.10 icmp_seq=1 Destination Host Unreachable
 From 192.168.1.10 icmp_seq=2 Destination Host Unreachable

--- 192.168.2.10 ping statistics ---
 2 packets transmitted, 0 received, +2 errors, 100% packet loss, time 1004ms
 pipe 2
 mininet>
```

On the controller screen, we can see the two ARP entries, one from `h1` for `192.168.1.1` and the other for `h2` from the gateways:

```
 EVENT ofp_event->MySimpleStaticRouter EventOFPPacketIn
 Received ARP for 192.168.1.1
  ('ethernet',
ethernet(dst='ff:ff:ff:ff:ff:ff',ethertype=2054,src='00:00:00:00:00:01'))
  ('arp',
arp(dst_ip='192.168.1.1',dst_mac='00:00:00:00:00:00',hlen=6,hwtype=1,opcode=1,plen=
 datapath: 1 in_port: 1
 EVENT ofp_event->MySimpleStaticRouter EventOFPPacketIn
 Received ARP for 192.168.2.1
 ('ethernet',
ethernet(dst='ff:ff:ff:ff:ff:ff',ethertype=2054,src='00:00:00:00:00:02'))
 ('arp',
arp(dst_ip='192.168.2.1',dst_mac='00:00:00:00:00:00',hlen=6,hwtype=1,opcode=1,plen=
 datapath: 2 in_port: 1
```

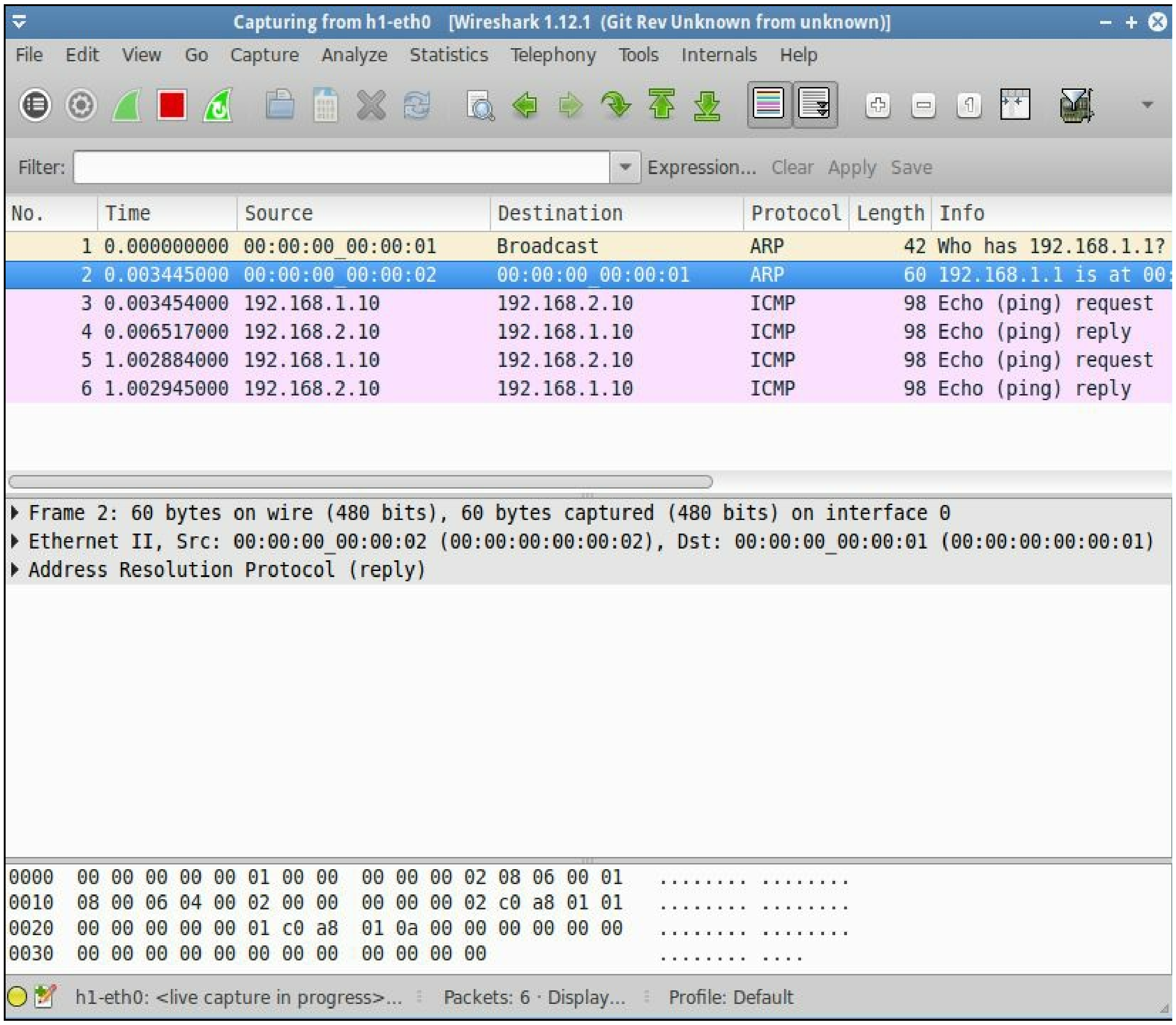
In the `h1-eth0` packet capture, we can see the arp reply:



```
2 0.003445000 00:00:00_00:00:02 00:00:00_00:00:01 ARP 60 192.168.1.1 is at 00:00:00:00:00:02
▸ Frame 2: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
▾ Ethernet II, Src: 00:00:00_00:00:02 (00:00:00:00:00:02), Dst: 00:00:00_00:00:01 (00:00:00:00:00:01)
  ▸ Destination: 00:00:00_00:00:01 (00:00:00:00:00:01)
  ▸ Source: 00:00:00_00:00:02 (00:00:00:00:00:02)
    Type: ARP (0x0806)
    Padding: 000000000000000000000000000000000000
▾ Address Resolution Protocol (reply)
    Hardware type: Ethernet (1)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reply (2)
    Sender MAC address: 00:00:00_00:00:02 (00:00:00:00:00:02)
    Sender IP address: 192.168.1.1 (192.168.1.1)
    Target MAC address: 00:00:00_00:00:01 (00:00:00:00:00:01)
    Target IP address: 192.168.1.10 (192.168.1.10)

0000  00 00 00 00 00 01 00 00  00 00 00 02 08 06 00 01   ........ ........
0010  08 00 06 04 00 02 00 00  00 00 00 02 c0 a8 01 01   ........ ........
0020  00 00 00 00 00 01 c0 a8  01 0a 00 00 00 00 00 00   ........ ........
0030  00 00 00 00 00 00 00 00  00 00 00 00               ........ ....
```

S1 Gateway ARP Reply

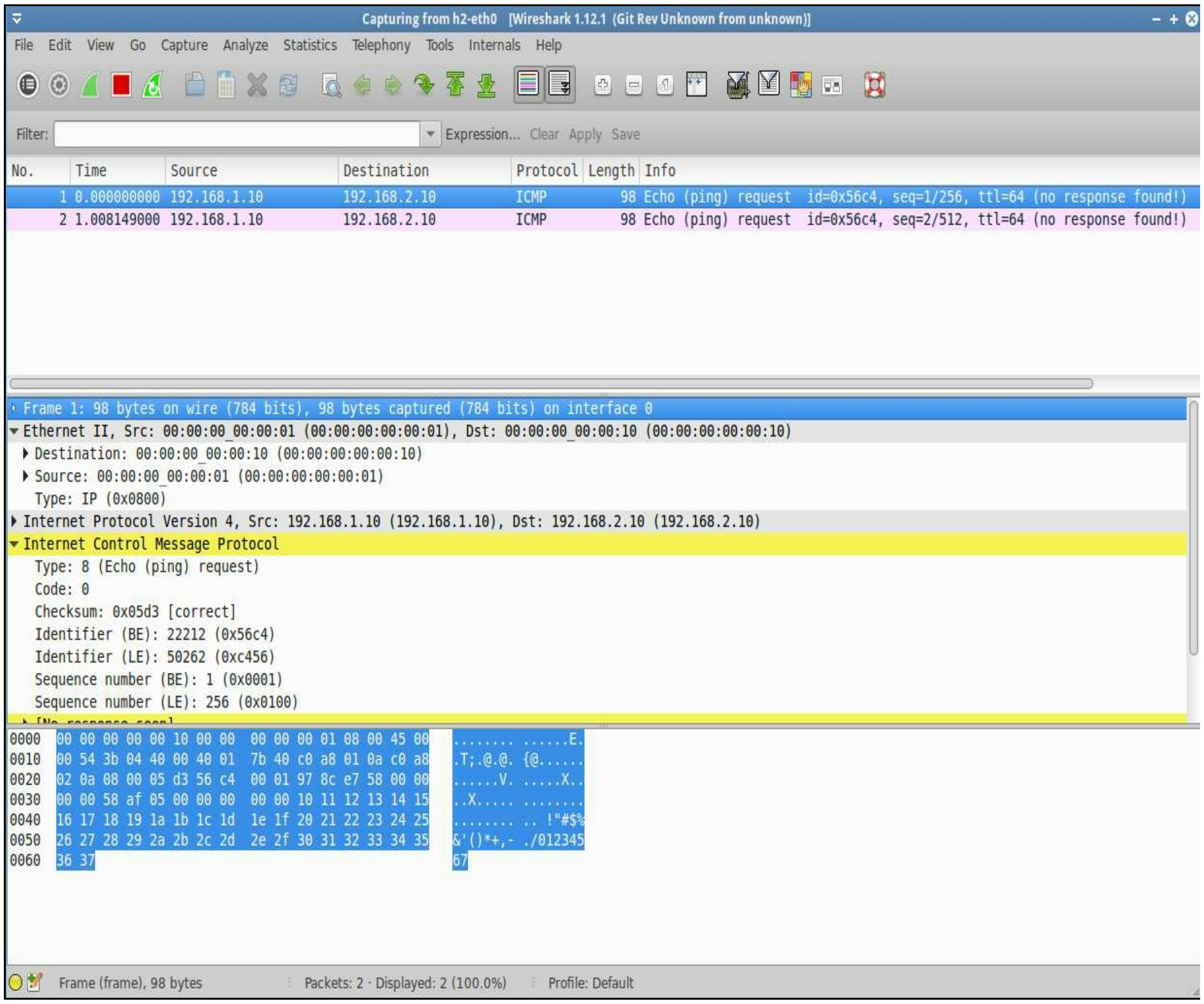
We can also see the ICMP request and reply from `h1` to `h2`:


Capturing from h1-eth0 [Wireshark 1.12.1 (Git Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

```
No.   Time         Source              Destination         Protocol Length Info
    1 0.000000000  00:00:00_00:00:01   Broadcast           ARP          42 Who has 192.168.1.1?
    2 0.003445000  00:00:00_00:00:02   00:00:00_00:00:01   ARP          60 192.168.1.1 is at 00:
    3 0.003454000  192.168.1.10        192.168.2.10        ICMP         98 Echo (ping) request
    4 0.006517000  192.168.2.10        192.168.1.10        ICMP         98 Echo (ping) reply
    5 1.002884000  192.168.1.10        192.168.2.10        ICMP         98 Echo (ping) request
    6 1.002945000  192.168.2.10        192.168.1.10        ICMP         98 Echo (ping) reply

▸ Frame 2: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:02 (00:00:00:00:00:02), Dst: 00:00:00_00:00:01 (00:00:00:00:00:01)
▸ Address Resolution Protocol (reply)

0000  00 00 00 00 00 01 00 00  00 00 00 02 08 06 00 01   ........ ........
0010  08 00 06 04 00 02 00 00  00 00 00 02 c0 a8 01 01   ........ ........
0020  00 00 00 00 00 01 c0 a8  01 0a 00 00 00 00 00 00   ........ ........
0030  00 00 00 00 00 00 00 00  00 00 00 00               ........ ....

h1-eth0: <live capture in progress>...   Packets: 6 · Display...   Profile: Default
```

ICMP Request and Response

The final piece we will touch on for this section is the spoofing of the MAC address of the gateway. Of course, in accordance with the Ethernet standard, when we transfer from port to port, we need to rewrite the source and destination MACs. For example, had we kept everything the same but used the `00:00:00:00:00:10` MAC address for `192.168.1.1` on `s1`, `h2` would have received the following `ICMP` packet from `h1`:


Capturing from h2-eth0 [Wireshark 1.12.1 (Git Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: Expression... Clear Apply Save

```
No.  Time         Source        Destination   Protocol Length Info
   1 0.000000000  192.168.1.10  192.168.2.10  ICMP         98 Echo (ping) request  id=0x56c4, seq=1/256, ttl=64 (no response found!)
   2 1.008149000  192.168.1.10  192.168.2.10  ICMP         98 Echo (ping) request  id=0x56c4, seq=2/512, ttl=64 (no response found!)

▸ Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
▾ Ethernet II, Src: 00:00:00_00:00:01 (00:00:00:00:00:01), Dst: 00:00:00_00:00:10 (00:00:00:00:00:10)
  ▸ Destination: 00:00:00_00:00:10 (00:00:00:00:00:10)
  ▸ Source: 00:00:00_00:00:01 (00:00:00:00:00:01)
    Type: IP (0x0800)
▸ Internet Protocol Version 4, Src: 192.168.1.10 (192.168.1.10), Dst: 192.168.2.10 (192.168.2.10)
▾ Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0x05d3 [correct]
    Identifier (BE): 22212 (0x56c4)
    Identifier (LE): 50262 (0xc456)
    Sequence number (BE): 1 (0x0001)
    Sequence number (LE): 256 (0x0100)
  ▸ [No response seen]

0000  00 00 00 00 00 10 00 00  00 00 00 01 08 00 45 00   ........ ......E.
0010  00 54 3b 04 40 00 40 01  7b 40 c0 a8 01 0a c0 a8   .T;.@.@. {@......
0020  02 0a 08 00 05 d3 56 c4  00 01 97 8c e7 58 00 00   ......V. .....X..
0030  00 00 58 af 05 00 00 00  00 00 10 11 12 13 14 15   ..X..... ........
0040  16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25   ........ .. !"#$%
0050  26 27 28 29 2a 2b 2c 2d  2e 2f 30 31 32 33 34 35   &'()*+,- ./012345
0060  36 37                                              67

Frame (frame), 98 bytes    Packets: 2 · Displayed: 2 (100.0%)    Profile: Default
```

H1 to H2 ICMP without MAC spoofing

Since the MAC address was not modified, `h2` would not know the packet is destined for it and would simply drop it. Of course, we can use the OpenFlow action `OFPactionSetField` to rewrite the destination MAC address, but I have chosen to use spoofing to illustrate an important point. The point is that in a software-defined scenario, we have total control of our network. We cannot bypass certain restrictions, such as host ARP, but once the packet enters the network, we are free to route and modify based on our needs.

In the next section, we will take a look at how to add REST APIs to our static router so we can change flows dynamically.

## Router with API

We have been manually entering the flow in the last example; in particular, the flows were inserted upon the completion of switch feature negotiation. It would be great if we could dynamically insert and remove flows. Luckily, Ryu already ships with many reference applications with the REST API under the `ryu/app` directory, such as `rest_router.py`. The REST API portion of the code is also separated into its own example filename, `ofctl_rest.py`:

```
ubuntu@sdnhubvm:~/ryu_latest[15:40] (master)$ ls ryu/app/rest*
ryu/app/rest_conf_switch.py ryu/app/rest_qos.py ryu/app/rest_topology.py
ryu/app/rest_firewall.py ryu/app/rest_router.py ryu/app/rest_vtep.py
...
ubuntu@sdnhubvm:~/ryu_latest[12:34] (master)$ ls ryu/app/ofctl_rest.py
ryu/app/ofctl_rest.py
```

The `rest_router.py` application usage is covered in detail in the Ryu book, https://osrg.github.io/ryu-book/en/html/rest_router.html, with a three-switch, three-host linear topology. The example mainly consists of API usage for dealing with flow and VLAN, which would be a great reference to see what is possible with the Ryu framework. However, if you dive directly into the code, it is a bit too complex for starting out.

In this section, we will focus on adding a few API for our router controller from the previous section. We will do this by leveraging the methods and classes already created in `ofctl_rest.py`. By limiting the scope as well as leveraging our existing example, we are able to focus on adding just the API. In the following example, we will use HTTPie, which we have seen before for `HTTP GET`:

```
$ sudo apt-get install httpie
```

I find the HTTPie color-highlighting and output pretty-print easier for viewing, but using HTTPie is of course optional. I personally find the `HTTP POST` operation with a complex body a bit easier with curl. In this section, I will use `curl` for `HTTP POST` operations that include complex JSON bodies, and HTTPie for `HTTP GET` operations.

# Ryu controller with API

The example code is included in `Chapter11_4.py`. We will delete all the lines with manually added flows. These will be replaced by the following:

- `HTTP GET /network/switches` to show the DPID for all the switches
- `HTTP GET /network/desc/<dpid>` to query the individual switch description
- `HTTP GET /network/flow/<dpid>` to query the flow on the particular switch
- `HTTP POST /network/flowentry/add` to add a flow entry to a switch
- `HTTP POST /network/flowentry/delete` to delete a flow entry on a switch

To begin with, we will import the functions and classes in the `ryu.app.ofctl_rest` module:

```
from ryu.app.ofctl_rest import *
```

The import will include the WSGI modules included with the Ryu application. We already worked with WSGI in the Flask API section, and the URL dispatch and mapping will be similar:

```
from ryu.app.wsgi import ControllerBase
from ryu.app.wsgi import Response
from ryu.app.wsgi import WSGIApplication
```

Contexts are ordinary Python objects shared among Ryu applications. We are using these context objects in our code:

```
_CONTEXTS = {
    'dpset': dpset.DPSet,
    'wsgi': WSGIApplication
}
```

The `init` method includes the datastore and attributes required for the `StatsController` class, inherited from `ryu.app.wsgi.ControllerBase` and imported with `ofctl_rest.py`.

```
def __init__(self, *args, **kwargs):
    super(MySimpleRestRouter, self).__init__(*args, **kwargs)
    self.s1_gateway_mac = '00:00:00:00:00:02'
    self.s2_gateway_mac = '00:00:00:00:00:01'
    self.dpset = kwargs['dpset']
    wsgi = kwargs['wsgi']
    self.waiters = {}
    self.data = {}
    self.data['dpset'] = self.dpset
    self.data['waiters'] = self.waiters
    mapper = wsgi.mapper

    wsgi.registory['StatsController'] = self.data
```

We will declare a URL path for accessing resources. We have declared the root to be `/network`, along with the necessary action methods to obtain the resources. Luckily for us, these methods were already created under the `StatsController` class in `ofctl_rest.py`, which abstracted the actual OpenFlow actions for us:

```
path = '/network'

uri = path + '/switches'
mapper.connect('stats', uri,
               controller=StatsController, action='get_dpids',
               conditions=dict(method=['GET']))

uri = path + '/desc/{dpid}'
mapper.connect('stats', uri,
               controller=StatsController, action='get_desc_stats',
               conditions=dict(method=['GET']))

uri = path + '/flow/{dpid}'
mapper.connect('stats', uri,
               controller=StatsController, action='get_flow_stats',
               conditions=dict(method=['GET']))

uri = path + '/flowentry/{cmd}'
mapper.connect('stats', uri,
               controller=StatsController, action='mod_flow_entry',
               conditions=dict(method=['POST']))
```

As you can see, three of the URLs were `GET` with `dpid` as the variable. The last `POST` method includes the `cmd` variable, where we can use either `add` or `delete`, which will result in an `OFPFlowMod` call to the right router. The last portion is to include the function to handle different OpenFlow events, such as `EventOFPStatsReply` and `EventOFPDescStatsReply`:

```
# handling OpenFlow events
@set_ev_cls([ofp_event.EventOFPStatsReply,
             ofp_event.EventOFPDescStatsReply,
             ...
             ofp_event.EventOFPGroupDescStatsReply,
             ofp_event.EventOFPPortDescStatsReply
             ], MAIN_DISPATCHER)
```

```
def stats_reply_handler(self, ev):
    msg = ev.msg
    dp = msg.datapath
...
```

We are now ready to see the new APIs in action.

# API usage examples

We are only adding APIs for the query and flow modification with the rest of the code not changed, so the steps for launching Mininet and Ryu are identical. What has changed is that now we see the WSGI server started on port `8080` and the two switches registered as `datapath`:

```
(4568) wsgi starting up on http://0.0.0.0:8080
...
DPSET: register datapath <ryu.controller.controller.Datapath object at
0x7f83556866d0>
DPSET: register datapath <ryu.controller.controller.Datapath object at
0x7f8355686b10>
```

We can now use the API to query for existing switches:

```
$ http GET http://localhost:8080/network/switches
HTTP/1.1 200 OK
Content-Length: 6
Content-Type: application/json; charset=UTF-8
Date: <skip>
[
 1,
 2
]
```

Then we can use the more specific URL to query switch descriptions:

```
$ http GET http://localhost:8080/network/desc/1
...
{
 "1": {
 "dp_desc": "None",
 "hw_desc": "Open vSwitch",
 "mfr_desc": "Nicira, Inc.",
 "serial_num": "None",
 "sw_desc": "2.3.90"
 }
}

$ http GET http://localhost:8080/network/desc/2
...
{
 "2": {
 "dp_desc": "None",
 "hw_desc": "Open vSwitch",
 "mfr_desc": "Nicira, Inc.",
 "serial_num": "None",
 "sw_desc": "2.3.90"
 }
}
```

At this point, the switch should only have one flow for sending flow misses to the controller, which we can verify using the API as well:

```
$ http GET http://localhost:8080/network/flow/1
HTTP/1.1 200 OK
Content-Length: 251
Content-Type: application/json; charset=UTF-8
Date: <skip>

{
 "1": [
 {
 "actions": [
 "OUTPUT:CONTROLLER"
 ],
 "byte_count": 1026,
 "cookie": 0,
 "duration_nsec": 703000000,
 "duration_sec": 48,
 "flags": 0,
 "hard_timeout": 0,
 "idle_timeout": 0,
 "length": 80,
 "match": {},
 "packet_count": 13,
 "priority": 0,
 "table_id": 0
 }
 ]
}
```

We will then be able to add the flow to `s1` with the API:

```
curl -H "Content-Type: application/json"
 -X POST -d '{"dpid":"1", "priority":"1", "match": {"in_port":"1",
"dl_type":"2048", "priority":"2", "nw_src":"192.168.1.0/24",
"nw_dst":"192.168.2.0/24"}, "actions":[{"type":"OUTPUT","port":"2"}]}'
http://localhost:8080/network/flowentry/add

curl -H "Content-Type: application/json"
 -X POST -d '{"dpid":"1", "priority":"1", "match": {"in_port":"2",
"dl_type":"2048", "nw_src":"192.168.2.0/24", "nw_dst":"192.168.1.0/24"}, "actions":
[{"type":"OUTPUT","port":"1"}]}' http://localhost:8080/network/flowentry/add
```

We can repeat the process for `s2`:

```
curl -H "Content-Type: application/json"
 -X POST -d '{"dpid":"2", "priority":"1", "match": {"in_port":"2",
"dl_type":"2048", "nw_src":"192.168.1.0/24", "nw_dst":"192.168.2.0/24"}, "actions":
[{"type":"OUTPUT","port":"1"}]}' http://localhost:8080/network/flowentry/add

curl -H "Content-Type: application/json"
 -X POST -d '{"dpid":"2", "priority":"1", "match": {"in_port":"1",
"dl_type":"2048", "nw_src":"192.168.2.0/24", "nw_dst":"192.168.1.0/24"}, "actions":
```

```
[{"type":"OUTPUT","port":"2"}]}' http://localhost:8080/network/flowentry/add
```

We can use the same API for verifying flow entries. If you ever need to delete flows, you can simply change `add` to `delete`, as follows:

```
curl -H "Content-Type: application/json"
 -X POST -d '{"dpid":"1", "priority":"1", "match": {"in_port":"1",
"dl_type":"2048", "priority":"2", "nw_src":"192.168.1.0/24",
"nw_dst":"192.168.2.0/24"}, "actions":[{"type":"OUTPUT","port":"2"}]}'
http://localhost:8080/network/flowentry/delete
```

The last step is to once again verify that the ping packets work:

```
mininet> h1 ping -c 2 h2
PING 192.168.2.10 (192.168.2.10) 56(84) bytes of data.
64 bytes from 192.168.2.10: icmp_seq=1 ttl=64 time=9.25 ms
64 bytes from 192.168.2.10: icmp_seq=2 ttl=64 time=0.079 ms

--- 192.168.2.10 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 0.079/4.664/9.250/4.586 ms

# Ryu application console output for ARP only with the flow match for
# ICMP packets.
Received ARP for 192.168.1.1
('ethernet',
ethernet(dst='ff:ff:ff:ff:ff:ff',ethertype=2054,src='00:00:00:00:00:01'))
('arp',
arp(dst_ip='192.168.1.1',dst_mac='00:00:00:00:00:00',hlen=6,hwtype=1,opcode=1,plen=
datapath: 1 in_port: 1
EVENT ofp_event->MySimpleRestRouter EventOFPPacketIn
Received ARP for 192.168.2.1
('ethernet',
ethernet(dst='ff:ff:ff:ff:ff:ff',ethertype=2054,src='00:00:00:00:00:02'))
('arp',
arp(dst_ip='192.168.2.1',dst_mac='00:00:00:00:00:00',hlen=6,hwtype=1,opcode=1,plen=
datapath: 2 in_port: 1
```

The OpenFlow REST API is one of the most useful features in the Ryu framework. It is widely used in many of the Ryu application examples and production software. In the next section, we will take a look at using the BGP library for the Ryu controller.

## BGP router with OpenFlow

BGP has long been used as the external routing protocol, since the beginning of internet. It is meant for exchanging routing information between different autonomous systems without having a centralized management system. As the network starts to grow exponentially, traditional IGPs, such as IS-IS and OSPF, are not able to keep up with the resources required to keep up with the growth. The nature of BGP is also useful when different internal entities needs to operate over a common network infrastructure. An example would be to have a common core for inter-site networks in one autonomous system, a different distribution autonomous system in each data center, and a different autonomous system for each division behind the distribution fabric. In many large networks, BGP has become the routing protocol of choice in both internal and external networks.

The Ryu framework includes a BGP speaker library (http://ryu.readthedocs.io/en/latest/library_bgp_speaker.html#) that greatly simplifies the task if your SDN controller needs to communicate externally using BGP. We will look at using the Ryu SDN controller with BGP in detail in Chapter 13, *Hybrid SDN*, but this section will serve as a good introduction.

There are two ways to use the Ryu controller with BGP:

- Directly using Python and the BGP speaker library
- As a Ryu application, using `ryu/services/protocols/bgp/application.py`

We can use the VIRL router to verify our Ryu BGP configuration. Let's take a look at setting up the iosv routers.

## Lab router setup

In this section, we will use a simple two-iosv router in our VIRL lab to test a BGP neighbor connection. We will use the default auto-config for the topology so the two routers establish a BGP neighbor relationship, while using iosv-1 to establish a BGP neighbor relationship with our Ryu controller. By default, an iosv router places the first interface in the `Mgmt-intf` VRF:

```
!
vrf definition Mgmt-intf
 !
 address-family ipv4
 exit-address-family
 !
 address-family ipv6
 exit-address-family
!
interface GigabitEthernet0/0
 description OOB Management
 vrf forwarding Mgmt-intf
 ip address 172.16.1.237 255.255.255.0
 duplex full
 speed auto
 media-type rj45
!
```

We will Telnet to the console interface and take it out of the management VRF so that the routing tables converge. Note that when we do this, the IP address is removed from the interface, so we need to assign a new IP address to the interface:

```
iosv-1#confi t
Enter configuration commands, one per line. End with CNTL/Z.
iosv-1(config)#int gig 0/0
iosv-1(config-if)#description To Ryu BGP Neighbor
iosv-1(config-if)#no vrf forwarding Mgmt-intf
% Interface GigabitEthernet0/0 IPv4 disabled and address(es) removed due to
enabling VRF Mgmt-intf
iosv-1(config-if)#ip add 172.16.1.175 255.255.255.0
iosv-1(config-if)#end
iosv-1#
```

We will configure the router as ASN 65001 peering with a neighbor at the IP address of our Ryu virtual machine. We can also see that the iBGP neighbor with iosv-2 is established, but the eBGP neighbor with Ryu controller is idle:

```
iosv-1(config-if)#router bgp 1
iosv-1(config-router)#neighbor 172.16.1.174 remote-as 65000
iosv-1(config-router)#neighbor 172.16.1.174 activate
iosv-1(config-router)#end
iosv-1#
iosv-1#sh ip bgp summary
BGP router identifier 192.168.0.1, local AS number 1
...
Neighbor V AS MsgRcvd MsgSent TblVer InQ OutQ Up/Down State/PfxRcd
172.16.1.174 4 65000 0 0 1 0 0 never Active
192.168.0.2 4 1 6 6 3 0 0 00:02:34 1
iosv-1#
```

We are all set up with our BGP neighbor, so let's take a look at our first BGP speaker example.

# Python with the BGP speaker library

The first BGP speaker code in `Chapter11_5.py` is very similar to the BGP speaker example on the Ryu *readthedocs* documentation, http://ryu.readthedocs.io/en/latest/library_bgp_speaker.html#example, with a few exceptions, as we will see. Since we are launching the application directly from Python, we need to import the Ryu event queue in the form of `eventlet` as well as the BGP speaker library:

```
import eventlet
...
from ryu.services.protocols.bgp.bgpspeaker import BGPSpeaker
```

We will define two callback functions for the BGP speaker for BGP neighbor detection and best-path change:

```
def dump_remote_best_path_change(event):
    print 'the best path changed:', event.remote_as, event.prefix,
        event.nexthop, event.is_withdraw

def detect_peer_down(remote_ip, remote_as):
    print 'Peer down:', remote_ip, remote_as
```

We will define the BGP speaker with the necessary parameters and add iosv-1 as our neighbor. Every 30 seconds, we will announce a prefix as well as printing out our current neighbors and **Routing Information Base** (**RIB**):

```
if __name__ == "__main__":
    speaker = BGPSpeaker(as_number=65000, router_id='172.16.1.174',
                         best_path_change_handler=dump_remote_best_path_change,
                         peer_down_handler=detect_peer_down)

    speaker.neighbor_add('172.16.1.175', 1)
    count = 0
    while True:
        eventlet.sleep(30)
        prefix = '10.20.' + str(count) + '.0/24'
        print "add a new prefix", prefix
        speaker.prefix_add(prefix)
        count += 1
        print("Neighbors: ", speaker.neighbors_get())
        print("Routes: ", speaker.rib_get())
```

The application needs to spawn a child process for the BGP speaker, so we need to launch it as root:

```
$ python mastering_python_networking/Chapter11_5.py
```

We can see that the iosv-1 router now shows the neighbor as up and established:

```
*Apr  9 15:57:11.272: %BGP-5-ADJCHANGE: neighbor 172.16.1.174 Up
```

Every 30 seconds, the router will receive BGP routes from the Ryu controller:

```
 0iosv-1#sh ip bgp summary
BGP router identifier 192.168.0.1, local AS number 1
...
Neighbor V AS MsgRcvd MsgSent TblVer InQ OutQ Up/Down State/PfxRcd
172.16.1.174 4 65000 9 11 31 0 0 00:01:07 2
192.168.0.2 4 1 40 49 31 0 0 00:33:30 1

# Showing BGP routes from Ryu Controller
iosv-1#sh ip route bgp
...
 10.0.0.0/8 is variably subnetted, 4 subnets, 3 masks
B 10.20.0.0/24 [20/0] via 172.16.1.174,0:00:56
B 10.20.1.0/24 [20/0] via 172.16.1.174, 00:00:27
```

On the controller console, you can see the BGP keepalive exchange messages:

```
Peer Peer(ip: 172.16.1.175, asn: 1) asked to communicate path
...
Sent msg to ('172.16.1.175', '179') >> BGPKeepAlive(len=19,type=4)
Received msg from ('172.16.1.175', '179') << BGPKeepAlive(len=19,type=4)
```

Note that we can see the neighbors as well as routes from iosv-2 propagated from BGP:

```
('Neighbors: ', '[{"bgp_state": "Established", "ip_addr": "172.16.1.175", "as_num":
"1"}]')
...
API method operator.show called with args: {'params': ['rib', 'all'], 'format':
'json'}
('Routes: ', '{"vpnv6": [], "ipv4fs": [], "vpnv4": [], "vpnv4fs": [], "ipv4":
[{"paths": [{"origin": "i", "aspath": [1], "prefix": "192.168.0.1/32", "bpr": "Only
Path", "localpref": "", "nexthop": ... "prefix": "192.168.0.2/32"},...
```

Using Python and the BGP speaker library gives you very granular control, but it takes away the operations Ryu-manager does for you, such as managing the event queue and the OpenFlow event manager. It also requires you to have extensive knowledge of the BGP speaker library API, which can get pretty complicated when it comes to more advanced features such as MPLS or FlowSpec. Let's look at how to run BGP as a Ryu application, which simplifies the process quite a bit.

## Ryu BGP application

Ryu can run a BGP application with the `--bgp-app-config-file` option, which can be taken in as input by the `ryu/serivces/protocols/bgp/applications.py` module:

```
$ sudo ryu-manager --verbose --bgp-app-config-file <config_file>
ryu/services/protocols/bgp/application.py
```

There is a sample configuration file located in the same directory for reference:

```
$ less ryu/services/protocols/bgp/bgp_sample_conf.py
```

In our configuration file, `Chapter11_6.py`, we will start with a simple configuration file that establishes the BGP neighbor relationship and provides an SSH console:

```
BGP = {
    'local_as': 65000,
    'router_id': '172.16.1.174',
    'neighbors': [
        {
            'address': '172.16.1.175',
            'remote_as': 1,
            'enable_ipv4': True,
            'enable_ipv6': True,
            'enable_vpnv4': True,
            'enable_vpnv6': True
        },
    ]
}

SSH = {
    'ssh_port': 4990,
    'ssh_host': 'localhost',
    # 'ssh_host_key': '/etc/ssh_host_rsa_key',
    # 'ssh_username': 'ryu',
    # 'ssh_password': 'ryu',
}
```

You can launch the application using the `--bgp-app-config-file` option:

```
$ sudo ryu-manager --verbose --bgp-app-config-file
mastering_python_networking/Chapter11_6.py
ryu/services/protocols/bgp/application.py
```

You can SSH to the localhost and take a look at the neighbors, RIB, and so on:

```
$ ssh localhost -p 4990

Hello, this is Ryu BGP speaker (version 4.13).

bgpd>
bgpd> show neighbor
 IP Address AS Number BGP State
 172.16.1.175 1 Established
bgpd> show rib all
Status codes: * valid, > best
Origin codes: i - IGP, e - EGP, ? - incomplete
 Network Labels Next Hop Reason Metric LocPrf Path
Family: vpnv6
Family: ipv4fs
Family: vpnv4
Family: vpnv4fs
Family: ipv4
 *> 192.168.0.1/32 None 172.16.1.175 Only Path 0 1 i
 *> 192.168.0.2/32 None 172.16.1.175 Only Path 1 i
Family: ipv6
Family: rtfilter
Family: evpn
bgpd> quit
bgpd> bye.
Connection to localhost closed.
```

In the `Chapter11_7.py` configuration file, I've also included a section for route advertisement:

```
'routes': [
    # Example of IPv4 prefix
    {
        'prefix': '10.20.0.0/24',
        'next_hop': '172.16.1.1'
    },
    {
        'prefix': '10.20.1.0/24',
        'next_hop': '172.16.1.174'
    },
 ]
```

In it, we can see the Ryu controller advertised to iosv-1:

```
iosv-1#sh ip route bgp
...
Gateway of last resort is not set

 10.0.0.0/8 is variably subnetted, 4 subnets, 3 masks
B 10.20.0.0/24 [20/0] via 172.16.1.1, 00:01:16
B 10.20.1.0/24 [20/0] via 172.16.1.174, 00:01:16
```

Note that the `next_hop` attribute for the IPv4 prefix is pointed to `172.16.1.1` for the prefix `10.20.0.0/24`. This is a great example of being able to flexibly control the BGP attributes that you wish to make (if you know what you are doing). The `ryu/services/protocols/bgp/bgp_sample_config.py` file provides lots of great examples and features; I would encourage you to take a closer look at what is available in the configuration file. In the next section, we will take a look at how to use the Ryu controller to simulate a firewall or access-list function.

# Firewall with OpenFlow

For the firewall function demonstration, we will use the existing `ryu/app/rest_firewall.py` application and combine the knowledge we gained in this chapter about the REST API and flow modification. This is a simplified version of the Ryu book firewall example, https://osrg.github.io/ryu-book/en/html/rest_firewall.html. Besides firewall operations, the Ryu firewall example in the link provided illustrates the usage of multi-tenancy with VLAN. The example covered in this section will focus on the correlation between the firewall application and OpenFlow for learning purposes. This section and the linked example would be good complements of each other.
If you take a look at the `rest_firewall.py` code, you will notice a lot of similarities with `ofctl_rest.py`. In fact, most of the `rest_*` applications have similar functions when it comes to callback functions and URL dispatch. This is great news for us, since we only have to learn the pattern once and can apply it to multiple applications.
To begin with, we will launch a two-host, single-switch topology in Mininet:

```
$ sudo mn --topo single,2 --mac --switch ovsk --controller remote
```

We can launch the firewall application:

```
$ ryu-manager --verbose ryu/app/rest_firewall.py
```

We can perform a quick flow dump and see that the initial state of the flow includes a drop action at the top:

```
$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=5.849s, table=0, n_packets=0, n_bytes=0, idle_age=5,
priority=65535 actions=drop
 cookie=0x0, duration=5.848s, table=0, n_packets=0, n_bytes=0, idle_age=5,
priority=0 actions=CONTROLLER:128
 cookie=0x0, duration=5.848s, table=0, n_packets=0, n_bytes=0, idle_age=5,
priority=65534,arp actions=NORMAL
```

We can enable the firewall via the API under `/firewall/module/enable/<dpid>`, which we will do now:

```
$ curl -X PUT http://localhost:8080/firewall/module/enable/0000000000000001
[{"switch_id": "0000000000000001", "command_result": {"result": "success",
"details": "firewall running."}}]
```

If we perform the flow dump again, we will see the drop action is gone:

```
$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=964.894s, table=0, n_packets=0, n_bytes=0, idle_age=964,
priority=65534,arp actions=NORMAL
 cookie=0x0, duration=964.894s, table=0, n_packets=0, n_bytes=0, idle_age=964,
priority=0 actions=CONTROLLER:128adfa
```

But there is still no flow from `h1` to `h2`, so the ping packet will be blocked and dropped:

```
mininet> h1 ping -c 1 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

# On Ryu console
EVENT ofp_event->RestFirewallAPI EventOFPPacketIn
[FW][INFO] dpid=0000000000000001: Blocked packet =
ethernet(dst='00:00:00:00:00:02',ethertype=2048,src='00:00:00:00:00:01'),
ipv4(csum=56630,dst='10.0.0.2',flags=2,header_length=5,identification=18800,offset=

icmp(code=0,csum=37249,data=echo(data='Vjxe8Xx00x00x00x00xfe8tx00x00x00x00x00x10x11
 !"#$%&'()*+,-./01234567',id=25006,seq=1),type=8)
```

So let's add the rules for allowing `ICMP` packets--remember that this needs to be done bidirectionally, so we need to add two rules:

```
$ curl -X POST -d '{"nw_src": "10.0.0.1/32", "nw_dst": "10.0.0.2/32", "nw_proto":
"ICMP"}' http://localhost:8080/firewall/rules/0000000000000001
$ curl -X POST -d '{"nw_src": "10.0.0.2/32", "nw_dst": "10.0.0.1/32", "nw_proto":
"ICMP"}' http://localhost:8080/firewall/rules/0000000000000001
```

If we perform a flow dump again, we will see the flow installed:

```
$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=1143.144s, table=0, n_packets=2, n_bytes=84, idle_age=101,
priority=65534,arp actions=NORMAL
 cookie=0x1, duration=16.060s, table=0, n_packets=0, n_bytes=0, idle_age=16,
priority=1,icmp,nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=NORMAL
 cookie=0x2, duration=3.662s, table=0, n_packets=0, n_bytes=0, idle_age=3,
priority=1,icmp,nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=NORMAL
 cookie=0x0, duration=1143.144s, table=0, n_packets=1, n_bytes=98, idle_age=101,
priority=0 actions=CONTROLLER:128
```

As expected, the ping packet from `h1` to `h2` now succeeds:

```
mininet> h1 ping -c 1 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.423 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.423/0.423/0.423/0.000 ms
```

But since the flow is implemented specifically for ICMP, other traffic will still be blocked:

```
mininet> h2 python -m SimpleHTTPServer &
mininet> h1 curl http://10.0.0.2:8000

# Ryu console
[FW][INFO] dpid=0000000000000001: Blocked packet =
ethernet(dst='00:00:00:00:00:02',ethertype=2048,src='00:00:00:00:00:01'),
ipv4(csum=49315,dst='10.0.0.2',flags=2,header_length=5,identification=26134,offset=
 tcp(ack=0,bits=2,csum=6514,dst_port=8000,offset=10,option=
[TCPOptionMaximumSegmentSize(kind=2,length=4,max_seg_size=1460),
TCPOptionSACKPermitted(kind=4,length=2),
TCPOptionTimestamps(kind=8,length=10,ts_ecr=0,ts_val=34384732),
TCPOptionNoOperation(kind=1,length=1),
TCPOptionWindowScale(kind=3,length=3,shift_cnt=9)],seq=677247676,src_port=47286,urg
```

As you can see, the firewall application is a way to use SDN to implement a traditional network function based on packet characteristics and implement it as flows to the network device. In this example, we implemented what is traditionally a layer 4 firewall rule in terms of SDN, Ryu, and OpenFlow.

## Summary

In this chapter, we built on the previous chapter's knowledge on OpenFlow and Ryu for advanced functions. We started with parsing out the different OpenFlow functions and decoding packets with the Ryu packet library. We then implemented our own static router between two routers, each with a host connected to a `/24` subnet. The router is responsible for answering ARP packets and installing static flows between the two hosts. After the static router successfully transported traffic between the two hosts, we implemented the REST API and removed the static flow in the code. This gave us flexibility to dynamically insert flows in the router we coded.

In the section that followed, we looked at two ways to implement a BGP router with Ryu. The first method requires more Python knowledge by directly calling the BGP library of the Ryu framework. The second method uses the `--bgp-app-config-file` option with the Ryu BGP application. The first method gave us more granular control of our app but that can become tedious, while the second option abstracts the lower-level code but at the expense of control. Both methods have their advantages and disadvantages.

In the last section of the chapter, we looked at the firewall application Ryu provides as an example of how we can simulate a traditional layer 4 firewall function in a network device with OpenFlow.

In the next chapter, we will look at more SDN options with OpenStack and OpenDaylight. Both of the projects carry great industry momentum with both commercial vendors and open source community support.

# OpenStack, OpenDaylight, and NFV

It seems there is no shortage of network engineering projects named after the term **Open** these days. Besides the Open vSwitch andÂ OpenFlow projects, that we have already studied in the last two chapters, there are community driven projects such as **Open Network Operating System** (**ONOS**,Â http://onosproject.org/), OpenDaylight (https://www.opendaylight.org/), and OpenStack (https://www.openstack.org/). Furthermore, there are other projects that are open in nature but arguably heavily backed or bootstrapped by commercial companies (such as Ryu with NTT); for example **OpenContrail** (http://www.opencontrail.org/) with Juniper Networks and Open Compute Project (http://www.opencompute.org/) bootstrapped byÂ Facebook. There is also the **Open Networking User Group** (**ONUG**,Â https://opennetworkingusergroup.com/) that, according to their mission statement, enable greater choices and options for IT business leaders by advocating for open interoperable hardware and software-defined infrastructure solutions.Â

*For more open and community driven initiatives, there is a growing list on the Wikipedia page https://en.wikipedia.org/wiki/List_of_SDN_controller_software.Â*

On one hand, it is terrific to see soÂ much innovation and community involvement in the network engineering field. On the other hand, it feels a bit overwhelming if one triesÂ to understand and learn about all of these different projects. Many, if not all of them, use the term **Software Defined Networking** (**SDN**) and **Network Function Virtualization** (**NFV**)Â to describe their mission and features when it comes to the technology they bring forward. Many of the projects utilize the same technology, such as OpenFlow, but add different orchestration engines or APIs on top to target slightly different audience.

We have already discussed in detail about OpenFlow in the previous two chapters. In this chapter, we will take a look at the two otherÂ projects that have general industry support amongst the community and vendors:

- **OpenStack**: It is an open source Infrastructure-as-a- ServiceÂ cloud platform with integrated components forÂ the datacenter environment. There are many different components within the project, such as compute, storage, and networking resources. We will focus on the networking portion, code name Neutron, within OpenStack.Â
- **OpenDaylight**: It is a community-led and industry-supported open source project with aims to accelerate the adoption of SDN and NFV with more transparency. The project was administrated by the Linux Foundation from day one and the founding members include industry leaders such as Cisco, Juniper, Arista, Brocade, Microsoft, VMware, Red Hat, and many more.Â

WhenÂ discussing the two projects, we will look at how each project implements NFV. NFV is a concept where we can use software and virtualization technologies to simulate traditional network functions, such as routing, switching, load balancing, firewalls, caching, and more. In the previous chapter, we saw how we can use OpenFlow and Ryu controller to simulate functions such as routing, switching, and traditional firewall, all using the same Mininet Open vSwitch.Â
Many full size books have been written on both OpenStack and OpenDaylight. It would simply be a false hope trying to cover the two topics in depth within the constrains of a chapter. Both these subjects represent important steps forward in the network engineering world, that are worth spending time understanding. In this chapter, we will try to establish a solidÂ basic understanding of both projects from network engineering perspective, so the reader can dive into each topic deeper in the future.Â

Additionally, we will cover the following topics:Â

- Networking in OpenStack
- Trying out OpenStackÂ
- OpenDaylight Programming overviewÂ
- OpenDaylight example

## OpenStack

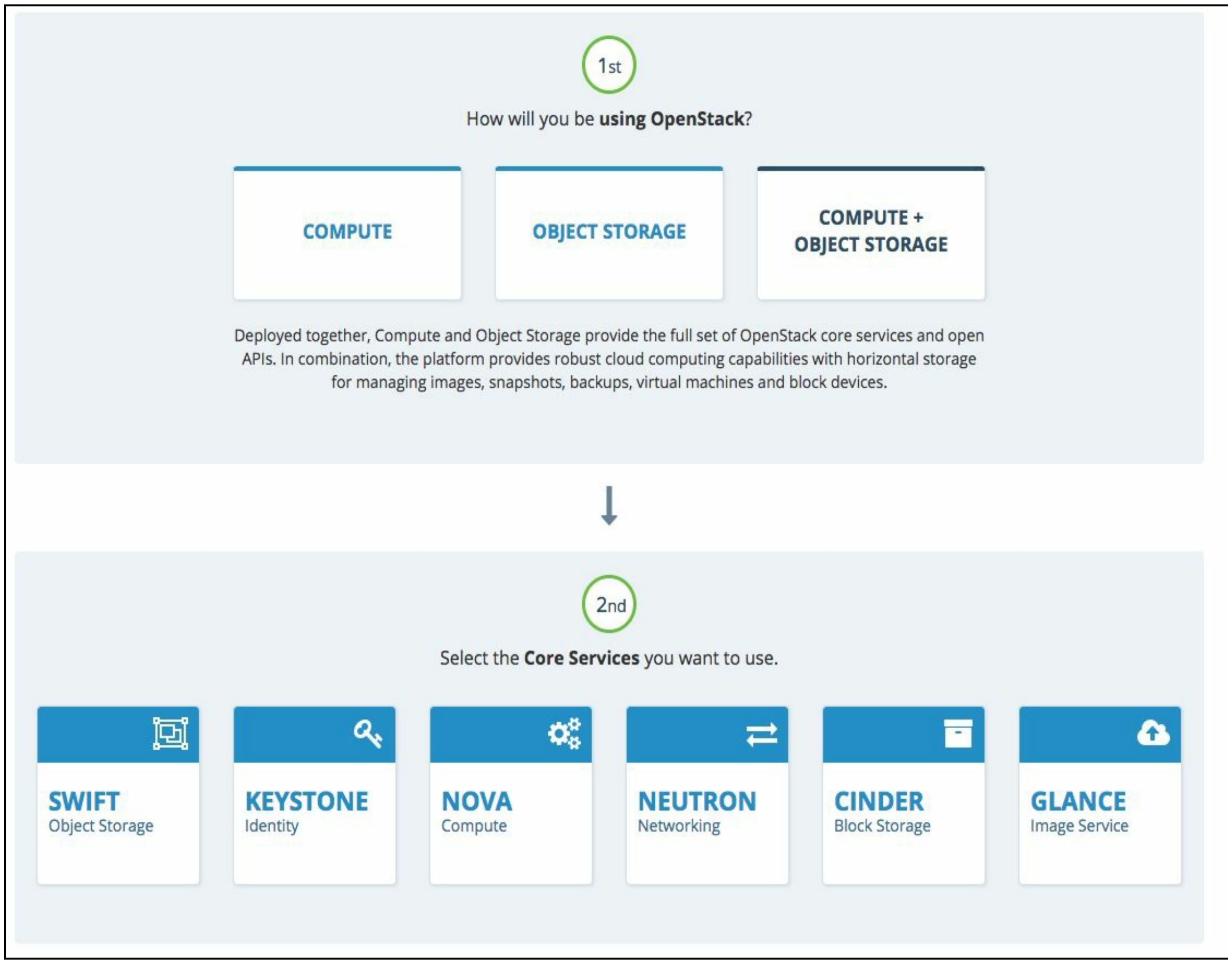
OpenStack tries to solve a very complex problem, which is to create an open source, flexible, extensible **Infrastructure-as-a- Service** (**IaaS**) datacenter. Imagine if you are creating an Amazon AWS or Microsoft Azure public cloud from your own bare metal server and off-the-shelf network equipment, and you can begin to see the complication that the process involves. In order to be scalable, the required components need to be broken off into self-contained components. Each of the component consists of sub-components and plugins that allow for more extension and community involvement. Let's take a look at the core and associated components at a high level in the next section.

## OpenStack overview

In OpenStack, the overall components can be separated into the following core services:

- **Compute**, **code named nova**: It is the computing fabric controller for managing and controlling computing resources. It works with bare metal servers as well as hypervisor technologies, such as KVM, Xen, Hyper-V, and many more. Nova is written in Python.
- **Networking**, **code name Neutron**: This is the system that manages networks and associated services. The physical network can be flat, VLAN, or VXLAN that separates control and user traffic with the virtual network work in conjunction with the underlying physical network. Network services such as IP addressing (DHCP or static), floating IP from control node to user network, L2 and L3 services reside within the networking services. This is where the network plugin can be installed to provide additional services such as intrusion detection system, load balancing, firewall, and VPN. This is the focus of our chapter.
- **Block Storage, code named Cinder**: The block storage is used with compute instances to provide block-level storage, similar to a virtual hard drive.
- **Identify, code named keystone**: Keystone provides central directory of user rights management.
- **Image service, code named glance**: This is service that provides discovery, registration, and delivery for disk and server images.
- **Object Storage, code named Swift**: Swift is the backend redundant storage system for data replication and integrity across multiple disk drives and servers.
- **Dashboard, code name Horizon**: This is the dashboard for access, provision, and automate deployment. Technically this is classified as an optional service for enhancement. However, I personally feel this is a core service that is required for deployment.

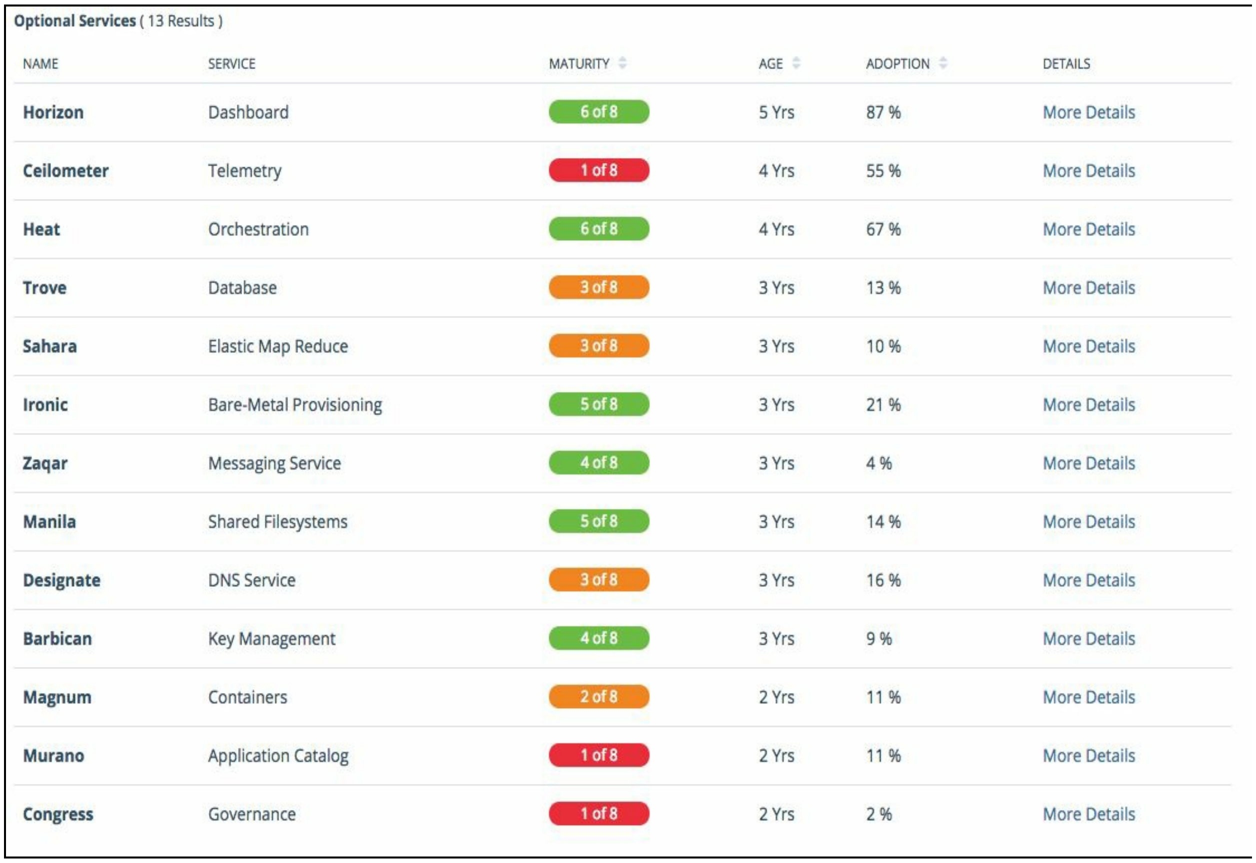
Here is a visualization of the core services of OpenStack:



Deployed together, Compute and Object Storage provide the full set of OpenStack core services and open APIs. In combination, the platform provides robust cloud computing capabilities with horizontal storage for managing images, snapshots, backups, virtual machines and block devices.

OpenStack Core Services (source: https://www.openstack.org/software/)

There are many more optional services that enhance the services, such as database, telemetry, messaging service, DNS, orchestration, and many more. Here is a brief overview of some of the optional services:

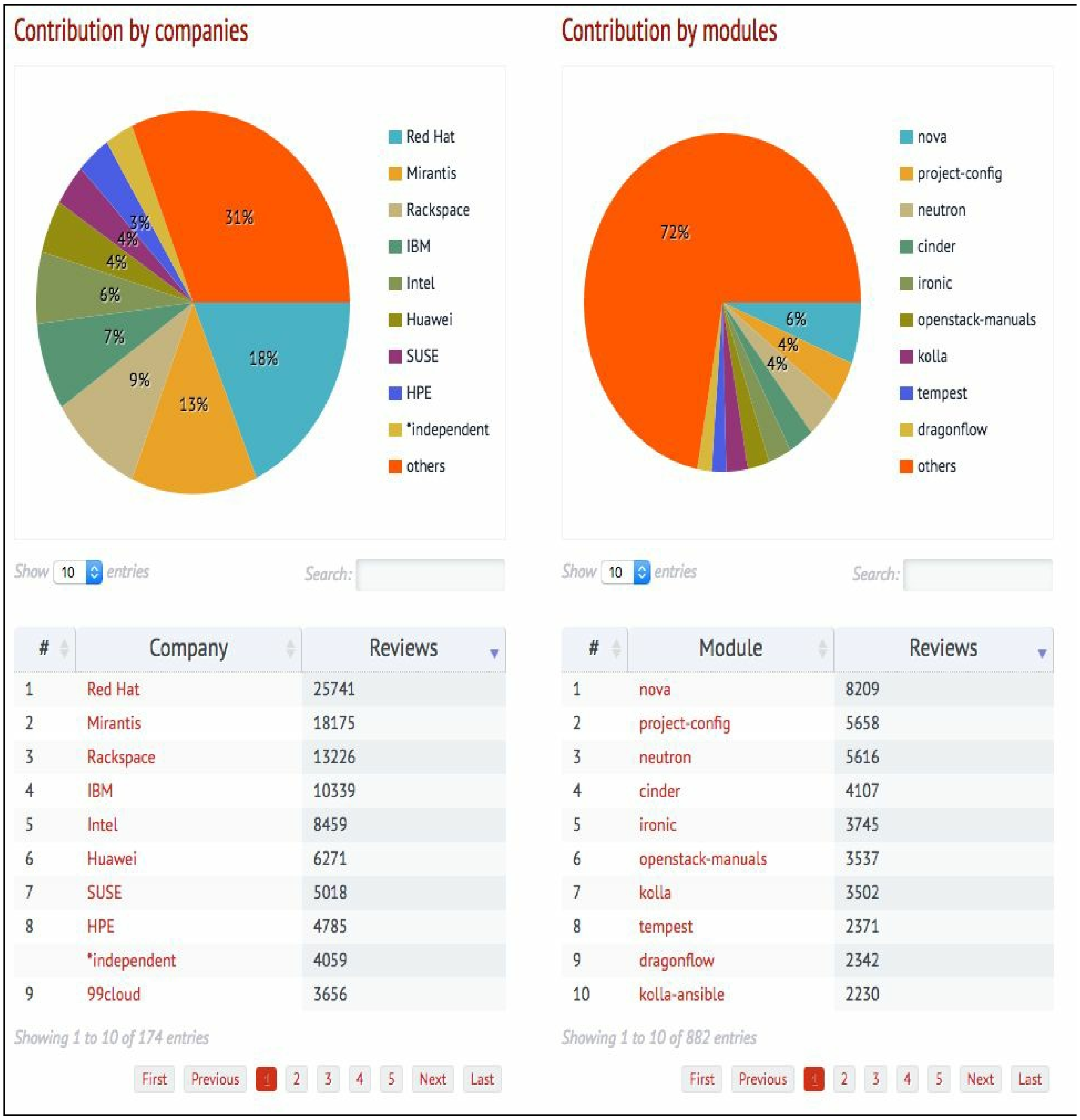


**Optional Services** ( 13 Results )

| NAME | SERVICE | MATURITY | AGE | ADOPTION | DETAILS |
|---|---|---|---|---|---|
| **Horizon** | Dashboard | 6 of 8 | 5 Yrs | 87 % | More Details |
| **Ceilometer** | Telemetry | 1 of 8 | 4 Yrs | 55 % | More Details |
| **Heat** | Orchestration | 6 of 8 | 4 Yrs | 67 % | More Details |
| **Trove** | Database | 3 of 8 | 3 Yrs | 13 % | More Details |
| **Sahara** | Elastic Map Reduce | 3 of 8 | 3 Yrs | 10 % | More Details |
| **Ironic** | Bare-Metal Provisioning | 5 of 8 | 3 Yrs | 21 % | More Details |
| **Zaqar** | Messaging Service | 4 of 8 | 3 Yrs | 4 % | More Details |
| **Manila** | Shared Filesystems | 5 of 8 | 3 Yrs | 14 % | More Details |
| **Designate** | DNS Service | 3 of 8 | 3 Yrs | 16 % | More Details |
| **Barbican** | Key Management | 4 of 8 | 3 Yrs | 9 % | More Details |
| **Magnum** | Containers | 2 of 8 | 2 Yrs | 11 % | More Details |
| **Murano** | Application Catalog | 1 of 8 | 2 Yrs | 11 % | More Details |
| **Congress** | Governance | 1 of 8 | 2 Yrs | 2 % | More Details |

OpenStack Optional Services (source: https://www.openstack.org/software/project-navigator)

As you can tell by the size of the components, OpenStack is a very big and complex project. In the latest Ocata release (less than 6 months release cycle between Newton in October 2016 to Ocata in February 2017), there are hundreds of contributors with tens of thousands of code and associated code reviews in each month to help bring the latest release to fruition.



Contribution by companies

| # | Company | Reviews |
|---|---|---|
| 1 | Red Hat | 25741 |
| 2 | Mirantis | 18175 |
| 3 | Rackspace | 13226 |
| 4 | IBM | 10339 |
| 5 | Intel | 8459 |
| 6 | Huawei | 6271 |
| 7 | SUSE | 5018 |
| 8 | HPE | 4785 |
|  | *independent | 4059 |
| 9 | 99cloud | 3656 |

Showing 1 to 10 of 174 entries

Contribution by modules

| # | Module | Reviews |
|---|---|---|
| 1 | nova | 8209 |
| 2 | project-config | 5658 |
| 3 | neutron | 5616 |
| 4 | cinder | 4107 |
| 5 | ironic | 3745 |
| 6 | openstack-manuals | 3537 |
| 7 | kolla | 3502 |
| 8 | tempest | 2371 |
| 9 | dragonflow | 2342 |
| 10 | kolla-ansible | 2230 |

Showing 1 to 10 of 882 entries

OpenStack Ocata Release Contribution (source: http://stackalytics.com/?release=ocata)

The point I am trying to make is that moving over to OpenStack only makes sense if you deploy the project as a whole, with all the components intact. You could choose to deploy any individual component, such as OpenStack Networking (neutron), but the benefit would be no different than from deploying a number of other network abstraction technologies. The true benefit of OpenStack comes from the direct and seamless integration and orchestration of all the components working together.

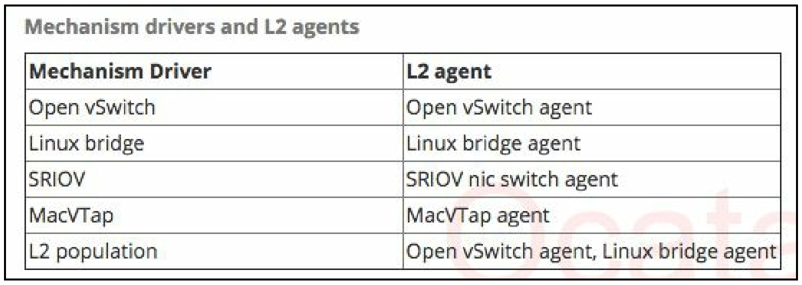
The decision to deploy OpenStack generally is not a network engineer driven one. It is generally a high-level business decision, jointly made by both business and technical personnels in a company. In my opinion, as a network engineer, we should exercise due diligence in investigating if our physical network and technology in-use today can accommodate the OpenStack virtualized network; for example, if we should use flat, VLAN, or VXLAN for management and user network overlay. We should start with how network virtualization is done within OpenStack, which we will examine in the next section.

# Networking in OpenStack

The OpenStack Networking Guide, https://docs.openstack.org/ocata/networking-guide/index.html, is a detailed and authoritative source for networking in OpenStack. The guide provides a high level overview of the technology, dependencies, and deployment examples. Here is an overview and summary of networking in OpenStack:

- OpenStack networking handles the creation and management of a virtual network infrastructure, including networks, switches, subnets, and routers for devices managed by OpenStack services.
- Additional services, such as firewalls or VPN, can also be used in the form of a plugins.
- OpenStack networking consists of the neutron-server, database for network information storage, and plugin agents. The agents are responsible for the interaction between the Linux networking mechanism, external devices, and SDN controllers.
- The networking service leverages Keystone identify services for user authentication, Nova compute services for connect and disconnect virtual NIC on the VM to a customer network, and horizon dashboard services for administrator and tenant users to create network services.
- The main virtualization underpin is the Linux namespace. A Linux namespace is a way of scoping a particular set of identifiers, which are provided for networking and processes. If a process or a network device is running within a process namespace, it can only see and communicate with other processes and network devices in the same namespace.
- Each network namespace has its own routing table, IP address space, and IP tables.
- Modular layer 2 (ml2) neutron plugin is a framework allowing OpenStack networking to simultaneously use the variety of layer 2 networking such as VXLAN and Open vSwitch. They can be used simultaneously to access different ports of the same virtual network.

- An L2 agent serves layer 2 network connectivity to OpenStack resources. It is typically run on each network node and compute node (please see diagram below for the L2 agents). Available agents are Open vSwitch agent, Linux bridge agent, SRIOV NIC switch agent, and MacVTap agent.



Mechanism drivers and L2 agents

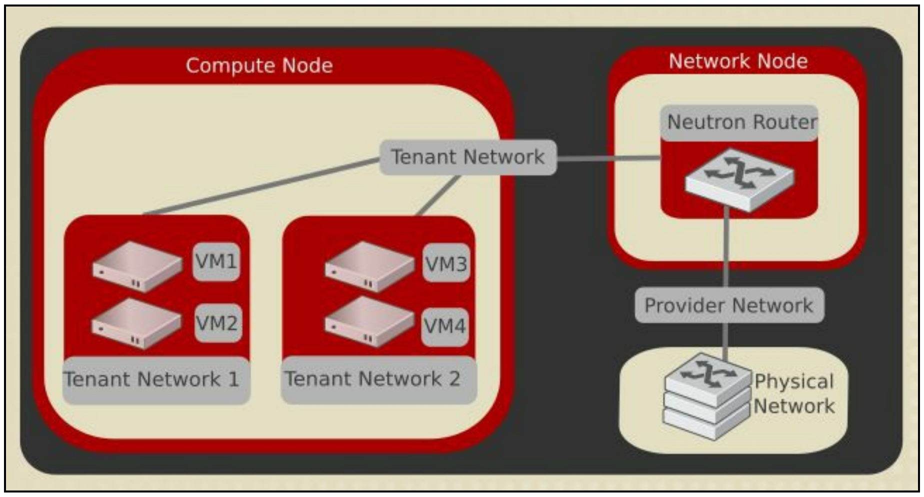
| Mechanism Driver | L2 agent |
|---|---|
| Open vSwitch | Open vSwitch agent |
| Linux bridge | Linux bridge agent |
| SRIOV | SRIOV nic switch agent |
| MacVTap | MacVTap agent |
| L2 population | Open vSwitch agent, Linux bridge agent |

L2 Agents (source: https://docs.openstack.org/ocata/networking-guide/config-ml2.html)

- An L3 agent offers advanced layer 3 services, such as virtual routers and fFloating IPs. It requires an L2 agent running in parallel.

OpenStack Networking service neutron includes the following components:

- **API server**: This component includes support for layer 2 networking and IP address space management, as well as an extension for a layer 3 router construct that enables routing between a layer 2 network and gateways to external networks.
- **PlugIn and agents**: This component creates subnets, provides IP addressing, connect and disconnect ports.
- **Messaging queue**: This queue accepts RPC requests between agents to complete API operations.
- The network service can reside in the control node or in its own node:

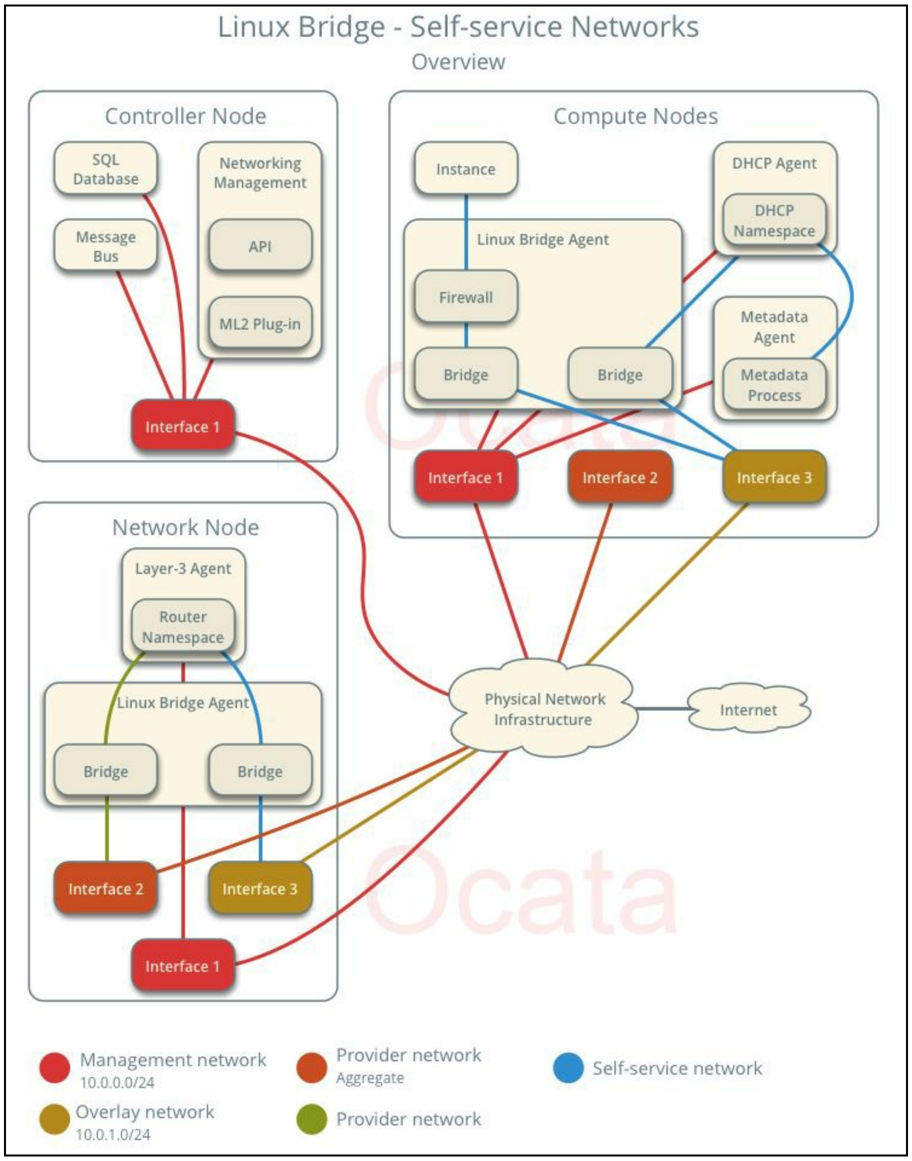


Compute and Network Node(source: https://docs.openstack.org/ocata/networking-guide/intro-os-networking.html)

An important difference in OpenStack networking is the difference in provider and tenant networks:

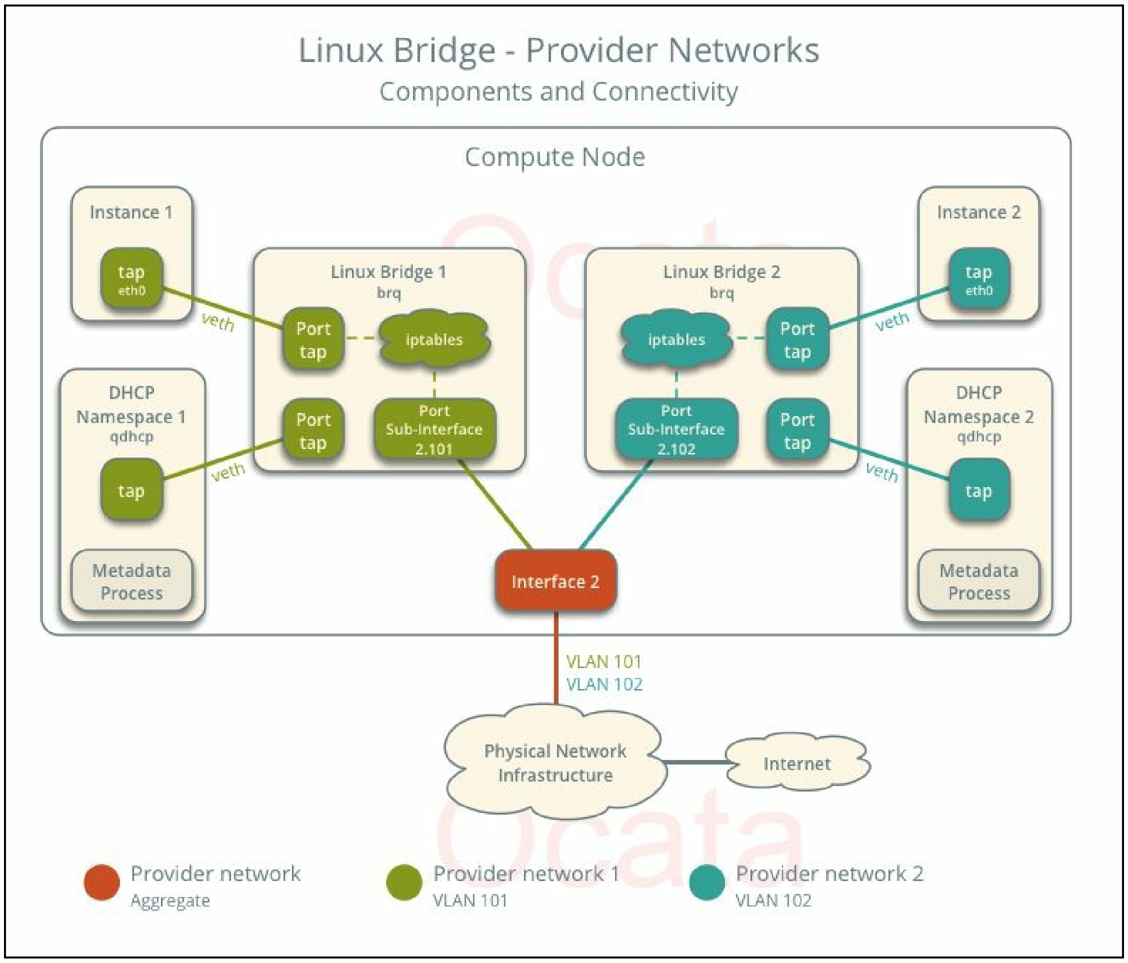
- OpenStack compute is a consumer of OpenStack networking to provide connectivity for its instances. This is done through the combination of provider and tenant networks.
- The provider network offers layer 2 connectivity to instances with optional support for DHCP and metadata services. The provider network connects to the existing layer 2 network in the datacenter, typically using VLAN. Only an administrator can manage the provider network.
- The tenant networks are self-service networks that use tunneling mechanism, such as VXLAN or GRE. They are entirely virtual and self-contained. In order for the tenant network to connect to the provider network or external network, a virtual router is used.
- From tenant network to provider network, source network address translation is used on the virtual router.
- From provider to tenant network, destination network address translation with a floating IP is used on the virtual router.

Using the Linux bridge as an example, let's take a look at how the provider network is mapped to the physical network:



Linux Bridge Tenant Network (source: https://docs.openstack.org/ocata/networking-guide/deploy-lb-selfservice.html)

The following graph includes are more detailed link topology for interface 2:



Linux Bridge Provider Network (source: https://docs.openstack.org/ocata/networking-guide/deploy-lb-provider.html)

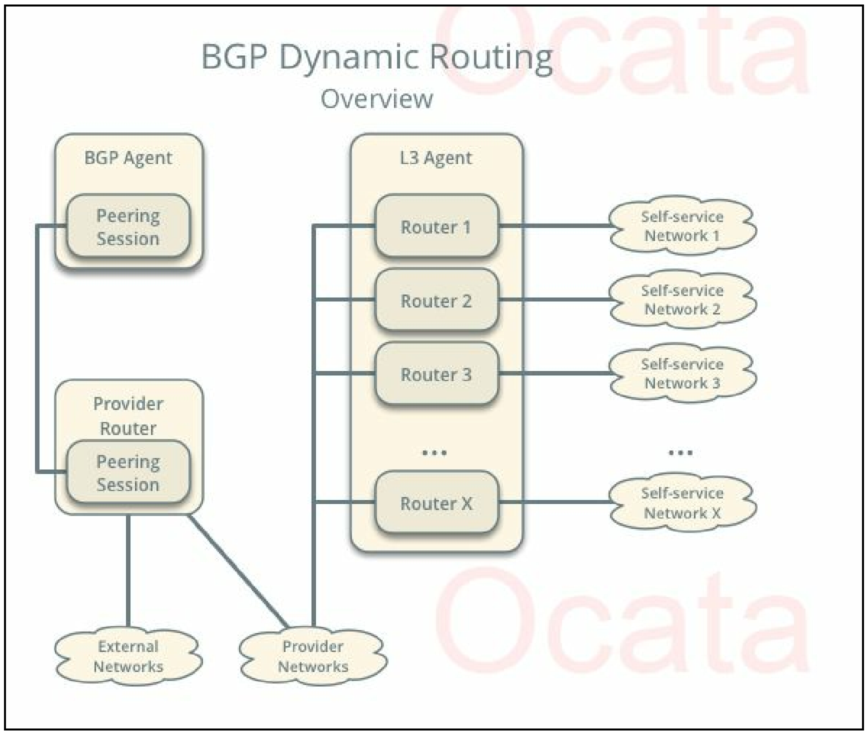
Let us take a closer look at the preceding topology:

- Interface 1 is a physical interface with an IP address dedicated for management purposes. This is typically an isolated VLAN in the physical infrastructure. The controller node receives the network management API call, and through the management interface on the compute node, configures the DHCP agent and metadata agent for the tenant network instance.
- Interface 2 on the compute node is dedicated as a L2 trunk interface to the physical network infrastructure. Each instance has its dedicated Linux bridge instance with separate IP table and virtual Ethernet ports. Each instance is tagged with a particular VLAN ID for separation and trunked through interface 2 to the provider network. In this deployment,

the instance traffic traverses through the same VLAN as the provider network in the same VLAN.

- Interface 3 provides an additional overlay option for tenant networks, if the physical network supports VXLAN. In the compute node, instead of having the Linux bridge tag with VLAN ID, the Linux bridge uses VXLAN VNI for identification. It then traverses through the physical network to reach the Network node with the same VXLAN that connects to the router namespace. The router namespace has a separate virtual connection to the Linux bridge to interface 2 for the provider network.

Here is an example of how a layer 3 agent can interact with the BGP router in the provider network, which in turn peers with the external network:



BGP Dynamic Routing (source: https://docs.openstack.org/ocata/networking-guide/config-bgp-dynamic-routing.html)

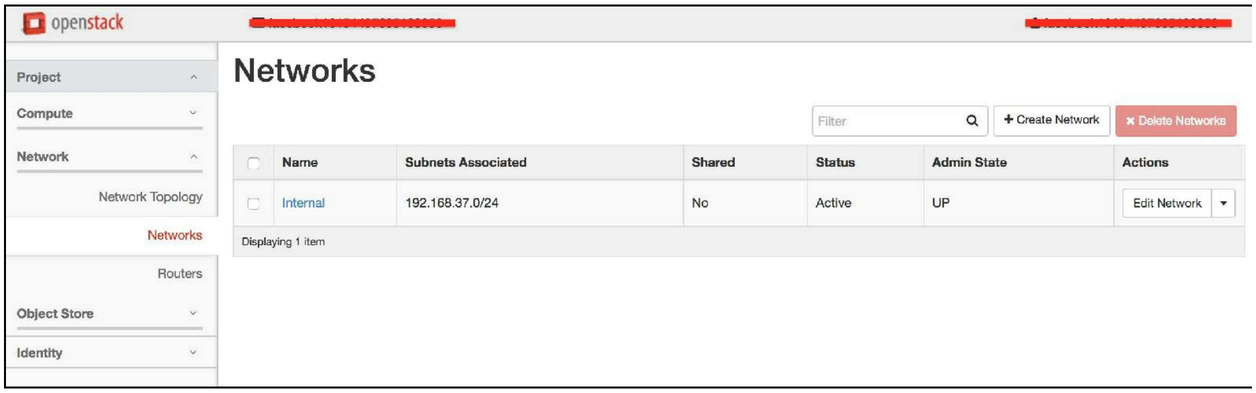
In the case of a Linux bridge with BGP routing, the L3 gateway would be on the network node that can directly or indirectly peer with the external network. In the next section, we will take a look at some of the command syntax of OpenStack configuration.

# Trying out OpenStack

There are two ways to try out OpenStack with minimal footprint. The first option is an all-in-one installation called **DevStack**, https://docs.openstack.org/developer/devstack/#all-in-one-single-vm. It consists of cloning the DevStack Git repository and installing a series of tools and scripts to enable a OpenStack in an all-in-one environment.

*DevStack will run as root and substantially make changes to your system. Therefore, it is best to run it on a server or a virtual machine that you can dedicate to this purpose.*

The second option is to use a cloud-based sandbox called **TryStack**, http://trystack.org/. The account is free but the process is somewhat manual at this time. Also, the project is made possible by the generosity of companies such as Dell, NetApp, Cisco, and Red Hat, so your mileage might vary by the time you read this. But TryStack is a truly easy way to get a look and feel of the dashboard and launch your first OpenStack instance.  They have a great quick start video that I will not repeat here, here is the screenshot of the horizon dashboard of the internal network:



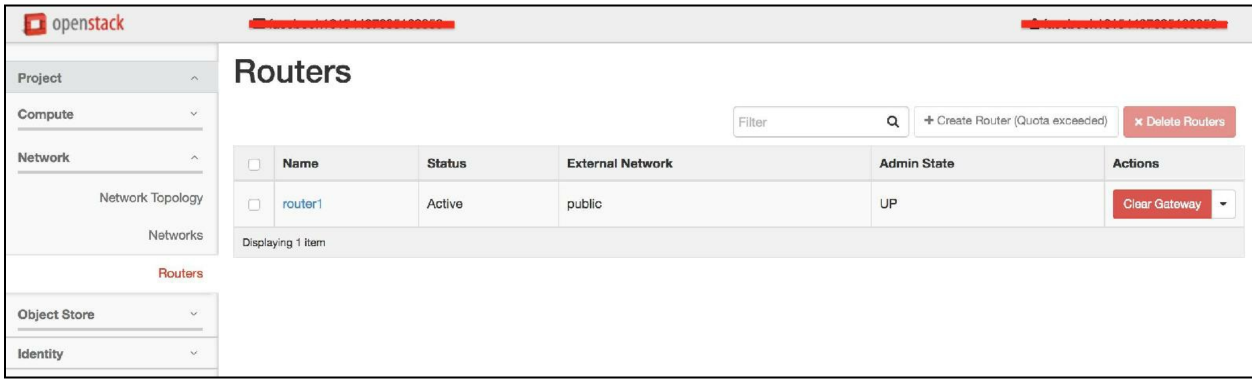
Networks

| | Name | Subnets Associated | Shared | Status | Admin State | Actions |
|---|---|---|---|---|---|---|
| | Internal | 192.168.37.0/24 | No | Active | UP | Edit Network |

Displaying 1 item

TryStack Internal Network

As well as the virtual router created:



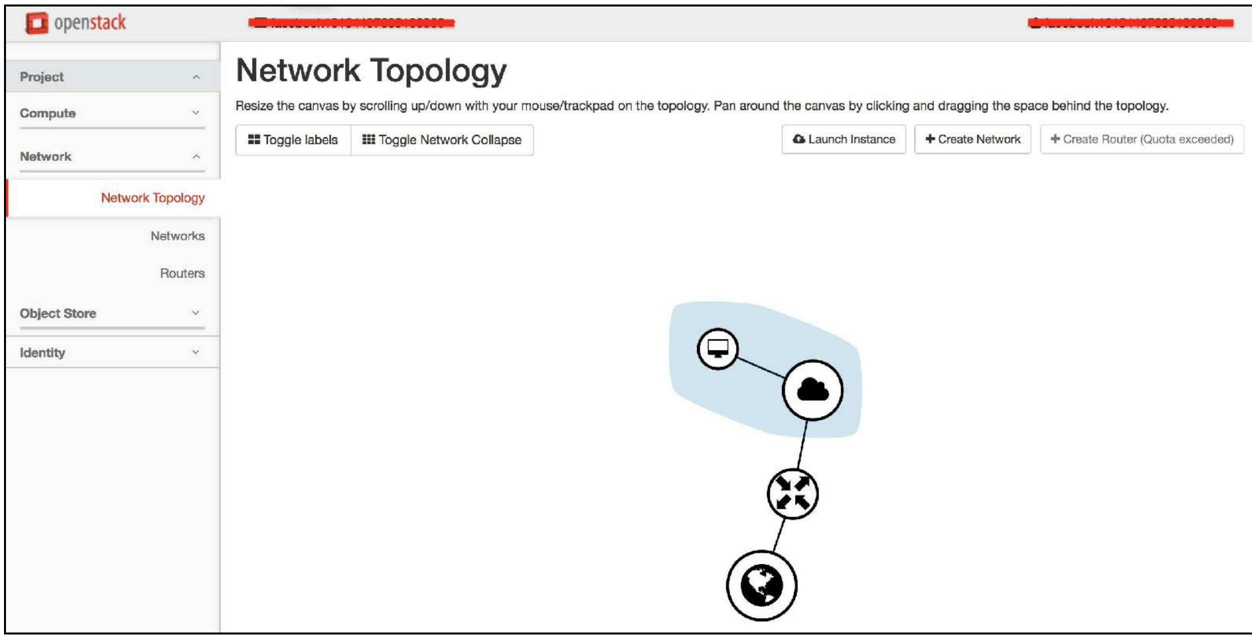
Routers

| | Name | Status | External Network | Admin State | Actions |
|---|---|---|---|---|---|
| ☐ | router1 | Active | public | UP | Clear Gateway ▾ |

Displaying 1 item

TryStack Virtual Router

The network topology shows if the internal network is connected to the router (or not) and the external network:

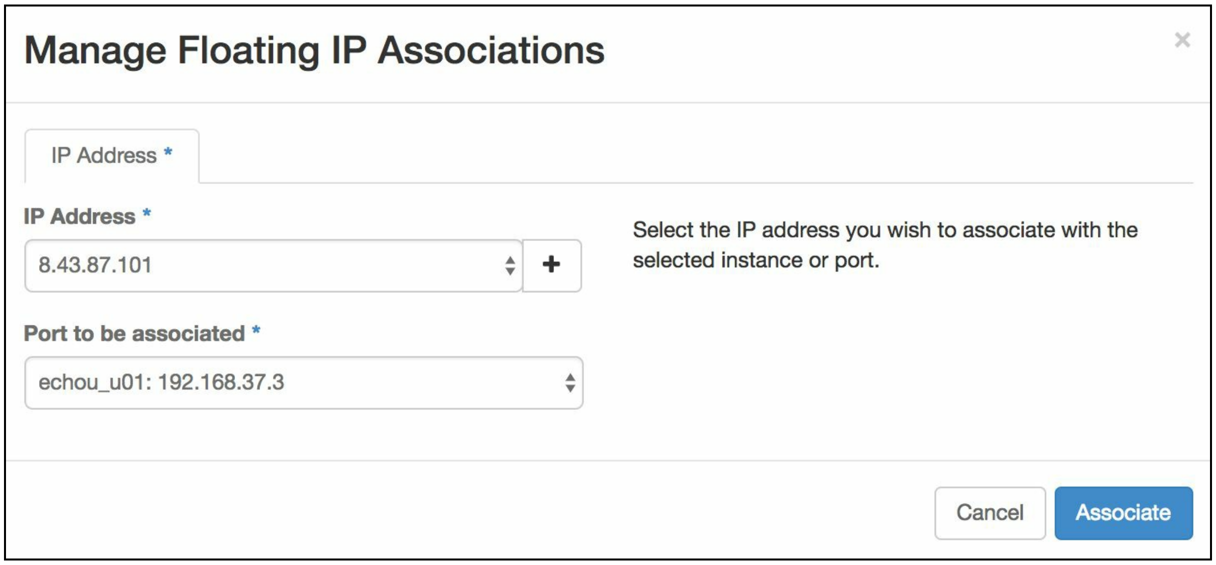


Network Topology

Resize the canvas by scrolling up/down with your mouse/trackpad on the topology. Pan around the canvas by clicking and dragging the space behind the topology.

TryStack Network Topology

As previously discussed, moving from provider network to internal network, a floating IP is required. TryStack will allocate an IP from its public pool:



Manage Floating IP Associations

IP Address *

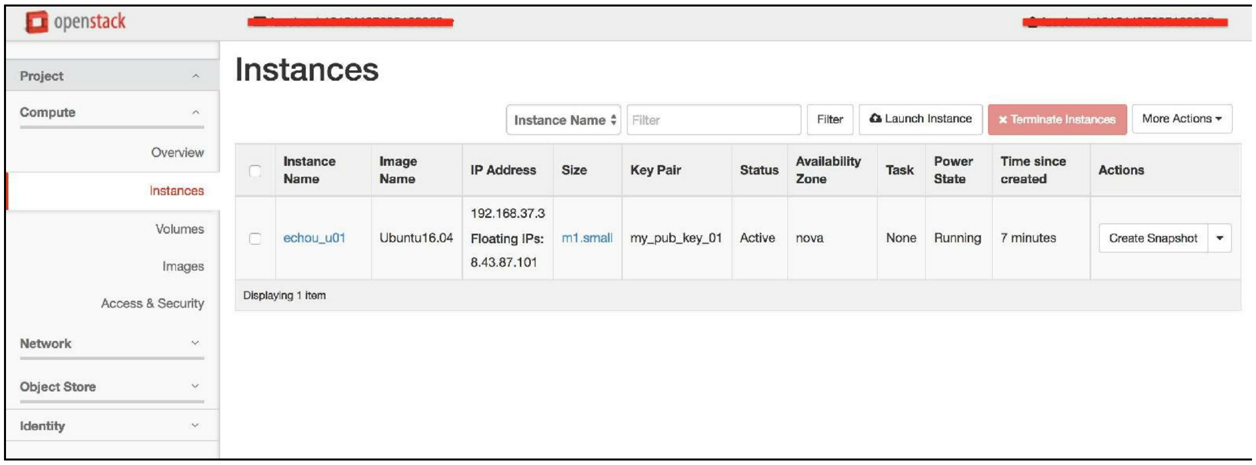
IP Address *: 8.43.87.101

Port to be associated *: echou_u01: 192.168.37.3

Select the IP address you wish to associate with the selected instance or port.

TryStack Floating IP

We can then launch an instance associated with both the internal network as well as the public floating IP:



Instances

| Instance Name | Image Name | IP Address | Size | Key Pair | Status | Availability Zone | Task | Power State | Time since created | Actions |
|---|---|---|---|---|---|---|---|---|---|---|
| echou_u01 | Ubuntu16.04 | 192.168.37.3 Floating IPs: 8.43.87.101 | m1.small | my_pub_key_01 | Active | nova | None | Running | 7 minutes | Create Snapshot |

Displaying 1 item

TryStack Compute Instance

During the instance launch process, you will need to either create or import your public key for security, as well as creating inbound security rules for the security group. In the preceding example, I created an inbound rule for allowing ICMP, SSH, as well as TCP port `8000` to test with Python's HTTP server module. We can now test the host with ICMP and SSH. Note that the following host is able to ping but unable to ssh:

```
echou@ubuntu:~$ ping 8.43.87.101
```

```
PING 8.43.87.101 (8.43.87.101) 56(84) bytes of data.
64 bytes from 8.43.87.101: icmp_seq=1 ttl=128 time=106 ms
64 bytes from 8.43.87.101: icmp_seq=2 ttl=128 time=100 ms
^C
--- 8.43.87.101 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 100.802/103.765/106.728/2.963 ms

# Host can ping but cannot ssh due to key authentication error
echou@ubuntu:~$ ssh ubuntu@8.43.87.101
Permission denied (publickey).
```

We can access the VM with the host using the correct key, install Python (did not come standard with the image), and launch the Python `SimpleHTTPServer` module. You an see next that

```
# Host with correct public key
ssh ubuntu@8.43.87.101
The authenticity of host '8.43.87.101 (8.43.87.101)' can't be established.
ECDSA key fingerprint is
Welcome to Ubuntu 16.04 LTS (GNU/Linux 4.4.0-21-generic x86_64)
...
ubuntu@echou-u01:~$

ubuntu@echou-u01:~$ python2.7 -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
[ip] - - [13/Apr/2017 14:42:25] "GET / HTTP/1.1" 200 -
[ip] - - [13/Apr/2017 14:42:26] code 404, message File not found
[ip] - - [13/Apr/2017 14:42:26] "GET /favicon.ico HTTP/1.1" 404 -
```

You can also install DevStack on a clean machine. If you are open for the Linux flavor, Ubuntu 16.04 is recommended. The installation should be installed with a user stack with root access, without prompting password:

```
$ sudo useradd -s /bin/bash -d /opt/stack -m stack
$ echo "stack ALL=(ALL) NOPASSWD: ALL" | sudo tee /etc/sudoers.d/stack
$ sudo su - stack
$ git clone https://git.openstack.org/openstack-dev/devstack
Cloning into 'devstack'...
..
$ cd devstack/
```

Create the `local.conf` file at the root of Git repo, and start the process:

```
$ cat local.conf
[[local|localrc]]
ADMIN_PASSWORD=secret
DATABASE_PASSWORD=$ADMIN_PASSWORD
RABBIT_PASSWORD=$ADMIN_PASSWORD
SERVICE_PASSWORD=$ADMIN_PASSWORD
```

You can then start the install:

```
# start install
$ ./stack.sh
```

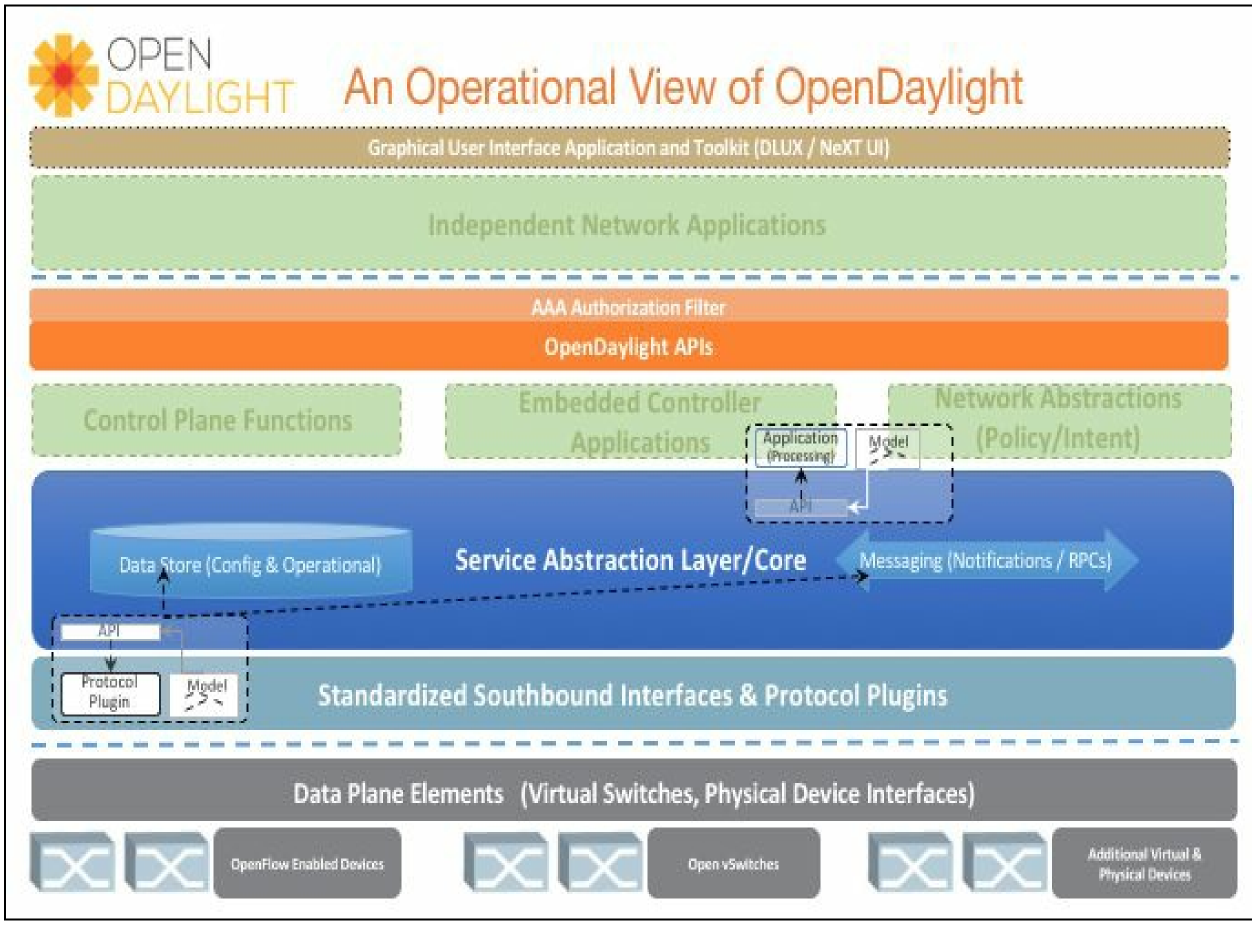
After installation, you can start tinkering with the different project and configuration files:

```
echou@ubuntu:~$ ls /opt/stack/
cinder devstack.subunit horizon neutron requirements
data examples.desktop keystone nova status
devstack glance logs noVNC tempest
echou@ubuntu:~$
```

OpenStack is a cool project that tries to solve a big problem. In this section, we looked at the overall architecture of the project and used two low footprint ways to try out the project, using TryStack and DevStack. In the next section, we will take a look at the OpenDaylight project.

## OpenDaylight

The **OpenDaylight** (**ODL**) project is an open source project founded in 2013 to promote software-defined networking and NFV. The software is written in Java. OpenDaylight supports technology such as OpenFlow, is modular in nature, and tries to enable network services across environments. In the middle of the OpenDaylight architecture is a service abstraction layer with standardized southbound API and protocol plugins, which allows operation with hardware devices such as OpenFlow and various vendor-centric:


OPEN DAYLIGHT
An Operational View of OpenDaylight
Graphical User Interface Application and Toolkit (DLUX / NeXT UI)
Independent Network Applications
AAA Authorization Filter
OpenDaylight APIs
Control Plane Functions
Embedded Controller Applications
Network Abstractions (Policy/Intent)
Application (Processing)
Model
API
Data Store (Config & Operational)
Service Abstraction Layer/Core
Messaging (Notifications / RPCs)
API
Protocol Plugin
Model
Standardized Southbound Interfaces & Protocol Plugins
Data Plane Elements (Virtual Switches, Physical Device Interfaces)
OpenFlow Enabled Devices
Open vSwitches
Additional Virtual & Physical Devices

An Operational View of OpenDaylight (source: https://www.opendaylight.org/platform-overview/)

Also shown in the preceding diagram is the northbound OpenDaylight API. This is normally a REST-based or NetConfig-based API by way of plugins.

In the next section, let's take a look at the ODL project components and the project's programming approach.

# OpenDaylight programming overview

The platform consists of the following essential components:

- **Java**: The Java language is the language ODL is written in. Java interfaces are used for event listening, specifications, and forming patterns.
- **Maven**: This is the build system for Java.
- **Open Service Gateway Initiative** (**OSGi**): This is a Java framework for developing and deploying modular software programs and libraries. It has two parts: the first part is the bundled plug-in and the second part is the **Java Virtual Machine** (**JVM**)-level service registry for bundles to publish, discover, and bind services.
- **Karaf**: This is a light-weight OSGi runtime for loading modules and bundles.
- **Config subsystem**: . This is done with an XML file that is read at runtime.
- **Model-Driven SAL**: **Model-driven Service Adaptation Layer** (**MD-SAL**) is the kernel of the controller platform. This is where different layers and modules are interconnected through pre-defined API.
    - Takes in the YANG data model at runtime and The data is sorted in two buckets: first is config data store that is always kept persistent, and second is operational data store that is ephemeral and typically not changed by RESTCONF.
    - It manages the contracts and state exchange between every application.

ODL takes the model view control approach for its application development in modular fashion. We saw the MVC approach in web API development, where the model is the database, the control is the Python code, and the view is the HTML page or the API output. In ODL, the MCV are broken down as follows:

- **Model**: The model in ODL uses the YANG modeling language, which is also the preferred configuration language for NETCONF that we saw in Chapter 3, *API and Intent-Driven Networking*, we saw how YANG can be used to describe data, in ODL, it is used not only to describe data but also notification to registered listeners and **Remote Procedure Call** (**RPC**) between different modules.
- **View**: The view is the RESTCONF view that is generated automatically.

Let us take a look at an ODL application example.

# OpenDaylight example

OpenDaylight is an important movement in software defined networking that we should at least have a basic understanding of, which is why the topic is included in this book. However, as stated in the overview section, OpenDaylight applications are written in Java, which is outside the scope of this book. Therefore, as with OpenStack examples, we will with the OpenDaylight examples.

The contains two OpenDaylight developer applications under `/home/ubuntu/SDNHub_Opendaylight_Tutorial`, which correspond to the tutorial page on http://sdnhub.org/tutorials/opendaylight/. The tutorial does a good job explaining the fundamentals of ODL programming and the different components involved. We will combine some of the `SDN_Hub` tutorial page and the tutorial on slide share, https://www.slideshare.net/sdnhub/opendaylight-app-development-tutorial/45, for the following steps.

*I find it easier if I increase the memory allocation for the VM to avoid running into memory error later. I increased the memory from 2G to 4G for this section.*

To begin with, let's change to the OpenDaylight directory and make sure we have the latest tutorial code as a matter of good practice:

```
$ cd SDNHub_Opendaylight_Tutorial/
$ git pull --rebase
```

Let us download the Lithium release and start the controller. We will be dropped into the controller shell:

```
$ wget
https://nexus.opendaylight.org/content/groups/public/org/opendaylight/integration/d
karaf/0.3.0-Lithium/distribution-karaf-0.3.0-Lithium.zip
$ unzip distribution-karaf-0.3.0-Lithium.zip
$ cd distribution-karaf-0.3.0-Lithium/
$ ./bin/karaf
karaf: Enabling Java debug options: -Xdebug -Xnoagent -Djava.compiler=NONE -
Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=5005
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=512m;
```

```
support was removed in 8.0
Listening for transport dt_socket at address: 5005

 ________ ________ .__ .__ .__ __
 \_____ \ _______ _____ _____ \______ \ _____ ___.__.| | |__| ____ | |___/ |_
 / | \\____ \_/ __ \ / \ | | \\__ \< | || | | |/ ___\| | \ __\
 / | \ |_> > ___/| | \| ` \/ __ \\___ || |_| / /_/ > Y \ |
 \_______ / __/ \___ >___| /_______ (____ / ____||____/__\___ /|___| /__|
 \/|__| \/ \/ \/ \/\/ /_____/ \/


Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown OpenDaylight.

opendaylight-user@root>
```

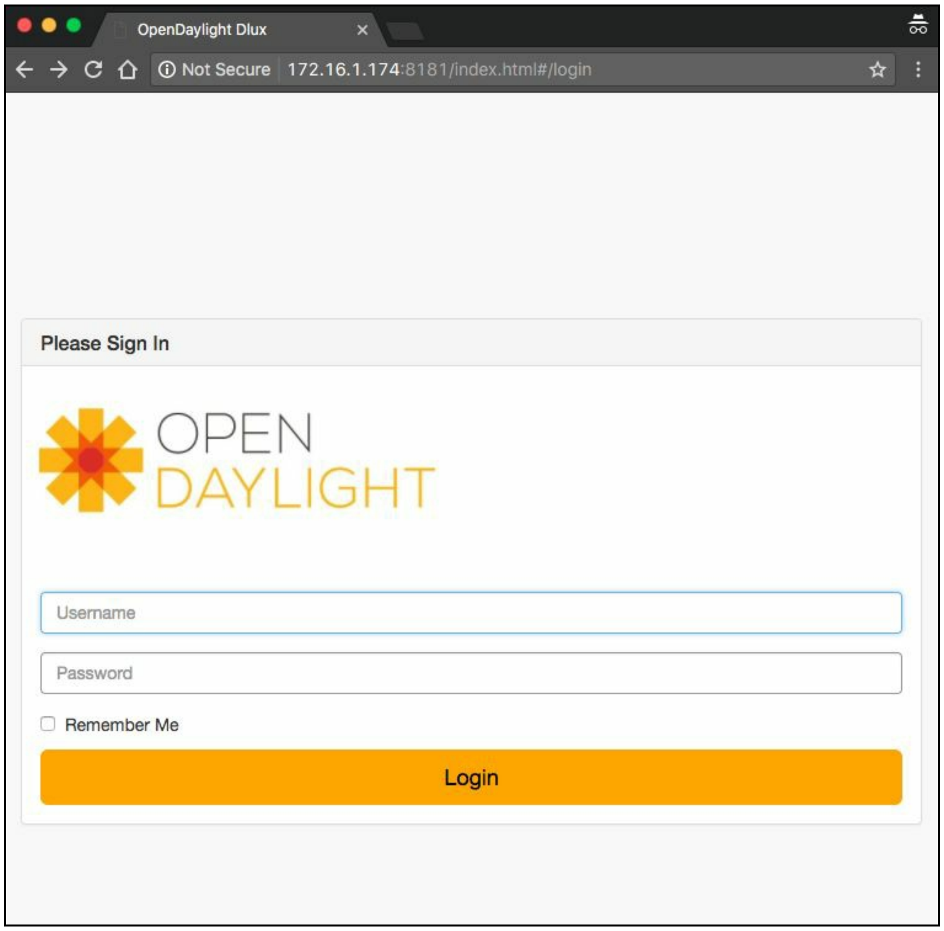
We can use `help` and `<tab>` for help on the available commands.

```
opendaylight-user@root>feature:list
Name | Version | Installed | Repository | Description
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
-------
framework-security | 3.0.3 | | standard-3.0.3 | OSGi Security for Karaf
standard | 3.0.3 | x | standard-3.0.3 | Karaf standard feature
aries-annotation | 3.0.3 | | standard-3.0.3 | Aries Annotations
wrapper | 3.0.3 | | standard-3.0.3 | Provide OS integration
...
```

We can install the UI, OpenFlow southbound plugin, and L2 switch:

```
opendaylight-user@root>feature:install odl-dlux-core
opendaylight-user@root>feature:install odl-openflowplugin-all
opendaylight-user@root>feature:install odl-l2switch-all
```

With the UI installed, we can connect to the UI at http://<ip>:8181/index.html. The default username and password is `admin` / `admin`:

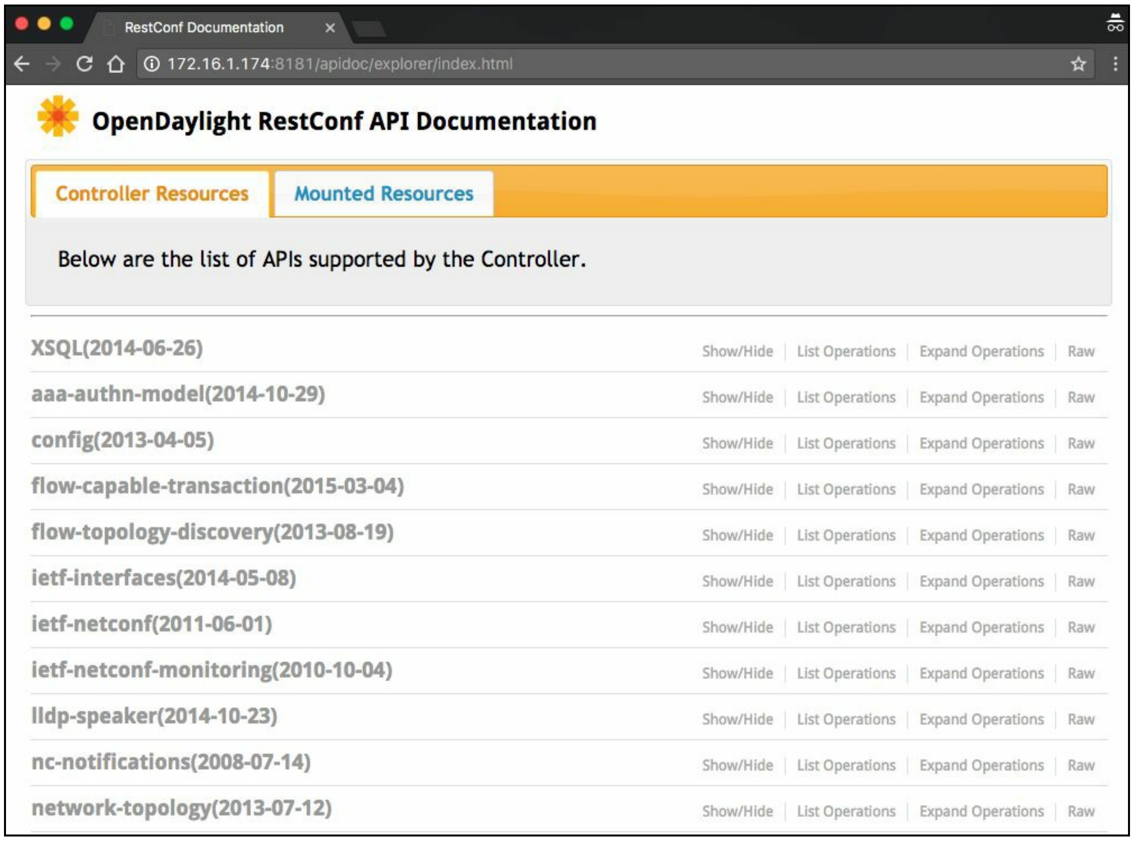


OpenDaylight UI

The default dashboard is pretty boring at this point:

```
opendaylight-user@root>feature:install odl-restconf
opendaylight-user@root>feature:install odl-mdsal-apidocs
```

The API doc can be accessed via `http://<ip>:8181/apidoc/explorer/index.html`:



OpenDaylight RestConf API Documentation

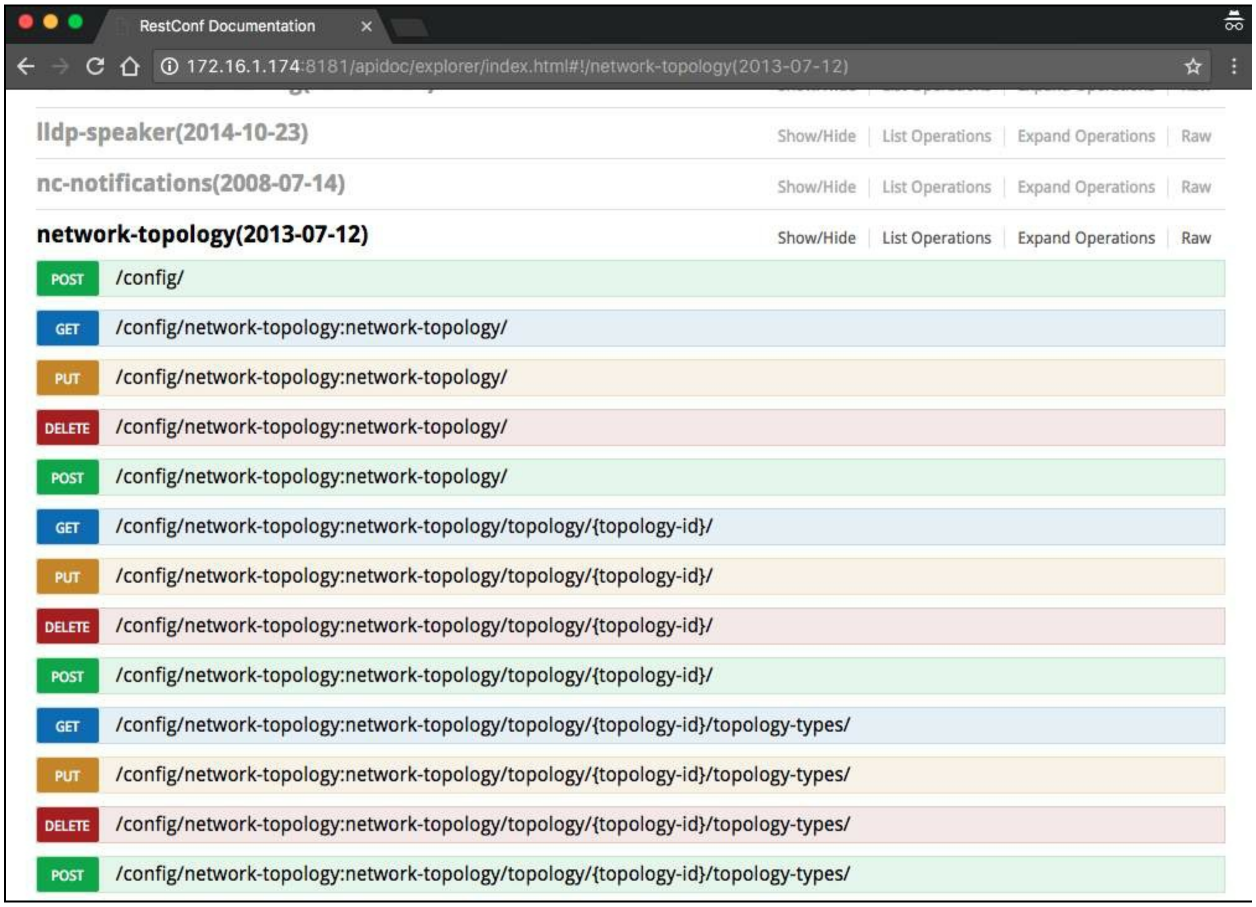
Controller Resources | Mounted Resources

Below are the list of APIs supported by the Controller.

| | | | | |
|---|---|---|---|---|
| XSQL(2014-06-26) | Show/Hide | List Operations | Expand Operations | Raw |
| aaa-authn-model(2014-10-29) | Show/Hide | List Operations | Expand Operations | Raw |
| config(2013-04-05) | Show/Hide | List Operations | Expand Operations | Raw |
| flow-capable-transaction(2015-03-04) | Show/Hide | List Operations | Expand Operations | Raw |
| flow-topology-discovery(2013-08-19) | Show/Hide | List Operations | Expand Operations | Raw |
| ietf-interfaces(2014-05-08) | Show/Hide | List Operations | Expand Operations | Raw |
| ietf-netconf(2011-06-01) | Show/Hide | List Operations | Expand Operations | Raw |
| ietf-netconf-monitoring(2010-10-04) | Show/Hide | List Operations | Expand Operations | Raw |
| lldp-speaker(2014-10-23) | Show/Hide | List Operations | Expand Operations | Raw |
| nc-notifications(2008-07-14) | Show/Hide | List Operations | Expand Operations | Raw |
| network-topology(2013-07-12) | Show/Hide | List Operations | Expand Operations | Raw |

ODL API Explorer

Each of the APIs can be drilled down further:


RestConf Documentation

172.16.1.174:8181/apidoc/explorer/index.html#!/network-topology(2013-07-12)

lldp-speaker(2014-10-23) Show/Hide | List Operations | Expand Operations | Raw

nc-notifications(2008-07-14) Show/Hide | List Operations | Expand Operations | Raw

network-topology(2013-07-12) Show/Hide | List Operations | Expand Operations | Raw

POST /config/
GET /config/network-topology:network-topology/
PUT /config/network-topology:network-topology/
DELETE /config/network-topology:network-topology/
POST /config/network-topology:network-topology/
GET /config/network-topology:network-topology/topology/{topology-id}/
PUT /config/network-topology:network-topology/topology/{topology-id}/
DELETE /config/network-topology:network-topology/topology/{topology-id}/
POST /config/network-topology:network-topology/topology/{topology-id}/
GET /config/network-topology:network-topology/topology/{topology-id}/topology-types/
PUT /config/network-topology:network-topology/topology/{topology-id}/topology-types/
DELETE /config/network-topology:network-topology/topology/{topology-id}/topology-types/
POST /config/network-topology:network-topology/topology/{topology-id}/topology-types/

ODL API Drilldown

Let's shutdown the controller:

```
opendaylight-user@root>system:shutdown
```

We will now switch to the tutorial directory `/home/ubuntu/SDNHub_Opendaylight_Tutorial/distribution/opendaylight-karaf/target/assembly` and restart the controller. Note the banner of SDNHub:

```
ubuntu@sdnhubvm:~/SDNHub_Opendaylight_Tutorial/distribution/opendaylight-
karaf/target/assembly[18:26] (master)$ ./bin/karaf
karaf: Enabling Java debug options: -Xdebug -Xnoagent -Djava.compiler=NONE -
Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=5005
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=512m;
support was removed in 8.0
Listening for transport dt_socket at address: 5005

  _____ _____  _   _ _    _       _
 / ____|  __ \| \ | | |  | |     | |
| (___ | |  | |  \| | |__| |_   _| |__
 \___ \| |  | | . ` |  __  | | | | '_ \
 ____) | |__| | |\  | |  | | |_| | |_) |
|_____/|_____/|_| \_|_|  |_|\__,_|_.__/
```

```
Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown OpenDaylight.

opendaylight-user@root>
```

We will install the learning-switch, OpenFlow plugin, and RESTCONF:

```
opendaylight-user@root>feature:install sdnhub-tutorial-learning-switch
opendaylight-user@root>feature:install odl-openflowplugin-all
opendaylight-user@root>feature:install odl-restconf
```

In a separate window, let's start the Mininet single switch with three hosts topology:

```
$ sudo mn --topo single,3 --mac --switch ovsk --controller remote
```

Of course, `h1` is unable to ping `h2`:

```
mininet> h1 ping -c2 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable

--- 10.0.0.2 ping statistics ---
2 packets transmitted, 0 received, +2 errors, 100% packet loss, time 999ms
pipe 2
```

It is because although we have installed the SDNHub's switch application, we have not installed the flow-miss entry for the controller. Let's do that:

```
$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
$ ovs-ofctl add-flow tcp:127.0.0.1:6634 -OOpenFlow13
priority=1,action=output:controller
$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=21.407s, table=0, n_packets=8, n_bytes=560, idle_age=2,
priority=1 actions=CONTROLLER:65535
```

Now we can see the ping packets go successfully from `h1` to `h2`:

```
mininet> h1 ping -c 2 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=21.5 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=3.49 ms

--- 10.0.0.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 3.493/12.523/21.554/9.031 ms
```

Similar to what we did earlier with the Ryu controller, to move forward with the programming, we need to start modifying the `TutorialL2Fowarding.java` file under `/home/ubuntu/SDNHub_Opendaylight_Tutorial/learning-switch/implementation/src/main/java/org/sdnhub/odl/tutorial/learningswitch/impl`. As mentioned earlier, going over Java construct is out of scope for this book:

```
$ pwd
/home/ubuntu/SDNHub_Opendaylight_Tutorial/learning-
switch/implementation/src/main/java/org/sdnhub/odl/tutorial/learningswitch/impl
$ ls
TutorialL2Forwarding.java*
TutorialL2Forwarding.single_switch.solution*
TutorialLearningSwitchModuleFactory.java*
TutorialLearningSwitchModule.java*
```

OpenDaylight is a important project in the SDN community. As Python-focused network engineers, even if we do not write Java-based controller applications, we can still familiarize ourselves with the basic understanding of OpenDaylight and how the pieces fit in at the high level. As a consumer of the service, we can always use the northbound REST API to interact with a ready-made application via Python, like we have done a number of times in the previous chapters.

## Summary

In this chapter, we learnt about the different components of OpenStack and OpenDaylight. We started with an overview of OpenStack and its different components, such as networking, identity, storage, and other core components.

OpenStack is a broad, complex project, aimed at for a virtualized datacenter. Therefore, we kept our focus on Networking in OpenStack and the different Plugins that interact with the physical network. In the section that followed, we used DevStack and TryStack to look at examples of OpenStack.
The OpenDaylight project is an open source project founded in 2013 to promote software-defined networking and network functions virtualization. The project provides an abstraction layer that provides both northbound and southbound interfaces, which allows multi-vendor compatibility. We took a high level view of the components of OpenDaylight, as well as using the Virtual Machine to construct a Hub application using the OpenDaylight controller.
In the next chapter, we will combine all our knowledge from the previous chapters and examine SDN in the real world; we'll explore how we can move from a legacy-based network into perhaps an SDN-based, flexible, and programmable network.

# Hybrid SDN

It seems that software defined networkingÂ is on every network engineer's mind these days, and rightfully so. Ever since the introduction of OpenFlow in 2010, we have seen steady news on traditional networks transition to a software-defined network. The news typically focuses on the infrastructure agility as competitive advantage that the change brings. Many high profile SDN startups were formed offering new network services, such as SD-WAN. In the slower paced standards bodies,Â SDN standards were gradually being ratified. In the marketplace, vendors such as Quanta and Pica8 started to join forces in making carrier-grade hardware that was de-coupled from software. The combination of results is a seemingly new SDN world, just waiting around the corner for all networks to transition to. However, the reality is a bit different. Despite all the progress SDNÂ have made, the technologyÂ deployment seems to be biasedÂ toward the very large, some might call Hyperscale, networks. Higher education research networks such as Internet2 and companies such as Google, Facebook, and Microsoft, all publicly release their SDN-driven projects, with many of them in detail and open source. But in the mid-tier service provider and enterprise space, there still remains to be a holding pattern.Â

What are the reasons causing this disparity? Every network is different, but let's take a look at some of the generalized reasons for not adapting SDN, and perhaps some counter arguments. First let's look at the argument ofÂ **My network is too small to utilize the benefits of SDN**.

It does not take a mathematician to figure out that larger networks have more network gears, and therefore can reap more benefits if they can save money or increase flexibility by moving to SDN. Each network is different and the degree of competitive advantage SDN can bring varies. However, if we believe that SDNÂ is the correct path for the future, it is never too early to start planning.

In this chapter, I will offer some steps for the migration. These steps do not need to be executed at once; in fact, many of them need to be executed sequentially. But the steps, such as standardizing your network, will help you gain some benefits today. The second argument against SDN migration might be: **The technology is not mature enough for production use**.Â

It is true that the SDN technology is moving at a faster pace than its more mature counterparts, and being on the cutting edge can sometimes put you on the bleeding edge. It is no fun to commit to a technology only to replace it later. In this chapter, I wish to offer you some thoughts about segmenting parts of your network, so you can safely evaluate the technology without putting your production network at risk. We see that both small and large networks can benefit from SDN, and segmenting large networks into smaller networks would be beneficial. A possible third argument against SDN migration might be:Â **I do not have enough engineering resources to invest in SDN**.

The SDN projects, such as OpenFlow, have come a long way since their humble beginning as Stanford University research projects. The fact that you are reading this chapter might indicate that even though there are only limited resources, you do believe that there is some value in investing the time to learn the technology. Each technology is more or less an evolution rather than revolution process; the knowledge you invest time in learning today, will serve as a foundation for newer technology tomorrow, even if you do not use it today. Finally, one might argue that **I do not have a transition plan to move to SDN**.Â

In this chapter, I wish to address the transition path by calling the network a hybrid SDN network. We might have an aspiration to move from a traditional, vendor driven network to an automated software defined network, but the truth of the matter is that the migration will take time, effort, and monetary investment. During that transition time, the network still needs to be functional and the light still needs to be on. In this chapter, I will use OpenFlow and Ryu controller as an example of an SDN network that we eventually want to migrate to. We will look at some of the planning steps, technology, and points of discussion for a successful migration. I wish to offer some generalized suggestions and examples that might come in handy for your own network. If you are wondering about a successful migration, you need to look no further than Google. At the *Open Networking Summit* of 2017, the typically secretive Google detailed their migration from traditional network to OpenFlow-based WAN at G-Scale (http://opennetsummit.org/archives/apr12/hoelzle-tue-openflow.pdf). By some estimate in 2010, if Google was an ISP, it would be the second largest in the world. Very few of us operate at that scale, but it is comforting to know that migration is possible without disruption even for a network as large as Google's.Â

*I picked OpenFlow and Ryu controller because of the combination of industry support, Python subject matter for this book, and personal preference.Â*

In particular, this chapter will focus on the following:Â

- Preparing the network for SDN and OpenFlow
- Greenfield deployment considerations
- Controller redundancy
- BGP interoperationÂ
- Monitoring integration
- Controller to switch secure TLS connectionÂ
- Physical switch selection considerationsÂ

By the end of the chapter, you should be ready to take the next steps in bringing your traditional network into a softwareÂ defined network.Â

## Preparing the network

Before we dive into the SDN network, we should take a moment to prepare our network. For example, as we have seen with the various examples in the book, network automation works because it offloads redundant, boring tasks with computer programs (Python in our case). A computer can complete the tasks fast and without mistakes, and is superior to its human counterpart. What a computer program does not do well, amongst other things, is analytics and interpretation of situation. Automation is most effective when our network is standardized with similar functions. The same goes for SDN migration. If we have many parts of the network that are dissimilar that requires constant thinking when making modifications, simply placing a controller within the network will not help us much.

The following are what I deem to be necessary steps that should be taken prior to SDN and OpenFlow migration. The points listed next are subjective in nature with no right or wrong answer, but only a continuum scale that shows relative strength in each area. For the engineer in me, I understand that we are all anxious to get to practical examples. However, I believe the mindset and practice will serve you well on your journey for migration. They are worth the time to think through and incorporate into your strategy.

## Familiarize yourself with the existing framework and tools

Being familiar with existing tools is one of the first steps we can take to prepare our network. As the old saying goes, *we don't know what we don't know*. By studying the existing framework and tools, we can start to evaluate them against our own network needs. For example, if your company already has lots of Java developers, you might want to take a look at Floodlight controller, which is written in Java. If you prefer to have strong vendor support, OpenDaylight projects might have an edge over OpenFlow. Of course, understanding a technology does not mean agreement, but it does give you a solid ground for evaluation of the pros and cons of each. A healthy debate between team members holding different points of view will usually result in better decisions.

Of course, with the speed of technological development, there is practically no way we can be experts in every framework and tool. More importantly, we can't fall into the traps of 'paralysis by analysis', where we over analyse a technology, so much so that we end up not making a decision at all. Personally, I would recommend going relatively deep into something that interests you while having at least a passing interest in similar technologies. For me, I choose to go deeper into Ryu and POX controller for OpenFlow. From there, I am able to compare and contrast the two controllers with other controllers and technologies that perform similar functions. This way, we can evaluate with confidence and yet not spend 100 percent of our time analyzing.

## Network standardization

I am sure standardization is nothing new to network engineers. After all, we rely on the TCP, UDP, ICMP, IP, and many other standards for our network nodes to communicate with each other. Standards give us a way to stand on common ground when comparing different elements. Think of a time you talked to your colleague about an OSI layer 3 technology to start the conversation, so your colleague could draw the right comparison in his or her mind. When we design networks, we know that a standardized topology using similar vendors and protocols helps us troubleshoot, maintain, and grow our network.
What is interesting is that as the network grows in size, so does non-standard configuration and one-off work. A single switch in a wiring closet will start to be daisy-chained when we run out of ports, our once standardized border access-list on devices no longer follows our number schema or preferred way of description when they starts to grow. Have you seen a snowflake under microscope? They are beautiful to look at, but each of them is different. A non-standard 'snowflake' network is probably one routing-loop away from causing an outage.

What is a good standardizing strategy? The truth always stands between a completely organic network and a single standardization. Only you can answer that question. However, over the years I have seen good results for having two or three standardized topologies separated by function, say, core, datacenter, and access. There are probably several non-flexible requirements depending on network functions, for example, the size of routing table in the core network. Therefore, starting with the network function and limit the standard as far as equipment, protocol, and topology generally yields good balance between the need for localization with standardization, in my opinion.

## Create minimum viable products

According to Wikipedia, https://en.wikipedia.org/wiki/Minimum_viable_product, the minimum viable product, or MVP, is *a product with just enough features to satisfy early customers, and to provide feedback for future development*. This is similar to the proof of concept that we are all familiar with. Before we deploy a new technology, we put it in the lab and run tests against it. There is no way to replicate production, but the second best option is to use minimum viable product for production simulation.

In this book, we have been using VIRL and virtual machines to create our own minimum viable network for practice. The beauty of virtual machine and simulation tools, such as VIRL and GNS3, cannot be overstated. Perhaps the best thing about software defined networking is that there are plenty of MVP tools we can use, many of them with very little footprint, such as a virtual machine. We have seen SDNHub's all-in-one VM bundled with Mininet and various controllers. It is easier than ever before for us to properly simulate our future network.

## Relentlessly experiment

We all learn better when we can safely deploy and break things. Time and time again, we know that things usually do not work right out of the box, and we need to spend time and energy to learn and experiment. This is especially true with newer technology such as SDN. By experimenting over and over with the same technology, you also gain muscle memories from repeated tasks. Do you remember the first time you logged in to the Cisco IOS or NX-OS shell? Did that feel strange? How about now after you have logged in perhaps hundreds of times? Often we learn best by doing.

The points I have made in this section might seem basic or common sense, but these are points I hold true throughout the years. I have seen projects not succeeding because some of the points mentioned previously were not followed through.

Perhaps a minimum viable product was not built that would have uncovered some detectable fault, or there was too much variation in the network for the new technology to adapt to. Hopefully, by preparing your network with familiarization of tools and frameworks, standardization, building minimum viable product, and relentlessly experimenting, you can decrease the friction of SDN migration for your own network.
In the next section, we will look at the consideration points for a greenfield SDN deployment.

## Greenfield deployment

A greenfield deployment, or building a network from scratch, is perhaps the easiest in terms of co-exist an SDN OpenFlow network with traditional network. The only interaction between your SDN network to traditional network equipment will almost only be at the edge, which in today's world, almost always uses BGP as the exchange protocol. Your external facing node will be using BGP, such as what we have seen with the Ryu BGP library, while internally you are free to use any type of connectivity and topology you wish.

If you already have a way to isolate your datacenter network with this approach, in the case of a greenfield deployment you can use the same method. If you use a bottom up approach, you can start by calculating the amount of compute resource you need, tally it up to the number of racks, and determine the number of top of rack switches as a starting point. Then calculate the number of over subscription, if any, at each of the aggregation layers, leaves, spines, cores, and such. In a way, this is no different than planning out a traditional network. The only variables in this case would be controller placement and orchestration.

For the controller placement, there have been many studies and use cases published. But the final answer typically resides at the answer to the questions, "How many nodes do you want each controller to manage?" and "How do you want each controller to communicate with each other?" The first question deals with the SLA-level of how big of a failure domain do you feel comfortable with (we will look at controller redundancy later on). The second question typically deals with the type of coordination between the controllers. The coordination can happen externally or simply tied in a statistics feedback loop. For example, the leaf controller can simply know that for certain destination *x* in and ECMP format, it can be reached through ports 1 - 4 of switch 1 as long the utilization on those 4 ports are below 70 percent. Using the OpenFlow port statistics, the controller can easily be alerted if those ports are above the utilization threshold and take the port out of load balance rotation if needed. There are also commercial vendors who can provide this orchestration for open source OpenFlow controllers as well.

## Controller redundancy

In the centralized controlled environment, such as OpenFlow, we need to be prepared for fault tolerance on the controller. This is generally the first thing we need to consider in our deployment. In production network, we need to plan for controller failure or any communication failure between controller and the switch. Prior to OpenFlow 1.3, the orchestration of controllers needed to be done via a third party software or between controllers themselves for negotiating the primary controller. Starting in OpenFlow 1.3, this can be done via the `OFPT_ROLE` messages between the controller and the switch.

In OpenFlow 1.3, the switch can connect to multiple controllers and the controller's role can be equal, master, or slave. In equal role, it is just as if the switch has two masters, each having full access to the switch. In a slave role, the controller has only read access to the switch, does not receive asynchronous messages, and is denied any `flow-mod`, `packet-out`, or other state change messages. In a master role, the controller has full access to the switch; the switch will only have one master controller. In the case of two controllers trying to become the master, the tie-breaker will be on the 64-bit generation ID that is included in the `OFPT_ROLE_REQUEST` messages; the larger value wins.

To have equal role of controllers that a switch connects to, is pretty tricky for coordination. The controllers must be in synchronization with each other to avoid any duplicate flows. So I would recommend having master-slave relationship in your deployment. The number of controllers in a cluster differs from network to network, but it is safe to say that you should have at least one master and one slave. Let's look at an example.

# Multiple controller example

We can launch multiple instances of Ryu OpenFlow 1.3 controllers listening on different ports:

```
$ ryu-manager --verbose ryu/app/simple_switch_13.py
$ ryu-manager --verbose --ofp-tcp-listen-port 5555 ryu/app/simple_switch_13.py
```

I have constructed a Mininet Python topology file, `chapter13_mininet_1.py`, that specifies two controllers with two switches connected to both the controllers. Each switch has two hosts connected to :

```
...
net = Mininet( controller=RemoteController, switch=OVSKernelSwitch )
c1 = net.addController('c1', controller=RemoteController, ip='127.0.0.1',
port=6633)
c2 = net.addController('c2', controller=RemoteController, ip='127.0.0.1',
port=5555)
...
s1 = net.addSwitch('s1')
s2 = net.addSwitch('s2')
...
c1.start()
c2.start()
s1.start([c1,c2])
s2.start([c1,c2])
...
```

We can launch the topology:

```
$ sudo python mastering_python_networking/Chapter13/chapter13_mininet_1.py
...
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
h3 h3-eth0:s2-eth2
h4 h4-eth0:s2-eth3
s1 lo: s1-eth1:h1-eth0 s1-eth2:h2-eth0 s1-eth3:s2-eth1
s2 lo: s2-eth1:s1-eth3 s2-eth2:h3-eth0 s2-eth3:h4-eth0
c1
c2
```

We see that there are actually some duplicated packets on the initial flood:

```
mininet> h1 ping -c 3 h4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=34.6 ms
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=38.4 ms (DUP!)
```

```
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=38.4 ms (DUP!)
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=0.197 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.067 ms

--- 10.0.0.4 ping statistics ---
3 packets transmitted, 3 received, +2 duplicates, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 0.067/22.375/38.480/18.213 ms
mininet>
```

Let's stop the Mininet topology and clean up the links (you should always do this after you stop your Mininet topology when using API):

```
$ sudo mn --clean
```

I have included two Ryu applications, `chapter13_switch_1.py` and `chapter_switch_2.py`, each based on `simple_switch_13.py` that we have seen. However, we made some changes. We added a few more new imports that we will use later on:

```
from ryu.controller import dpset
from ryu.controller.handler import HANDSHAKE_DISPATCHER
import random
```

In the initialization, we included the role list as well as a generation ID:

```
def __init__(self, *args, **kwargs):
    super(SimpleSwitch13, self).__init__(*args, **kwargs)
    self.mac_to_port = {}
    self.gen_id = 0
    self.role_string_list = ['nochange', 'equal', 'master', 'slave', 'unknown']
```

We have added additional methods to catch OpenFlow errors with decorator:

```
@set_ev_cls(ofp_event.EventOFPErrorMsg,
            [HANDSHAKE_DISPATCHER, CONFIG_DISPATCHER, MAIN_DISPATCHER])
def on_error_msg(self, ev):
    msg = ev.msg
    print 'receive a error message: %s' % (msg)
```

The following section is where we create a function to send role requests from the controller to the switch:

```
def send_role_request(self, datapath, role, gen_id):
    ofp_parser = datapath.ofproto_parser
    print 'send a role change request'
    print 'role: %s, gen_id: %d' % (self.role_string_list[role], gen_id)
    msg = ofp_parser.OFPRoleRequest(datapath, role, gen_id)
    datapath.send_msg(msg)
```

Note that in `chapter13_switch_1.py`, we assign the controller to `MASTER`, while in `chapter13_switch_2.py`, we assign it to `SLAVE`:

```
@set_ev_cls(dpset.EventDP, MAIN_DISPATCHER)
def on_dp_change(self, ev):
    if ev.enter:
        dp = ev.dp
        dpid = dp.id
        ofp = dp.ofproto
        ofp_parser = dp.ofproto_parser

        print 'dp entered, id is %s' % (dpid)
        self.send_role_request(dp, ofp.OFPCR_ROLE_MASTER, self.gen_id)
```

We also want to catch the `RoleReply` messages from the switch to confirm that the controller is now either in master or slave role:

```
@set_ev_cls(ofp_event.EventOFPRoleReply, MAIN_DISPATCHER)
def on_role_reply(self, ev):
    msg = ev.msg
    dp = msg.datapath
    ofp = dp.ofproto
    role = msg.role

    # unknown role
    if role < 0 or role > 3:
        role = 4
    print ''
    print 'get a role reply: %s, generation: %d' % (self.role_string_list[role],
msg.generation_id)
```

When we relaunch the Mininet topology, we will be able to see the master and slave assignment and confirmation:

```
# For master
dp entered, id is 1
send a role change request
role: master, gen_id: 0
...
dp entered, id is 2
send a role change request
role: master, gen_id: 0

# For slave
dp entered, id is 1
send a role change request
role: slave, gen_id: 0
...
dp entered, id is 2
send a role change request
role: slave, gen_id: 0
```

The switch will now only send the Packet-In message toward the master, with no duplicated flood:

```
mininet> h1 ping -c3 h4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=23.8 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=0.311 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.059 ms

--- 10.0.0.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 0.059/8.070/23.840/11.151 ms
```

Again, in strategizing for our deployment, the first thing to consider is the number of controllers and their roles. Unfortunately, for both questions, there is no one-size-fits-all formula. You typically make an educated guess in your deployment design, and modify as you gradually improve the work. For general guidance, in some literatures for OpenDaylight and Cisco ACI, they recommend having one master and two slave controllers. If you do not have any preference, that might be a good starting point for controller redundancy.

## BGP migration example

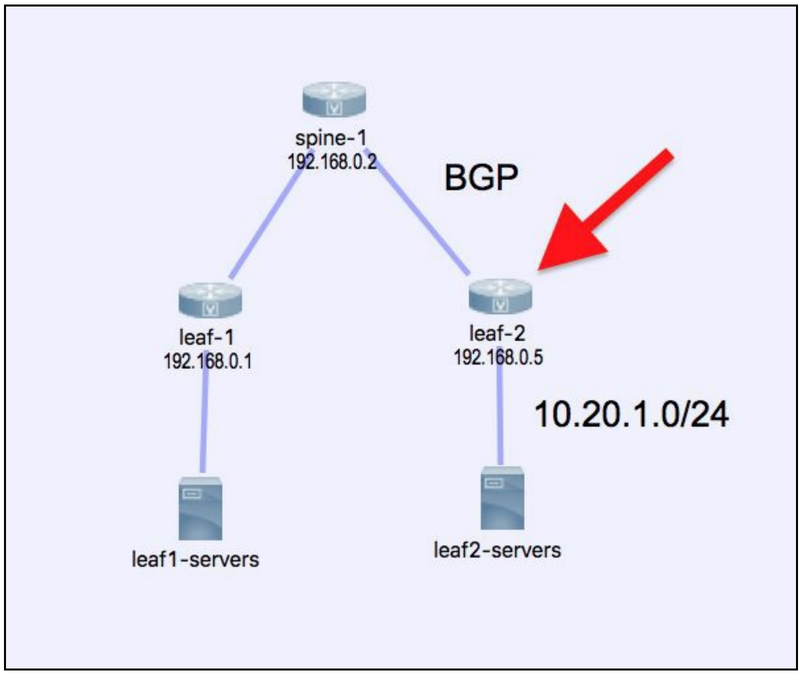
BGP is becoming the protocol of choice in intra and inter autonomous system networking, essentially using the same protocol for both IGP and border. There are many reasons for this. The fact that it is a robust and self-sustaining protocol makes it an ideal candidate for building scaled out network. Not surprisingly, BGP has the maximum support amongst SDN technologies. We have already seen that Ryu has a solid BGP speaker library. Let us use a mock scenario to see how we can construct a migration plan or use it in a hybrid mode.

TIP *RyuBGPDriver is also used as the plugin for BGP Speaker in OpenStack,* *https://docs.openstack.org/developer/neutron-dynamic-routing/functionality/bgp-speaker.html*. *Understanding Ryu BGP usage can possibly help you with OpenStack BGP speaker in the future.*

To begin, let's look at our scenario.

# Migration segmentation

Let us assume that BGP is used between all the following scaled down nodes. The **spine-1** is used as an aggregation router for **leaf-1** and **leaf-2**. Each of the leaf router is top of rack switch that advertises a `/24`; in this case, **leaf-2** is advertising `10.20.1.0/24`. We would like to migrate **leaf-2** from traditional IOS-based device to OpenFlow device with Ryu controller.



Original Topology

Of course, your network might look different than this. This topology was picked because:

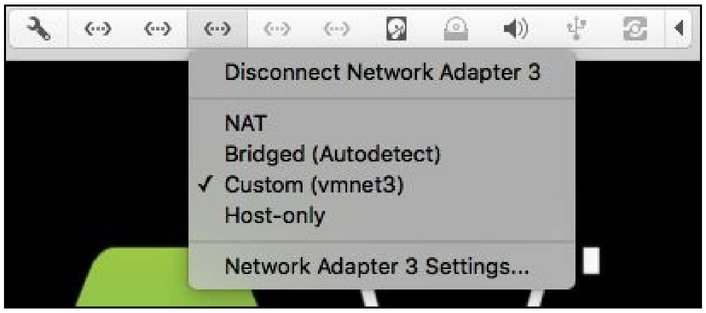
- It is an easy segmentation in your mind. You can easily point to the **leaf-2** rack and think of it as an OpenFlow rack.
- It is a clean swap and is easy to roll back. In other words, you are replacing a physical hardware for the same function. In the event of a rollback, at worst, you can always just take out the new router and put the old one back in.
- The logical addressing is clean. You can be on high alert for any server

reporting error in the `10.20.1.0/24` block upon migration.

Regardless of your scenario, these three points would be my recommendation for picking a spot in your network as a starting point for migration. In the next step, let's see how we can mimic this scenario with Mininet, VIRL, and our virtual machine.

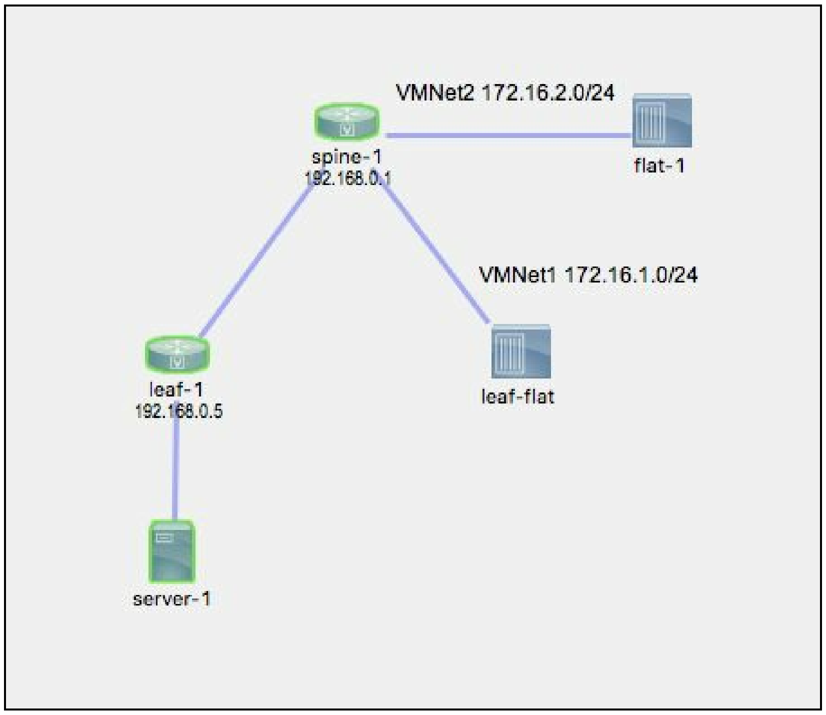
# VIRL and Mininet setup

In our VIRL machine, we will utilize the two flat networks presented by default. We have been using the flat network connected to VMNet2 on `172.16.1.0/24` as the management network. We will connect a third virtual Ethernet interface to VMnet3 on `172.16.2.0/24`:



Network Adapter

In our scenario, we will leave the management network as private project network (which we can still use to ssh to), and flat network as our link to the Mininet topology to simulate **leaf-2** while using **flat-1** as the control plane BGP peer to our Ryu BGP speaker.



Simulated Topology

It may seem like an overkill to have two separate controller applications, one for BGP peer and the other for router. The reason for performing the migration this way is because I believe it is useful to see that the two functions can be de-coupled, which is many times not the case for vendor-purchased equipment. Having two separate applications can also be more scalable down the line, if you want to keep adding ToR routers without needing to spin up a new BGP speaker. Finally, we have already worked with the BGP speaker application as well as the router application in Chapter 11, *OpenFlow Advance Topics*. By using the same tool, we can build on what we already know.

Now that we have the lab ready, let's take a look at the configuration we are started with.

# Cisco device configuration

One of the benefits of using VIRL is the auto-configuration generation function. I assume we are all familiar with the basics of BGP syntax; I will not spend too much time on it. Following are short descriptions that will help us in understanding the scenario.
`spine-1` and `leaf-1` are EBGP peers; `spine-1` in ASN1 while `leaf-1` in ASN2:

```
spine-1#sh ip bgp summary
BGP router identifier 192.168.0.1, local AS number 1
...
Neighbor V AS MsgRcvd MsgSent TblVer InQ OutQ Up/Down State/PfxRcd
10.0.128.2 4 2 262 264 30 0 0 03:54:18 2
```

The link between `spine-1` and `leaf-1` is in `10.0.128.0/17`, while the server rack behind `leaf-1` is in the `10.0.0.0/17` network. Spine-1's loopback IP is `192.168.0.1/32` and leaf-1's loopback IP is `192.168.0.5/32`. The most important point is that we can use `192.168.0.1 or 192.168.0.5` to test reachability between our new OpenFlow rack and the two routers. Ultimately, our new OpenFlow rack and Mininet host need to be able to ping the host in `10.0.0.0/17` range, specifically gateway IP `10.0.0.1` on `leaf-1` and IP `10.0.0.2` on the server:

```
spine-1#sh ip route bgp
...
  10.0.0.0/8 is variably subnetted, 4 subnets, 3 masks
B 10.0.0.0/17 [20/0] via 10.0.128.2, 03:56:53
B 10.20.1.0/24 [200/0] via 172.16.1.174, 01:35:14
 192.168.0.0/32 is subnetted, 2 subnets
B 192.168.0.5 [20/0] via 10.0.128.2, 03:56:53
```

Before we move forward with any Ryu and Mininet connections, we should always test reachability first. A simple ping sourcing from `spine-1` loopback toward the loopback of `leaf-1` and server should be sufficient:

```
spine-1#ping ip 192.168.0.5 source 192.168.0.1
Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 192.168.0.5, timeout is 2 seconds:
Packet sent with a source address of 192.168.0.1
!!!!!
Success rate is 100 percent (5/5), round-trip min/avg/max = 2/6/13 ms
spine-1#
spine-1#ping ip 10.0.0.2 source 192.168.0.1
Type escape sequence to abort.
```

```
Sending 5, 100-byte ICMP Echos to 10.0.0.2, timeout is 2 seconds:
Packet sent with a source address of 192.168.0.1
!!!!!
Success rate is 100 percent (5/5), round-trip min/avg/max = 3/4/9 ms
```

The last step would be to check the IP address that we received from DHCP in VMNet2 and VMnet3. For production, these will obviously be static, but in our lab, I left it as DHCP IP. In the following output, we can see that the Ryu BGP speaker should communicate to `172.16.2.51`, while the Mininet switch would have an upstream neighbor of `172.16.1.250`:

```
spine-1#sh ip int brief
Interface IP-Address OK? Method Status Protocol
GigabitEthernet0/0 10.255.0.68 YES NVRAM up up
GigabitEthernet0/1 10.0.128.1 YES NVRAM up up
GigabitEthernet0/2 172.16.2.51 YES manual up up
GigabitEthernet0/3 172.16.1.250 YES NVRAM up up
Loopback0 192.168.0.1 YES NVRAM up up [placeholder]
```

Great! Let's get our Ryu BGP speaker ready!

## Ryu BGP speaker

Since this is a pretty involved process, I would recommend opening four different SSH sessions to the OpenFlow virtual machine: one for Ryu speaker, one for Mininet, one for the router application, and one for executing the `ovs-vsctl` commands. The IP to `ssh` to should `NOT` be the VMnet2 IP in the `172.16.1.0/24` range, because we are using that as the data path between `spine-1` and our router, and we will add that interface to `s1` OVS bridge later on.
In order to communicate with the `flat1` VMnet3 network, we need to add the virtual Ethernet interface on the virtual machine as well. For me, this is Ethernet3, and I added the IP of `172.16.2.52` for the interface:

```
ubuntu@sdnhubvm:~/ryu_latest[02:44] (master)$ ifconfig eth3
eth3 Link encap:Ethernet HWaddr 00:0c:29:87:67:c9
 inet addr:172.16.2.52 Bcast:172.16.2.255 Mask:255.255.255.0
```

We will use the BGP configuration, `chapter13_bgp_1.py`, as illustrated next, where we have defined the BGP neighbor first:

```
BGP = {
    'local_as': 1,
    'router_id': '172.16.2.52',
    'neighbors': [
        {
            'address': '172.16.2.51',
            'remote_as': 1,
            'enable_ipv4': True,
            'enable_ipv6': True,
            'enable_vpnv4': True,
            'enable_vpnv6': True
        },
    ],
```

We will also configure the advertised routes:

```
    'routes': [
        # Example of IPv4 prefix
        {
            'prefix': '10.20.1.0/24',
            'next_hop': '172.16.1.174'
        },
        {
            'prefix': '172.16.1.0/24'
        },
        {
            'prefix': '172.16.2.0/24'
        }
    ]
}
```

Note that the BGP speaker is advertising on behalf of the ToR router in Mininet that we will see later in this section; therefore, the next hop is the IP of the Mininet OVS segment IP. We are using iBGP peer to allow us the flexibility to specify non-local next hop without the Cisco device rejecting it. Also, note that because we are using multiple applications on the same host, listening on all ports will be very confusing. Therefore, we have restricted the application interface via the `ofp-listen-host` option in Ryu-manager.
We will start the Ryu BGP speaker using the BGP application that we saw in Chapter 11, *OpenFlow Advance Topics*:

```
$ sudo ryu-manager --verbose --ofp-listen-host 172.16.1.52 --bgp-app-config-file
mastering_python_networking/Chapter13/chapter13_bgp_1.py
ryu/services/protocols/bgp/application.py
```

Following are some useful outputs from Ryu, `spine-1`, and `leaf-1` routers for BGP neighbor relationship establishment that you can look for in your lab:

```
# Ryu
Loading config file mastering_python_networking/Chapter13/chapter13_bgp_1.py...
Loading BGP settings...
Starting BGPSpeaker...
...
Adding route settings: {'prefix': '10.20.1.0/24', 'next_hop': '172.16.1.174'}
...
Peer 172.16.2.51 BGP FSM went from Idle to Connect
Peer 172.16.2.51 BGP FSM went from Connect to OpenSent
Connection to peer: 172.16.2.51 established

# spine-1
*<date> 22:56:39.322: %BGP-5-ADJCHANGE: neighbor 172.16.2.52 Up
spine-1#sh ip route 10.20.1.0
Routing entry for 10.20.1.0/24
 Known via "bgp 1", distance 200, metric 0, type internal
 Last update from 172.16.1.174 00:05:16 ago
 Routing Descriptor Blocks:
 * 172.16.1.174, from 172.16.2.52, 00:05:16 ago
 Route metric is 0, traffic share count is 1
 AS Hops 0
 MPLS label: none

# leaf-1
leaf-1#sh ip route 10.20.1.0
Routing entry for 10.20.1.0/24
 Known via "bgp 2", distance 20, metric 0
 Tag 1, type external
 Last update from 10.0.128.1 00:08:27 ago
 Routing Descriptor Blocks:
 * 10.0.128.1, from 10.0.128.1, 00:08:27 ago
 Route metric is 0, traffic share count is 1
 AS Hops 1
 Route tag 1
 MPLS label: none
leaf-1#
```

We can see that the BGP routes are being advertised correctly across the network. Let's build the Mininet network.

# Mininet and REST Router

The Mininet setup is pretty simple and straight forward, which is the beauty of OpenFlow and SDN. The intelligence is in the controller application. We have already seen most of the setup steps in Chapter 11, *OpenFlow Advance Topics*. I will just quickly show the steps that were taken.

For good measure, we will clean up the Mininet topology:

```
$ sudo mn --clean
```

The Mininet topology is located under `chapter13_mininet_2.py`, where we build the network with one controller and two hosts. Note that we change the default port of the controller to port `5555`:

```
...
net = Mininet( controller=RemoteController, switch=OVSKernelSwitch )
c1 = net.addController('c2', controller=RemoteController, ip='127.0.0.1',
port=5555)

h1 = net.addHost('h1', ip='10.20.1.10/24')
h2 = net.addHost('h2', ip='10.20.1.11/24')
s1 = net.addSwitch('s1')
...
```

We can start the Mininet network:

```
$ sudo python mastering_python_networking/Chapter13/chapter13_mininet_2.py
```

The hosts need to have a default gateway point to our router:

```
mininet> h1 ip route add default via 10.20.1.1
mininet> h2 ip route add default via 10.20.1.1
```

We can use the same `REST_Router` reference application for our lab:

```
$ ryu-manager --verbose --ofp-tcp-listen-port 5555 ryu/app/rest_router.py
```

We will add the addresses the router should listen for and respond to:

```
$ curl -X POST -d '{"address":"10.20.1.1/24"}'
http://localhost:8080/router/0000000000000001
```

```
$ curl -X POST -d '{"address":"172.16.1.174/24"}'
http://localhost:8080/router/0000000000000001
```

We can add the routes to the loopbacks and the server rack connected to `leaf-1`:

```
$ curl -X POST -d '{"destination": "192.168.0.1/32", "gateway": "172.16.1.250"}'
http://localhost:8080/router/0000000000000001

$ curl -X POST -d '{"destination": "192.168.0.5/32", "gateway": "172.16.1.250"}'
http://localhost:8080/router/0000000000000001

$ curl -X POST -d '{"destination": "10.0.0.0/17", "gateway": "172.16.1.250"}'
http://localhost:8080/router/0000000000000001

$ curl -X POST -d '{"destination": "10.0.128.0/17", "gateway": "172.16.1.250"}'
http://localhost:8080/router/0000000000000001
```

The last step regarding our Mininet topology involves putting the interface connecting VMnet2 to OVS bridge, so `spine-1` has visibility to it:

```
$ sudo ovs-vsctl add-port s1 eth1
```

We can verify the Mininet topology via `ovs-vsctl`:

```
$ sudo ovs-vsctl show
873c293e-912d-4067-82ad-d1116d2ad39f
    Bridge "s1"
        Controller "tcp:127.0.0.1:5555"
        fail_mode: secure
        Port "eth1"
            Interface "eth1"
        Port "s1-eth2"
            Interface "s1-eth2"
        Port "s1"
            Interface "s1"
                type: internal
        Port "s1-eth1"
            Interface "s1-eth1"
    ovs_version: "2.3.90"
```

We can also verify the routes and addresses of our `REST_Router`:

```
$ curl http://localhost:8080/router/0000000000000001 | python -m json.tool
...
 "internal_network": [
 {
 "address": [
 {
 "address": "172.16.1.174/24",
 "address_id": 2
 },
 {
```

```
 "address": "10.20.1.1/24",
 "address_id": 1
 }
 ],
...
 "route": [
 {
 "destination": "10.0.128.0/17",
 "gateway": "172.16.1.250",
 "route_id": 4
 },
...
```

After all that work, we are now ready to test the connectivity from our shining new OpenFlow rack to the rest of the network!

## Result and verification

We can now test for connectivity to `spine-1` and `leaf-1`, as well as to the gateway of the `leaf-1` block:

```
# connectivity to spine-1 loopback
mininet> h1 ping -c 3 192.168.0.1
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=254 time=11.0 ms
64 bytes from 192.168.0.1: icmp_seq=2 ttl=254 time=2.39 ms

# connectivity to leaf-1 loopback
mininet> h1 ping -c 3 192.168.0.5
PING 192.168.0.5 (192.168.0.5) 56(84) bytes of data.
64 bytes from 192.168.0.5: icmp_seq=1 ttl=253 time=9.91 ms
64 bytes from 192.168.0.5: icmp_seq=2 ttl=253 time=5.28 ms

# connectivity to leaf-1 server block
mininet> h1 ping -c 3 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=253 time=5.49 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=253 time=5.23 ms
```

Success! In order to establish reachability to the server, we need to add a route from the server to our new rack. This is strictly a VIRL auto-config setting.

The host was configured with a default route to the management network while static routes were added to each of the networks. In the real world, you would not need to do this:

```
cisco@server-1:~$ sudo bash
[sudo] password for cisco:

root@server-1:~# route add -net 10.20.1.0 netmask 255.255.255.0 gw 10.0.0.1
root@server-1:~# route -n
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
0.0.0.0 10.255.0.1 0.0.0.0 UG 0 0 0 eth0
...
10.20.1.0 10.0.0.1 255.255.255.0 UG 0 0 0 eth1
...
```

Now we can reach `server-1` from `h1` in Mininet:

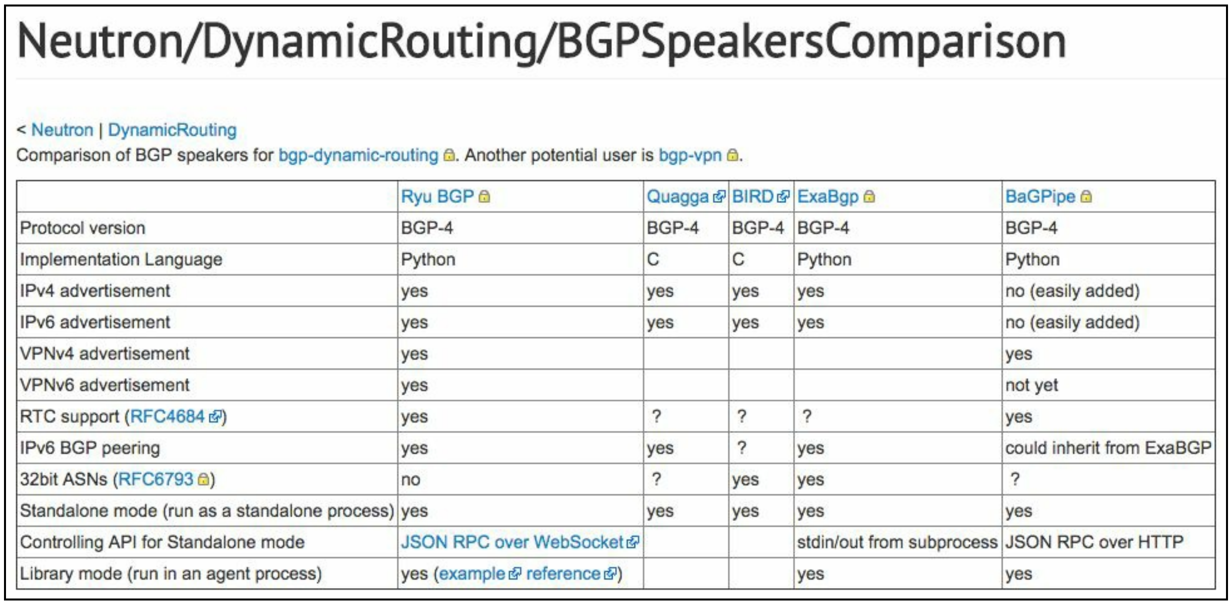
```
mininet> h1 ping -c 3 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
```

```
64 bytes from 10.0.0.2: icmp_seq=1 ttl=61 time=13.3 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=61 time=3.48 ms
...
mininet> h1 ssh cisco@10.0.0.2
cisco@10.0.0.2's password:

Welcome to Ubuntu 14.04.2 LTS (GNU/Linux 3.13.0-52-generic x86_64)

Last login: Thu Apr 20 21:18:13 2017 from 10.20.1.10
cisco@server-1:~$ exit
logout
Connection to 10.0.0.2 closed.
mininet>
```

How cool is that! We now have an OpenFlow SDN rack replacing a traditional gear with all the flexibility and modularity that we could not have achieved otherwise. We are on a roll, so let's look at more BGP examples.

# More BGP example

You can see that BGP is a good protocol of choice to segment your network at a logical location and start your migration process. What if you have a more complicated BGP setup, such as MP-BGP, MPLS, or VxLAN? Would Ryu or other OpenFlow-based BGP speaker be able to provide enough base code so you do not need to re-write lots of code?
We mentioned that Ryu BGP is one of OpenStack's Neutron networking BGP plugins. It speaks to the feature, maturity, and community support for the Ryu project. At the time of writing in Spring of 2017, you can see the BGP speaker comparison of the Neutron project. The Ryu BGP speaker supports MP-BGP, IPv4 and IPv6 peering and VPN.



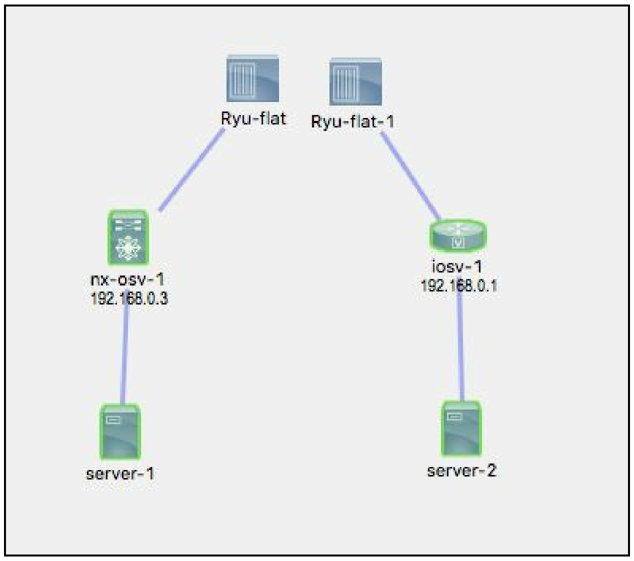
Neutron/DynamicRouting/BGPSpeakersComparison

< Neutron | DynamicRouting
Comparison of BGP speakers for bgp-dynamic-routing. Another potential user is bgp-vpn.

| | Ryu BGP | Quagga | BIRD | ExaBgp | BaGPipe |
|---|---|---|---|---|---|
| Protocol version | BGP-4 | BGP-4 | BGP-4 | BGP-4 | BGP-4 |
| Implementation Language | Python | C | C | Python | Python |
| IPv4 advertisement | yes | yes | yes | yes | no (easily added) |
| IPv6 advertisement | yes | yes | yes | yes | no (easily added) |
| VPNv4 advertisement | yes | | | | yes |
| VPNv6 advertisement | yes | | | | not yet |
| RTC support (RFC4684) | yes | ? | ? | ? | yes |
| IPv6 BGP peering | yes | yes | ? | yes | could inherit from ExaBGP |
| 32bit ASNs (RFC6793) | no | ? | yes | yes | ? |
| Standalone mode (run as a standalone process) | yes | yes | yes | yes | yes |
| Controlling API for Standalone mode | JSON RPC over WebSocket | | | stdin/out from subprocess | JSON RPC over HTTP |
| Library mode (run in an agent process) | yes (example reference) | | | yes | yes |

Neutron BGP Speaker Comparison

The proceeding picture demonstrated the BGP speaker and dynamic routing comparison. Neutron project uses JSON RPC over WebSocket to communicate with Ryu BGP speaker. Let us take a look at an example. In our setup, let us place the speaker at the spine level this time, with the ToR switches being NX-OSv and IOSv devices, as shown here:



Ryu SPINE BGP Speaker

Note that even though the flat network is separated as **Ryu-flat** and **Ryu-flat-1**, they are both connected to VMnet2 on `172.16.1.0/24`. The differentiation in naming is a limitation on VIRL. On the NX-OS, we will add the BGP neighbor:

```
nx-osv-1# sh run | section bgp
feature bgp
router bgp 1
 router-id 192.168.0.3
 address-family ipv4 unicast
 network 192.168.0.3/32
 neighbor 172.16.1.174 remote-as 1
 description iBGP peer Ryu
 address-family ipv4 unicast
 neighbor 192.168.0.1 remote-as 1
 description iBGP peer iosv-1
 update-source loopback0
 address-family ipv4 unicast
```

The IOSv neighbor has similar configuration with slightly different syntax:

```
iosv-1#sh run | section bgp
router bgp 1
 bgp router-id 192.168.0.1
 bgp log-neighbor-changes
 neighbor 172.16.1.174 remote-as 1
 neighbor 192.168.0.3 remote-as 1
 neighbor 192.168.0.3 description iBGP peer nx-osv-1
 neighbor 192.168.0.3 update-source Loopback0
 !
 address-family ipv4
 network 192.168.0.1 mask 255.255.255.255
 neighbor 172.16.1.174 activate
 neighbor 192.168.0.3 activate
 exit-address-family
```

For our example, we will install the `websocket-client` package on the SNDHub virtual machine:

```
$ sudo pip install websocket-client
```

We can re-use the same Mininet topology and router example as well; however, let's focus on the BGP speaker for this exercise. Remember, we are de-coupling the two functions.

We can use the sample code provided with the Ryu package `ryu/services/protocols/bgp/api/jsonrpc.py`, written by *Fujita Tomonori* (https://sourceforge.net/p/ryu/mailman/message/32391914/). On a separate window, launch the BGP application with the `jsonrpc.py` component:

```
$ sudo ryu-manager ryu/services/protocols/bgp/api/jsonrpc.py
ryu/services/protocols/bgp/application.py
```

By default, the `websocket-client` package will install or put a link for `wsdump.y` under `/usr/local/bin`.

```
$ wsdump.py ws://127.0.0.1:8080/bgp/ws
Press Ctrl+C to quit
```

We can launch it for our speaker via the `websocket` call:

```
> {"jsonrpc": "2.0", "id": 1, "method": "core.start", "params" : {"as_number":1,
"router_id":"172.16.1.174"}}
```

We will be able to add the two BGP neighbors, where `172.16.1.109` is the NX-OSv neighbor IP  and `172.16.1.110` is the IOSv neighbor IP:

```
> {"jsonrpc": "2.0", "id": 1, "method": "neighbor.create", "params" :
{"ip_address":"172.16.1.109", "remote_as":1}}
> {"jsonrpc": "2.0", "id": 1, "method": "neighbor.create", "params" :
{"ip_address":"172.16.1.110", "remote_as":1}}
```

We can verify the two BGP neighbor relationship:

```
# NX-OSv
nx-osv-1# sh ip bgp summary

Neighbor V AS MsgRcvd MsgSent TblVer InQ OutQ Up/Down State/PfxRcd
```

```
172.16.1.174 4 1 964 1005 63 0 0 00:00:14 1
192.168.0.1 4 1 51 85 0 0 0 00:44:13 Active

# IOSv
iosv-1#sh ip bgp summary

Neighbor V AS MsgRcvd MsgSent TblVer InQ OutQ Up/Down State/PfxRcd
172.16.1.174 4 1 5 7 30 0 0 00:00:35 1
192.168.0.3 4 1 0 0 1 0 0 00:43:28 Idle
```

Cool! We have used the JSON RPC over WebSocket to configure our BGP speaker. Note that by default, the auto-configuration tries to have the two routers become BGP speakers over the loopback interface. This is normally done over OSPF as the IGP that advertises the loopback. However, in our scenario, OSPF is not enabled on our connection toward the flat network. This offers us a great opportunity to look underneath the hood of the `jsonrpc.py` file.

# Examine the JSONRPC over WebSocket

If we take a look at the `ryu/services/protocols/bgp/api/jsonrpc.py` file, we see that the methods are Python decorators, it calls the various functions from `ryu/services/protocols/bgp`. Let us make a new file under `mastering_python_networking` named `chapter13_jsonrpc_1.py`, by making a new file, we can make our changes safely independent of the original file. This is where we will experiment and make our changes.
We see that the file imports the neighbors module from `rtconf`:

```
from ryu.services.protocols.bgp.rtconf import neighbors
```

From that module, we retrieve the `IP_ADDRESS and REMOTE_AS` attribute that we use when creating the neighbor. We can see in the neighbors section of the configuration that were is also a route reflector client option:

```
IS_ROUTE_REFLECTOR_CLIENT = 'is_route_reflector_client'
```

Aha, from the BGP API documentation (http://ryu.readthedocs.io/en/latest/library_bgp_speaker_ref.html#), we can tell that this is where we can specify our route reflector client for the neighbors. Therefore, we can add the additional attribute when we add our neighbors. We need to cast the value as Boolean, as this is either true or false:

```
@rpc_public('neighbor.create')
def _neighbor_create(self, ip_address='192.168.177.32',
                     remote_as=64513, is_route_reflector_client=False):
    bgp_neighbor = {}
    bgp_neighbor[neighbors.IP_ADDRESS] = str(ip_address)
    bgp_neighbor[neighbors.REMOTE_AS] = remote_as
    bgp_neighbor[neighbors.IS_ROUTE_REFLECTOR_CLIENT] =
bool(is_route_reflector_client)
    call('neighbor.create', **bgp_neighbor)
    return {}
```

We can now relaunch our BGP speaker with our own RPC file:

```
$ sudo ryu-manager mastering_python_networking/Chapter13/chapter13_jsonrpc_1.py
ryu/services/protocols/bgp/application.py
```

Proceed to repeat the initialization process and add the new neighbors with the `route-reflector-client` option:

```
$ wsdump.py ws://127.0.0.1:8080/bgp/ws
Press Ctrl+C to quit
> {"jsonrpc": "2.0", "id": 1, "method": "core.start", "params" : {"as_number":1,
"router_id":"172.16.1.174"}}
< {"jsonrpc": "2.0", "id": 1, "result": {}}
> {"jsonrpc": "2.0", "id": 1, "method": "neighbor.create", "params":
{"ip_address":"172.16.1.109", "remote_as":1, "is_route_reflector_client": "True"}}
< {"jsonrpc": "2.0", "id": 1, "result": {}}
> {"jsonrpc": "2.0", "id": 1, "method": "neighbor.create", "params":
{"ip_address":"172.16.1.110", "remote_as":1, "is_route_reflector_client": "True"}}
< {"jsonrpc": "2.0", "id": 1, "result": {}}
```

We can see the two leaf routers have established the BGP relationship over the loopback interface:

```
iosv-1#sh ip bgp summary

Neighbor V AS MsgRcvd MsgSent TblVer InQ OutQ Up/Down State/PfxRcd
172.16.1.174 4 1 9 11 32 0 0 00:01:18 1
192.168.0.3 4 1 5 4 32 0 0 00:00:48 1

iosv-1#sh ip bgp summary

Neighbor V AS MsgRcvd MsgSent TblVer InQ OutQ Up/Down State/PfxRcd
172.16.1.174 4 1 9 11 32 0 0 00:01:18 1
192.168.0.3 4 1 5 4 32 0 0 00:00:48 1
```

This example demonstrates how we can take an example code and make adjustments by reading into the code and the Ryu documentation for the APIs. For more advanced BGP speaker examples, you can check Toshiki Tsuboi's SimpleRouter for Raspberry Pi 2 (https://github.com/ttsubo/simpleRouter), where he uses Raspberry Pi 2 devices and Ryu controller to establish MP-BGP for VPNv4 over external BGP devices. You can also refer to the Ryu-SDN-IP project by Takeshi Tseng (https://github.com/sdnds-tw/Ryu-SDN-IP), where he makes a Ryu version of the SDN-IP project from ONOS.

## Monitoring integration

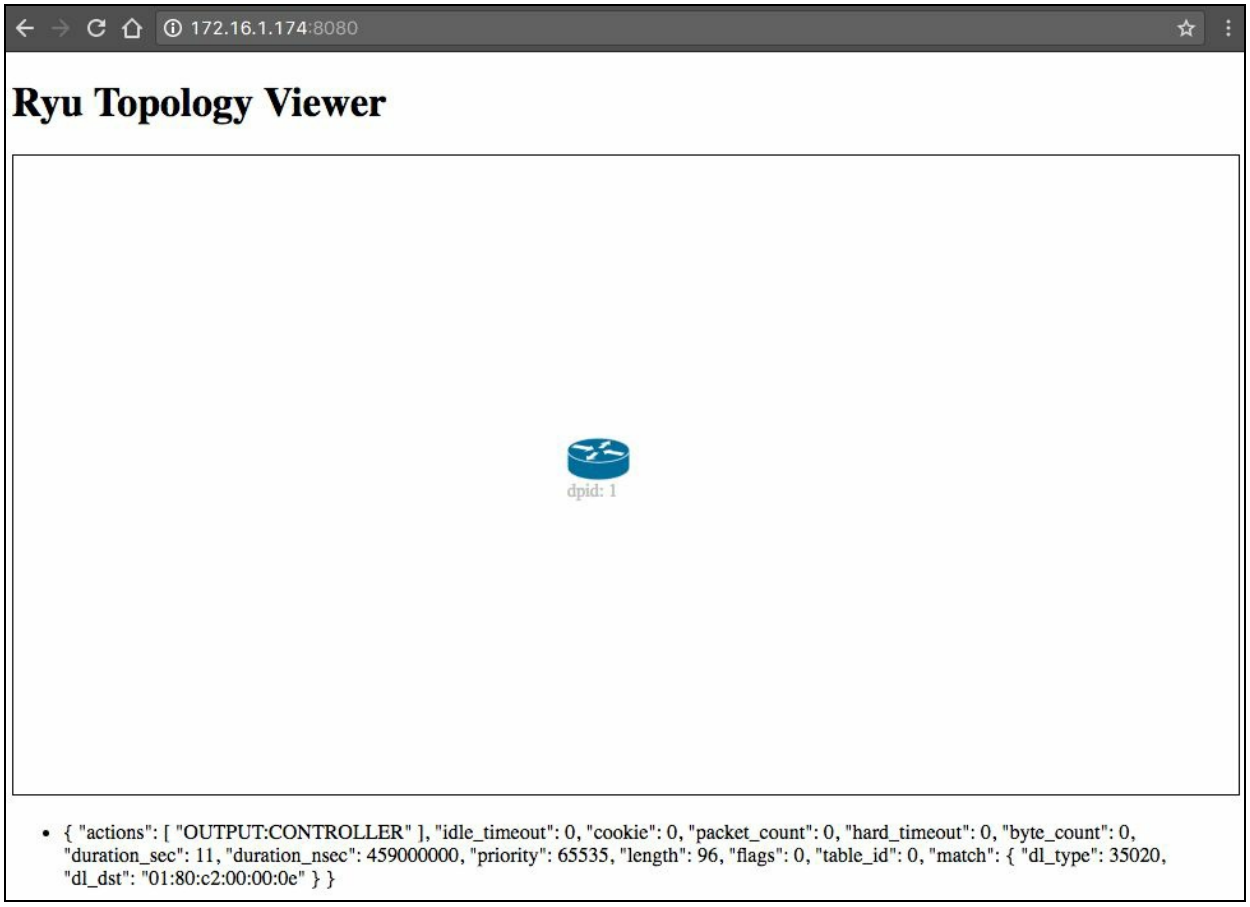
As you start to migrate more devices toward OpenFlow, it becomes increasingly important to monitor the links. We discussed various monitoring tool options in Chapter 7, *Network Monitoring with Python – Part 1* and Chapter 8, *Network Monitoring with Python – Part 2*. Next we can easily integrate monitoring for the migrated devices. We will show the built-in topology viewer that came with Ryu (http://ryu.readthedocs.io/en/latest/gui.html). For easier demonstration, I have used the same Mininet topology we saw earlier in the chapter, but have changed the controller port back to `6633` in `chapter13_mininet_3.py`:

```
net = Mininet( controller=RemoteController, switch=OVSKernelSwitch )
c1 = net.addController('c2', controller=RemoteController,
 ip='127.0.0.1', port=6633)
```

Launch the Mininet topology and the Gui Topology application:

```
# Mininet Topology
$ sudo python mastering_python_networking/Chapter13/chapter13_mininet_3.py

# Ryu Application
$ PYTHONPATH=. ./bin/ryu run --observe-links ryu/app/gui_topology/gui_topology.py
```

Point your browser to `http://<ip>:8080` and you will be able to see the switch along with flow statistics, as shown here:



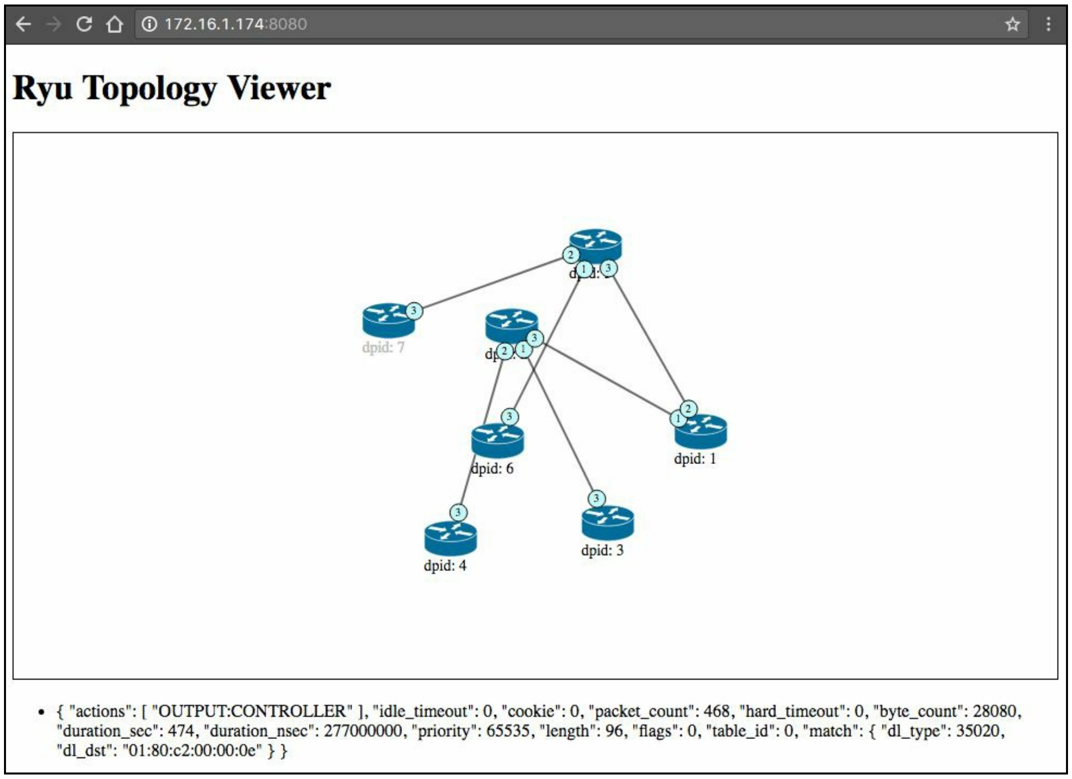
- { "actions": [ "OUTPUT:CONTROLLER" ], "idle_timeout": 0, "cookie": 0, "packet_count": 0, "hard_timeout": 0, "byte_count": 0, "duration_sec": 11, "duration_nsec": 459000000, "priority": 65535, "length": 96, "flags": 0, "table_id": 0, "match": { "dl_type": 35020, "dl_dst": "01:80:c2:00:00:0e" } }

Gui Topology Viewer 1

For multi-device topology testing, you can launch the 8-switch reference topology in the reference example:

```
$ sudo mn --controller remote --topo tree,depth=3
...
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7
...
```

You will be able to point to different devices, and drag and drop to move them around:



```
{ "actions": [ "OUTPUT:CONTROLLER" ], "idle_timeout": 0, "cookie": 0, "packet_count": 468, "hard_timeout": 0, "byte_count": 28080,
"duration_sec": 474, "duration_nsec": 277000000, "priority": 65535, "length": 96, "flags": 0, "table_id": 0, "match": { "dl_type": 35020,
"dl_dst": "01:80:c2:00:00:0e" } }
```

Gui Topology Viewer 2

The graph can be placed side-by-side with your existing monitoring tool or integrated into your existing monitoring tool. Let's talk about encrypting the message between the controller and the switch in the next section.

# Secure TLS connection

At this point, you have a pretty good migration story, having already developed your application to work with existing infrastructure, and on your way to flip more devices. You might be wondering if you should encrypt the messages between the controller and the network devices. In fact, you might be wondering why we waited this long to discuss this topic.

There are several reasons:

- During development and initial migration, you want to have as little moving variable as possible. It is difficult enough to do something new; you do not need to worry about encryption if you don't have to.
- Sometimes for troubleshooting, you might not have a switch that can capture packets natively or the switch not having enough buffer to perform verbose captures. You would be required to do tcpdump outside of your two endpoints.
- You may already have your own PKI infrastructure, if that is the case, please follow your own cert generation and steps for signing by your own certificate authority.

Having said that, we are here, so let's go over the steps to have the Open vSwitch communicating with Ryu controller over SSL.

*You can consult the Ryu document on setting up TLS connection (http://ryu.readthedocs.io/en/latest/tls.html) and the article on configure Open vSwitch for SSL (http://openvswitch.org/support/dist-docs-2.5/INSTALL.SSL.md.html). I combined the steps from those two articles to be more specific to our SDNHub virtual machine. For example, we do not need to initialize the PKI script and `cacert.pem` is under `/var/lib/openvswitch/pki/controllerca/ instead` of `/usr/local/var/lib/openvswitch/pki/controllerca`.*

Let's change to the Open vSwitch directory on our VM and create a switch private key and certificate:

```
$ cd /etc/openvswitch
$ sudo ovs-pki req+sign sc switch
```

Let's also create the controller private key and certificate in the same directory:

```
$ sudo ovs-pki req+sign ctl controller
```

I have created the Mininet topology that is similar to the other topology, with the exception of the following line to specify SSL connection for the controller:

```
c1 = net.addController('c0')
...
s1 = net.addSwitch('s1')
...
s1.cmd('ovs-vsctl set-controller s1 ssl:127.0.0.1:6633')
```

We can launch the Mininet topology:

```
$ sudo python mastering_python_networking/Chapter13/chapter13_mininet_4.py
```

We will launch the controller with the private key and cert options:

```
$ sudo ryu-manager --ctl-privkey /etc/openvswitch/ctl-privkey.pem --ctl-cert
/etc/openvswitch/ctl-cert.pem --ca-certs
/var/lib/openvswitch/pki/switchca/cacert.pem --verbose ryu/app/simple_switch_13.py
```

You should be able to see the SSL message on the Ryu console. You can also verify using the `ovs-vsctl show` command:

```
# Ryu console
connected socket:<eventlet.green.ssl.GreenSSLSocket object at 0x7f603f8b0c08>
address:('127.0.0.1', 59430)

# ovs-vsctl
$ sudo ovs-vsctl show
873c293e-912d-4067-82ad-d1116d2ad39f
 Bridge "s1"
 Controller "ssl:127.0.0.1:6633"
 is_connected: true
 fail_mode: secure
```

Great! Now we can avoid any casual snooping of our OpenFlow messages between Ryu and the switch (yes, I realize our setup is in a single virtual machine). In the next section, let's examine a few of the marketplace switches implementation of OpenFlow to help us pick the best switch for our network.

## Physical switch selection

Physical switch selection is always an interesting question to ask when building your network. The term 'switch' in this context should be broadly interpreted as both L2 and L3 devices. In the SDN world, where we separate the control and data plane, any network function, such as routing, switching, firewall, monitoring, or firewall function, are software function while we reduce the physical packet transport into flows. The line between software-based versus hardware switch is almost non-existent when it comes to management and control plane constructs. However, we still need to be concerned with hardware capacity when it comes to building our network infrastructure.

If you draw a comparison between network engineering and systems engineering when purchasing servers, a system engineer would consider hardware specifications, such as CPU, memory, and storage, while separating the software aspects of operating system and software packages. The two sides of hardware and software still need to work together, though the industry has matured enough that for the most part the interoperability is not a big issue.

Network engineering with SDN and OpenFlow is moving toward that direction, but is not as matured. When choosing a physical switch, besides considering the size, port count, and number of programmable flows, it is always a good idea to choose switches that have gone through some testing with the SDN technology in mind. In our use case, we shoud take a closer look at switches with stated OpenFlow specification support, and perhaps consulting vendor-neutral lab testing results, such as InCNTRE from Indiana University (http://incntre.iu.edu/SDNlab). More specifically, controller projects such as Ryu and commercial controllers such as BigSwitch and Cisco ONE, all provide their own testing results.
In the years since the first introduction of OpenFlow, the market for physical switch selection has grown tremendously. I have picked three categories and representative devices to help you in your own evaluation.

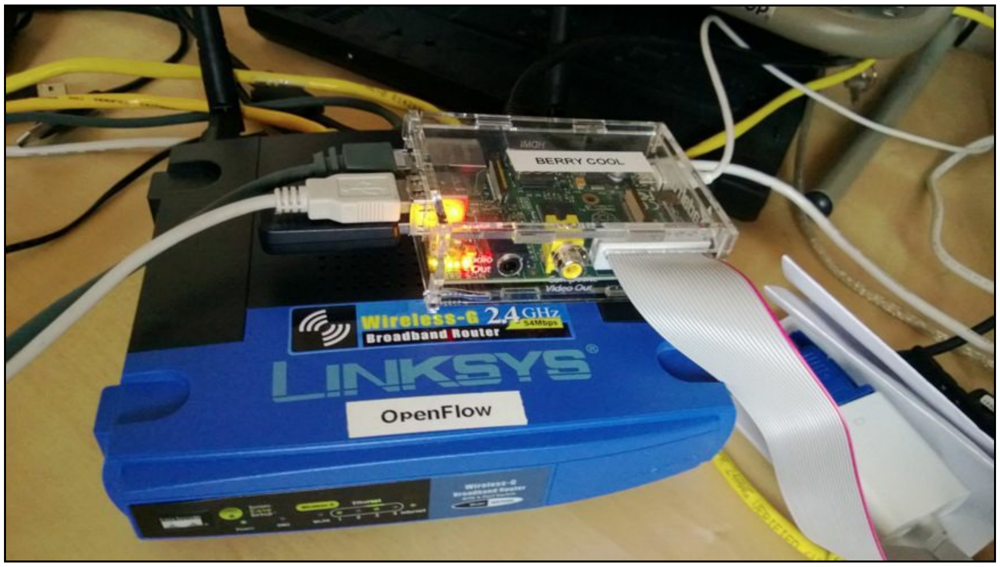
*Note that I am not promoting the switches when pointing out the model number or vendor. I simply want to include them to be more precise and allow you to have less friction when doing your own research.*

# Lab OpenFlow switches

The first category is my favorite, which we can call the hacker category. This typically consists of hardware specifications that pale in comparison to datacenter switches, but are easy to obtain and do not break your wallet. In the SimpleRouter project (https://github.com/ttsubo/simpleRouter), you can see how Toshiki Tsuboi uses a USD 35 Raspberry Pi 2 device (https://www.raspberrypi.org/) with 3 USB-Ethernet interfaces to run Open vSwitch and Ryu and demonstrate MP-BGP and VPN to external BGP devices.

Small, budget-friendly switches are also available in this category. Home routers and switches, such as TP-LINK TL-WR1043ND or Cisco-Linksys WRT54GL, have OpenWRT images that enable them to be quick OpenFlow 1.0 or 1.3 switches. For roughly USD 35 (at the time of writing), you can install an OpenFlow switch in your home and run live traffic control by your controller. Imagine a small switch that costs less than a few cups of coffee, running BGP protocol. How cool is that!

The following picture shows my first physical OpenFlow switch at home in 2011, which is a Linksys WRT54GL running OpenFlow 1.0 with a Raspberry Pi 1 acting as a controller:



The small consumer-based switches typically contain a small Linux implementation, such as `BusyBox`:

```
ssh root@192.168.1.6
BusyBox v1.15.3 (2010-05-28 12:28:17 PDT) built-in shell (ash)
Enter 'help' for a list of built-in commands.

 _______ ________ __
 | |.-----.-----.-----.| | | |.----.| |_
 | - || _ | -__| || | | || _|| _|
 |_______|| __|_____|__|__||________||__| |____|
 |__| W I R E L E S S F R E E D O M
 Backfire (10.03, r23206) --------------------------
 * 1/3 shot Kahlua In a shot glass, layer Kahlua
 * 1/3 shot Bailey's on the bottom, then Bailey's,
 * 1/3 shot Vodka then Vodka.
 ---------------------------------------------------
root@OpenWrt:~#
```

The OpenFlow configuration can be manually examined or manipulated under pre-defined category:

```
root@OpenWrt:/etc/config# cat openflow
config 'ofswitch'
 option 'dp' 'dp0'
 option 'ofports' 'eth0.0 eth0.1 eth0.2 eth0.3'
 option 'ofctl' 'tcp:192.168.1.7:6633'
 option 'mode' 'outofband'
```

The switch can associate with the controller normally:

```
ubuntu@sdnhubvm:~/ryu[10:24] (master)$ bin/ryu-manager --verbose
ryu/app/simple_switch.py
loading app ryu/app/simple_switch.py
loading app ryu.controller.ofp_handler
...
switch features ev version: 0x1 msg_type 0x6 xid 0x4498ae12
OFPSwitchFeatures(actions=3839,capabilities=199,datapath_id=150863439593,n_buffers=
{1:
OFPPhyPort(port_no=1,hw_addr='c0:c1:c0:0d:36:63',name='eth0.0',config=0,state=0,cur
 2:
OFPPhyPort(port_no=2,hw_addr='c0:c1:c0:0d:36:63',name='eth0.1',config=0,state=0,cur
 3:
OFPPhyPort(port_no=3,hw_addr='c0:c1:c0:0d:36:63',name='eth0.2',config=0,state=0,cur
 4:
OFPPhyPort(port_no=4,hw_addr='c0:c1:c0:0d:36:63',name='eth0.3',config=0,state=0,cur
move onto main mode
```

I have put this switch in my work lab a few times for quick tests. It is a very low cost, low friction way to get started when you are ready to step outside of software switches.

## Incumbent vendor switches

If you have existing vendor equipments that support OpenFlow, it would be a great way to start your migration path. Vendors such as Cisco, Juniper, and Arista, all have equipments that support OpenFlow. The depth of support and the degree of difficulty varies from vendor to vendor and in some cases, from device to device. Given some of the limitation as we will see later on, using incumbent vendor switch would be a great way to migrate to OpenFlow if you already have matched gears in your possession. In this section, I will use Arista gears running EOS 4.17 as an example. For more details, you can consult the EOS 4.17 OpenFlow configuration manual at https://www.arista.com/assets/data/docs/Manuals/EOS-4.17.0F-Manual.pdf.

In EOS 4.17, Arista supports OpenFlow in 7050 and 7050X series of switches. However, there are a number of limitations stated in the manual, so read through them to save yourself from some troubleshooting headache in future. For example, it is stated that for switch to controller interaction, TLS is not supported. If you had not read that before you tried to implement TLS, you could have spent hours troubleshooting fruitlessly. Arista also recommends using OpenFlow for the switch, even though in configuration you can bind to certain VLANs.

```
switch(config)#openflow
switch(config-openflow)#controller tcp:1.2.3.4:6633
switch(config-openflow)#bind mode vlan
switch(config-openflow)#bind vlan 1
switch(config-OpenFlow)#no shutdown
```

You can examine the state of OpenFlow:

```
switch(config)# show openflow
OpenFlow configuration: Enabled
DPID: 0x0000001c73111a92
Description: sw3-Arista
Controllers:
 configured: tcp:172.22.28.228:6633
 connected: tcp:172.22.28.228:6633
 connection count: 3
 keepalive period: 10 sec
Flow table state: Enabled
Flow table profile: full-match
```

```
Bind mode: VLAN
 VLANs: 1-2
 native VLAN: 1
IP routing state: Disabled
Shell command execution: Disabled
Total matched: 7977645 packets
```

Arista also supports an OpenFlow-like feature called DirectFlow. It supports all OpenFlow 1.0 match criteria and actions. The big difference between DirectFlow and OpenFlow is that there is no default match action, so table-miss entries are to follow the normal L2/L3 pipeline. DirectFlow is mutually exclusive with OpenFlow. Configuration is straight forward:

```
switch(config)#directflow
switch(config-directflow)#
switch(config-directflow)#no shutdown
switch(config-directflow)#flow Test1
switch(config-directflow-Test1)#match ethertype ip
switch(config-directflow-Test1)#match source ip 10.10.10.10
switch(config-directflow-Test1)#action egress mirror ethernet 7
switch(config-directflow-Test1)#action set destination mac 0000.aaaa.bbbb
```

The Arista example demonstrates the limitation we have discussed. The supportability of OpenFlow from incumbent vendor varies from device to software. The vendors typically add their own flavor as they see fit. We have seen an example of Arista, but they are not alone in that each vendor have their own limitation from years of building non-OpenFlow devices. However, if you already have some vendor equipment that matches the criteria, which is a big if, it is a great way to get started.

## Whitebox Switches

Whitebox switches represent a new category of switches that were brought to mass market by the SDN movement. They represent a class of switches that typically made by vendors who made network hardware as outsourcers for the typical commercial vendors such as Cisco. Since the SDN movement allows network engineers to only care about 'raw' forwarding plane, the whitebox switch vendors can now sell directly to the network engineers. They are just blank switches upon which you can load separate software for them to operate on. The hardware vendors include Accton, Celestica, and Quanta, amongst others.
The software you can load on the switch varies from just an agent interacting with controller, such as Big Switch Networks Switch Light OS, to standalone full-feature operating system, such as Pica8 and Cumulus Linux. This is an area that is quickly evolving and offers no shortage of innovation. The following outputs show a relatively old Quanta LB9 switch running Pica8 PicOS 2.8.1. The software offers different modes, such as OpenFlow native or hybrid mode, where the port follows the normal L2/L3 pipeline for flow-miss.

```
XorPlus login: admin
Password:
admin@XorPlus$

admin@XorPlus$cli
Synchronizing configuration...OK.Pica8 PicOS Version 2.8.1

Welcome to PicOS on XorPlus
admin@XorPlus>

admin@XorPlus> show version
Copyright (C) 2009-2017 Pica8, Inc.
===================================
Base ethernet MAC Address : <skip>
Hardware Model : P3295
Linux System Version/Revision : 2.8.1/f2806cd
Linux System Released Date : 01/04/2017
L2/L3 Version/Revision : 2.8.1/f2806cd
L2/L3 Released Date : 01/04/2017

admin@XorPlus>
```

Can you tell it was a Quanta LB9 switch? No, which is the point. Pica8 software or other software will run the same on all whitebox switches. This category of switches represents the true potential of SDN and the closest that we have seen of a decoupled future of network hardware and software.

## Summary

In this chapter, we discussed the migration steps of implementing Software Defined Networking into your network. Using OpenFlow and the Ryu controller, that we have discussed in previous chapters, we put the knowledge we gained into practical use. We began with possible reasons for why you would NOT want to implement SDN and OpenFlow, and debunked them. Then we laid out the steps and strategies to prepare your network for SDN migration. We discussed greenfield deployment and controller redundancy. In the next section, we also gave two extensive BGP examples that would assist in the hybrid mode.

We then discussed monitoring of your SDN OpenFlow network, as well as securing the communication from switch to controller communication by using TLS encryption. The last section listed out three categories of physical OpenFlow switches that you can purchase today, from lab to commercial vendor equipment. Each of the categories has its own pros and cons. We also briefly saw examples of each category.

It is no doubt an exciting time to be a network engineer, or however the term will evolve in the future. You might be called a network programmer, network developer, or simply 'The Network Guy' in the ever changing field. What does not change is change itself, as well as the desire to have better, faster, more flexible network that will assist the business in achieving its goals. From one network engineer to another, I thank you for allowing me to take part in this journey with you. I hope I have provided some assistance to you with this book and the topics we have covered, and I humbly ask for your feedback, if any, to help me improve the future contents. Happy engineering!