AMBASSADOR

DAY ONE: JUNOS® PyEZ COOKBOOK

Co-written by Juniper Customers, Ambassadors, Partners, and Employees

Automate your network tasks with Junos PyEZ scripts that save time, energy, and effort. You don't have to be a coder to take advantage of Junos and PyEZ.

By Peter Klimai, Matt Mellin, Michel Tepper, Jac Backus, Ivan Del Rio Fernandez, Paul McGinn, Scott Ware, Michelle Zhang, Diogo Montagner, Stephen Steiner, Ben Dale, Sean Sawtell, and Jessica Garrison

DAY ONE: JUNOS® PyEZ COOKBOOK

Day One: Junos PyEZ Cookbook is a complete network automation cookbook with a set-up guide, a start-up sandbox, and a complete showcase of automation scripts that are readily available on GitHub. The cookbook is a joint effort by Juniper's engineers and all the many Junos users and customers who want to show you how 'EZ' network automation can be. You don't have to be a coder when you can leverage Junos as your network OS.

> "The decision to read a technical book involves a cost-benefit analysis. *'Is the information I'm going to learn worth my time and energy?'* If you're currently operating a network of Juniper devices using the Junos OS command line interface, then the unequivocal answer is *'Yes!'* Written by an experienced team of Juniper customers, ambassadors, partners, and employees, *Day One: Junos PyEZ Cookbook* demonstrates its authors' real-world experience in operating and automating networks. More importantly, the book breaks what could be daunting tasks into small and relevant recipes. You will be creating useful network automation tools with the Junos PyEZ library on day one. Happy Automating!"
>
> *Stacy Smith, Sr. Software Developer for Junos Automation, Juniper Networks,*
> *Co-Author of* Automating Junos Administration

IT'S DAY ONE AND YOU HAVE A JOB TO DO, SO LEARN HOW TO:

- Understand Basic Python Concepts; Get the Downloads and Resources to Set Up Your Lab
- Automate NETCONF Service Activation
- Learn Terminal Server Mapping
- Check File System Directory Usage on Multiple Devices in Parallel
- Configure Devices using Junos PyEZ and Jinja2 Templates
- Take a Snapshot on ACXs Access Routers
- Extract Operational Data from Devices Running on a Bridge Domain Environment
- Reserve Bandwidth for MPLS Access Rings
- Add a Graphical Interface to the PyEZ Script
- Monitor IPSEC Tunnels
- Work with Junos Enhanced Auto-provision Process (JEAP)
- Code PyEZ for On-box Scripts
- Automate Network Testing with Junos PyEZ
- Create a Menu Script for Address Book Editing
- Provision L3VPN Services on PE Routers
- Identify and Disable Unused Interfaces with Ansible
- Track Down IP Conflicts with PyEZ
- Code a Configuration Audit Using XML Schema (XSD)

ISBN 978-1-941441-59-6
53500
9 781941 441596

JUNIPER NETWORKS

Day One: Junos® PyEZ Cookbook

by Peter Klimai, Matt Mellin, Michel Tepper, Jac Backus, Ivan Del Rio Fernandez, Paul McGinn, Scott Ware, Michelle Zhang, Diogo Montagner, Stephen Steiner, Ben Dale, Sean Sawtell, and Jessica Garrison

Published by Juniper Networks Books
ISBN: 978-1-941441-59-6 (print)
Printed in the USA by Vervante Corporation.
ISBN: 978-1-941441-60-2 (ebook)
Version History: v1, October, 2017
2 3 4 5 6 7 8 9 10

Table of Contents

Contributors

Peter Klimai currently works as an instructor and content developer at Poplar Systems, a Juniper Networks Authorized Education Partner. He is a Juniper Ambassador and certified JNCIE-SEC #98, JNCIE-ENT #393, JNCIE-SP #2253, JNCIP-DC, and JNCIS-SDNA. Peter is enthusiastic about network automation using various tools, as well as software-defined networking and network function virtualization. Peter's past includes several years of experience supporting Juniper equipment for multiple customers of varying sizes, as well as writing a PhD thesis in cosmology.

Matt Mellin currently works at Juniper Networks as a data analyst and domain expert within the CTO office's Data Analytics Group. His group applies machine learning to various network-related issues. His past includes several years as technical lead for Juniper's Proof-of-Concept (POC) lab in Sunnyvale, and (stretching even further back) as a sales engineer at NetScreen. Matt is enjoying the transition from network engineer to software developer and is enthusiastic about things like big data, cloud native network design, application security, network automation, and dev ops.

Michel Tepper has been a Juniper consultant and instructor for the last ten years of his 30+ years career. He teaches at Westcon in the Netherlands and holds a number Juniper certifications. He started his career as a programmer, and now that networking is moving towards SDN is enjoying the benefits from those years. Michel has also been a Juniper Ambassador for several years, calling the Ambassadors "the nicest peer group in the IT industry."

Jac Backus is a network engineer at BugWorks, a Juniper partner. He has worked for over 30 years in the IT industry and has several years of experience installing and supporting Juniper equipment. He is always eager to learn more about networking, virtualization, and automation, and is also interested in computer and network security.

Ivan Del Rio Fernandez currently works as an IP Engineer at DQE Communications administering ISP-related functions such as routing and switching using Juniper equipment. He is JNCIA and RHCE certified. Before his venture in networking he was a Linux System Administrator in Europe for four years. Ivan is passionate about networking, Linux, scripting languages, and optimizing efficiency through automation. In his spare time, he enjoys flying drones and working on open-source robotics projects.

Paul McGinn is an IP Engineer with DQE Communications in Pittsburgh Pennsylvania. When Paul isn't chasing kids around a soccer field, he is working on customer service design, network monitoring and automation, while leading the DQE implementation team. He has seen the benefits that SDN brings to network environments and is excited for what the future holds.

Scott Ware currently works as a Senior Security Engineer for a large retailer. He is a Juniper Ambassador and certified GSEC, JNCIS-SEC, and JNCDA. Scott has been a fan of automation for most of his career, contributing multiple open-source packages and tools to the community, written in various languages.

Michelle Zhang currently works at Juniper Networks as a Systems Engineer Specialist with focus on automation. She is a bright new college grad who is passionate about network automation with scripting as well as non-scripting tools, and a big fan of Cloud and Big Data.

Diogo Montagner is a consulting engineer in the Center of Excellence within Juniper Networks APAC, focused on automation and software solutions. He holds JNCIE-SP #1050 and PMP certifications, as well as a 2nd Dan black belt in Taekwondo. Diogo is a network enthusiast who helps customers to design, build, automate, and operate their networks.

Steve Steiner currently works as an automation solutions consultant and DevOps engineer at Juniper Networks and is a certified JNCIE-SP #323. Steve is a veteran Tech Controller of the US Air Force and has been in networking since before the Internet was the Internet. When he's not working, he enjoys music, movies, caffeine, and driving in his Jeep with the top off.

Ben Dale is a Network Engineering Manager at Comlinx, a Juniper Elite Partner based in Brisbane, Australia. He is a Juniper Ambassador, and certified JNCIE-SEC #63, JNCIP-SP, JNCIP-ENT, JNCP-DC. Ben is passionate about using network automation to solve day-to-day operational issues. When Ben isn't neck deep in networking, he enjoys playing the ukulele (badly) and skateboarding with his daughters.

Sean Sawtell has been with Juniper Networks since 2002, and has been a Network Engineer with Juniper's internal network team since 2004. Sean's focus today is on network automation. In 2014 Sean earned a Master of Science degree in Computer Science, and subsequently was an adjunct professor for two years teaching the CS curriculum. Before joining Juniper, Sean taught Microsoft and Novell courses and held MCSE, MCI, CNE, and CNI certifications.

Jessica Garrison currently works as a Technical Marketing Engineer at Juniper Networks. Since completing a master's degree in electrical engineering, she has accumulated over a decade of networking experience within tech support, consulting, sales, and technical marketing. Jessica enjoys evangelizing the culture and methodology of network automation and occasionally gets her hands dirty with some good-enough coding. She used to enjoy hiking, biking, and cooking before expanding her family to include a daughter and two large dogs.

This Cookbook's PyEZ Script Repository

This cookbook's PyEZ scripts exist on GitHub as open source files. Look for the original files and updates here: https://github.com/Juniper/junosautomation/tree/master/pyez/PyEZ_Cookbook_2017.

List of Resources for Community Help and Support

Resource Type	Description	Location
Website	PyEZ Cookbook on GitHub	https://github.com/Juniper/junosautomation/tree/master/pyez/PyEZ_Cookbook_2017
Forums	PyEZ Google Group	https://groups.google.com/forum/ - !forum/junos-python-ez
Forums	Juniper J-NET TechWiki	http://forums.juniper.net/t5/Automation/tkb-p/Automation_Scripting
Forums	StackOverflow PyEZ tag	https://stackoverflow.com/questions/tagged/pyez
Book	Automating Junos Administration (specifically, Chapter 4)	http://shop.oreilly.com/product/0636920041498.do
Website	The TechLibrary's PyEZ Documentation	https://www.juniper.net/documentation/en_US/junos-pyez/information-products/pathway-pages/junos-pyez-developer-guide.html
Website	PyEZ at ReadTheDocs	http://junos-pyez.readthedocs.io

10 Things About Coding for Non-Coders

If you are one of those people who think "I will never be a coder," start here:

1. You don't have to be a software developer to code.

2. You also don't have to be "super" smart. It doesn't take a genius.

3. Mistakes and failure are okay. Trial and error is a great way to learn. This is why labs and demos exist.

4. The more people who review your code, the stronger it gets. Swap ideas. Open source your code.

5. Computers and servers will do exactly what you code them to do.

6. Embrace change.

7. Go slowly. Step-by-step. Use version control.

8. Join a community. GitHub is great. They have guides online. The Juniper Forum is also wonderful (forums.juniper.net). Communities are great ways to share code, borrow ideas, and submit questions.

9. Copy and paste are your friends. Most of us do not write code from scratch. You borrow and manipulate, and one hundred iterations later, you have something that works.

10. Learn from this book.

Preface

The typical IT priorities of a network operator are to increase productivity of resources, ensure compliances, improve network security, reduce IT costs, and simplify IT management – all of which are aimed at reducing OpEx and improving bottom line profits. The operators that understand these business priorities look at how they can leverage technology and become more competitive in the markets they do play in, and if you drill down into any of these priorities or goals, you'll see that network automation is a key ingredient in their operations.

Network automation can be found in broad categories of tools like Operator Support Systems (OSS), Business Support Systems (BSS), Orchestrators, Controllers, Element Management Systems (EMS), and more, which operate at various hierarchies in the network. But if you get closer to the individual network devices, there is a category of tools that enable device automation, and here, Juniper's PyEZ (Python Easy) excels!

Why easy? Because PyEZ is a Python library that enables administration and automation of Junos® devices by invoking remote procedure calls on Junos devices. It provides an abstraction built on top of the NETCONF protocol leveraging the NETCONF client library underneath, and talks to Junos devices via XML remote procedure calls (RPCs). The RPC responses can be returned as XML-specific Python data structures and that makes it easy to consume within the Python script. PyEZ provides several predefined tables and views for common RPCs and also allows users to define their own scripts to extract information from any Junos RPC. On the configuration side, PyEZ allows formats such as XML, Junos CLI set commands, and also text. One can also leverage YAML and Jinja to generate device, feature, or customer specific configurations based on predefined templates and values.

PyEZ is an open-source project (see https://github.com/Juniper/py-junos-eznc) developed and supported by Juniper that welcomes feature contributions, bug fixes, and issue reports from the user community. It can be used from the interactive Python shell to quickly perform tasks, or can be incorporated into full blown Python scripts for more complex use cases. You just need to install the PyEZ package on a server that has connectivity to the Junos devices intended to be automated, and you are ready to start making your network more agile, compliant, efficient, and scalable.

The best part about PyEZ is that it has a very gentle learning curve. You don't need to be a programmer or a coder to be able to get automation up and running. Besides, it will get you a step closer to learning Python, one of the most popular languages within the programmer community, thus expanding your skillset.

Moving from manual to automation mode needs an enabler and this cookbook is aimed at just that! The authors have compiled some of the most useful recipes (use cases) we've seen in the field, and have presented them in the simplest possible manner. Beginning with Hello World to get you started, and continuing with production-ready recipes for configuration management, templating, service provisioning, and more, this cookbook gets you started, gets you going, and then gets you using PyEZ for device automation.

PyEZ is a popular tool and it is extensively used globally by Junos OS users in production networks. Juniper is committed to investing in PyEZ to make it an even more loved tool by its users, and all of us hope that you as a network operator are able to get much more out of your network by automating with PyEZ! Enjoy the book.

Raunak Tibrewal, Juniper Networks Product Manager for Junos OS
Sunnyvale, California, Novembers, 2017

Recipe 1 - Installing Python and PyEZ

by Steve Steiner

Before you can start automating Junos devices, you need to have the necessary tools installed and functioning properly. This recipe helps you get the Python language and the PyEZ libraries installed on your development machine.

Problem

You want to start your network automation tasks but are confused about languages, dependencies, libraries, and modules. Installing all the required "stuff" seems complicated, and you want to get started the right way.

Solution

There are several components required in order to run the various PyEZ recipes in this cookbook, and the steps to take to install them vary somewhat, depending on your development machine.

Microsoft Windows

(Tested against Windows 10)

Installing Python

Download Python 3.x for Windows from https://www.python.org/downloads/ (the Windows x86-64 executable installer) and double click the installer file.

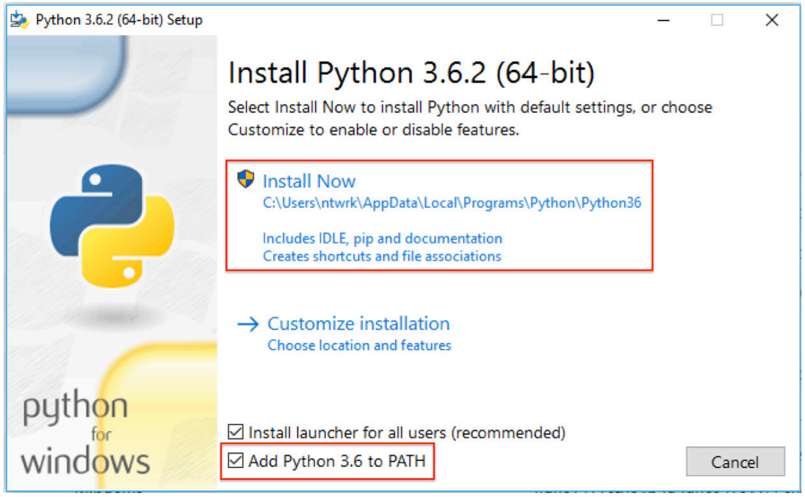

Python 3.6.2 (64-bit) Setup
Install Python 3.6.2 (64-bit)
Select Install Now to install Python with default settings, or choose Customize to enable or disable features.
Install Now
C:\Users\ntwrk\AppData\Local\Programs\Python\Python36
Includes IDLE, pip and documentation
Creates shortcuts and file associations
Customize installation
Choose location and features
☑ Install launcher for all users (recommended)
☑ Add Python 3.6 to PATH
Cancel
python for windows

Figure 1.1 Installing Python

Check the box next to the "Add Python 3.6 to PATH" option and then click on "Install Now", as shown in Figure 1.1.

Installing PyEZ

To install PyEZ, open up a command window and type `pip install junos-eznc`. Pip will automatically download and install all of the required packages.

Once that's finished, you should test the installation.

Testing

Open up a command window and start Python by typing `python` then type `from jnpr.junos import Device` at the prompt. You'll see something similar to the following:

```
C:\Users\ntwrk>python
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from jnpr.junos import Device
>>>
```

If everything is installed properly, it should look like the above example. If PyEZ is not installed properly, you will see something similar to this output:

```
>>> from jnpr.junos import Device
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'jnpr'
>>>
```

To quit the Python interpreter, type `quit()`.

If you experience any issues with the installation of PyEZ you can search for help in past issues, or submit a new issue at the PyEZ GitHib project located at https://github.com/Juniper/py-junos-eznc.

Apple MacOS

(Tested against MacOS Sierra)

Installing Python

While MacOS (and OS X) ship with a working version of Python, it's best to install another instance to avoid any potential issues arising from the vendor-installed software.

Download Python 3.x for MAC OS X from https://www.python.org/downloads/

Double click on the .pkg and follow the prompts. This will install Python in `/usr/local/bin`. The executable is called `python3`.

Installing PyEZ

Installing PyEZ is as simple as typing:

```
$ pip3 install junos-eznc
```

Testing

You can test it by opening the Python interpreter and attempting to load the PyEZ module:

```
$ python3
Python 3.6.2 (v3.6.2:5fd33b5926, Jul 16 2017, 20:11:06)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type “help”, “copyright”, “credits” or “license” for more information.
>>> from jnpr.junos import Device
>>>
```

If there are no errors, then PyEZ is ready to use. If the module is not installed properly, you will see an error similar to the following:

```
>>> from jnpr.junos import Device
Traceback (most recent call last):
  File “<stdin>”, line 1, in <module>
ModuleNotFoundError: No module named ‘jnpr’
>>>
```

To quit the Python interpreter, type `quit()`.

If you experience any issues with the installation of PyEZ you can search for help in past issues, or submit a new issue at the PyEZ GitHib project located at https://github.com/Juniper/py-junos-eznc.

Ubuntu

(Tested against Ubuntu 16.04)

Installing Python

Ubuntu ships with Python 3.5, which is sufficient for our needs, but we do need to install some dependencies prior to installing PyEZ.

So, let's update the `apt` cache and install the dependencies:

```
$ sudo apt-get update
$ sudo apt-get install -y libxslt1-dev libssl-dev libffi-dev python-dev build-essential --no-install-
recommends
```

Next, install pip:

```
$ wget https://bootstrap.pypa.io/get-pip.py -O - | sudo -H python3
```

And install PyEZ:

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

Testing

You can test it by opening the Python interpreter and attempting to load the PyEZ module:

```
$ 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
>>> from jnpr.junos import Device
>>>
```

If there are no errors, then PyEZ is ready to use. If the module is not installed properly, you would see an error similar to the following:

```
$ 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.
>>> from jnpr.junos import Device
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'jnpr'
>>>
```

To quit the Python interpreter, type quit().

If you experience any issues with the installation of PyEZ you can search for help in past issues, or submit a new issue at the PyEZ GitHib project located at https://github.com/Juniper/py-junos-eznc.

Symlinks (optional)

By default, invoking `python` on Ubuntu will call Python 2.7:

```
$ python --version
Python 2.7.12
```

Since you'll be using Python 3, you can simplify the process by changing `python` to call `python3` instead of `python2`:

```
$ which python
/usr/bin/python
$ ls -al /usr/bin/ | grep python
<snipped>
lrwxrwxrwx  1 root root         9 Dec  9  2015 python -> python2.7
<snipped>
```

Remove the current symlink and add the new one:

```
$ sudo rm /usr/bin/python
$ sudo ln -s /usr/bin/python3 /usr/bin/python
$ python --version
Python 3.5.2
```

CentOS

(Tested against CentOS 7.3)

Installing Python

CentOS 7 ships with Python 2, so you'll need to install Python 3. Let's update `yum` and add some dependencies that you'll need later:

```
$ sudo yum makecache fast
$ sudo yum install gcc gcc-c++ kernel-devel make automake
```

You'll need to add the Inline with Upstream Stable (IUS) repository and install Python 3.6:

```
$ sudo yum -y install https://centos7.iuscommunity.org/ius-release.rpm
$ sudo yum -y install python36u
```

CentOS doesn't create symlinks for Python 3, so you can do that now to make it easier:

```
$ ls -al /usr/bin/ | grep python
-rwxr-xr-x.  1 root root     11232 Dec  2  2016 abrt-action-analyze-python
lrwxrwxrwx.  1 root root         7 Jul 27 11:44 python -> python2
lrwxrwxrwx.  1 root root         9 Jul 27 11:44 python2 -> python2.7
-rwxr-xr-x.  1 root root      7136 Nov  5  2016 python2.7
-rwxr-xr-x.  2 root root     11312 Apr  7 10:35 python3.6
-rwxr-xr-x.  2 root root     11312 Apr  7 10:35 python3.6m
```

First, delete the existing symlink for Python:

```
$ sudo rm /usr/bin/python
```

Next, symlink Python3 to Python 3.6:

```
$ sudo ln -s /usr/bin/python3.6 /usr/bin/python3
```

Finally, link Python to Python3:

```
$ sudo ln -s /usr/bin/python3 /usr/bin/python
$ ls -al /usr/bin/ | grep python
-rwxr-xr-x.   1 root root     11232 Dec  2  2016 abrt-action-analyze-python
lrwxrwxrwx.   1 root root        16 Jul 31 16:01 python -> /usr/bin/python3
lrwxrwxrwx.   1 root root         9 Jul 27 11:44 python2 -> python2.7
-rwxr-xr-x.   1 root root      7136 Nov  5  2016 python2.7
lrwxrwxrwx.   1 root root        18 Jul 31 16:00 python3 -> /usr/bin/python3.6
-rwxr-xr-x.   2 root root     11312 Apr  7 10:35 python3.6
-rwxr-xr-x.   2 root root     11312 Apr  7 10:35 python3.6m
```

Now, let's install pip:

```
$ wget https://bootstrap.pypa.io/get-pip.py -O - | sudo -H python
```

And to install PyEZ:

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

Testing

Test by starting the Python interactive shell and loading the PyEZ module:

```
$ python
Python 3.6.1 (default, Apr  7 2017, 09:32:32)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-11)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from jnpr.junos import Device
```

If there are no errors, then PyEZ is ready to use. If the module is not installed properly, you would see an error similar to the following:

```
$ python
Python 3.6.1 (default, Apr  7 2017, 09:32:32)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-11)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from jnpr.junos import Device
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'jnpr'
```

To quit Python, type `quit()` at the Python prompt.

If you experience any issues with the installation of PyEZ you can search for help in past issues, or submit a new issue at the PyEZ GitHib project located at https://github.com/Juniper/py-junos-eznc.

Discussion

There are some examples for getting started located at https://github.com/Juniper/py-junos-eznc. This is also a great site if you run into issues and need help either installing PyEZ or running a script.

Additionally, the Automation Forum on the Juniper Tech Wiki is a great place to seek help and see more script examples. It can be found at http://forums.juniper.net/t5/Automation/tkb-p/Automation_Scripting.

Recipe 2 - Enabling NETCONF

by Steve Steiner

You need to do one more thing before you can execute PyEZ scripts against a Juniper device – enable NETCONF.

Problem

Juniper devices must have NETCONF enabled prior to accepting incoming connections from PyEZ.

Solution

Enable NETCONF in the Juniper device configuration using the following guidelines:

Check NETCONF configuration

Check if NETCONF is already enabled. Do this by entering the following commands from the Operational Mode command prompt:

```
automation@MX11> show configuration system services netconf
```

If NETCONF is configured, you will see output similar to this:

```
automation@MX11> show configuration system services netconf
ssh;

automation@MX11>
```

If NETCONF is not configured, you will see output similar to this:

```
automation@MX11> show configuration system services netconf

automation@MX11>
```

Non-SRX devices

Enabling NETCONF on non-SRX devices is straightforward. Enter configuration mode by typing configure:

```
automation@MX11> configure
Entering configuration mode

[edit]
automation@MX11#
```

Then enable NETCONF over SSH:

```
[edit]
automation@MX11# set system services netconf ssh

[edit]
automation@MX11#
```

Finally, commit your changes and exit configuration mode:

```
[edit]
automation@MX11# commit and-quit
commit complete
Exiting configuration mode

automation@MX11>
```

On SRX Devices

Configuring NETCONF on an SRX device that is in flow-mode requires some extra steps. In addition to enabling NETCONF over SSH at the system services stanza, you'll also need to allow NETCONF (and SSH) into the `management functional-zone`.

To check if NETCONF is already allowed into the management zone, use the `show security zones functional-zone management` command:

```
[edit]
automation@vsrx# show security zones functional-zone management
interfaces {
   ge-0/0/1.0 {
      host-inbound-traffic {
         system-services {
            ssh;
         }
      }
   }

}
```

You can see that SSH is allowed, however, NETCONF is not. To enable NETCONF, add it to the `host-inbound-traffic system-services`, either under the zone itself, or under a specific interface. In this case, it will be added to interface ge-0/0/1.0:

```
[edit security zones functional-zone management interfaces ge-0/0/1.0]
automation@vsrx# set host-inbound-traffic system-services netconf
```

Verify the changes:

```
[edit security zones functional-zone management interfaces ge-0/0/1.0]
automation@vsrx# top
[edit]
automation@vsrx# commit check
configuration check succeeds
```

Then commit the changes:

```
[edit]
automation@vsrx# commit
commit complete
```

Recipe 3 - Hello World!

by Steve Steiner

"Hello World!" is the traditional first attempt when a user is learning a new programming language. While PyEZ isn't a language, per se, you do need a starting point and this recipe demonstrates how to use PyEZ to connect to a Juniper device and gather some basic information.

Problem

You want to ensure that your PyEZ installation is complete and can successfully connect to a Juniper device.

Solution

The basis for any automation project using PyEZ is to first be able to connect to a Juniper device. The following Python code will allow you to connect to a Juniper device and collect the basic facts about that device:

```
from pprint import pprint
from jnpr.junos import Device

dev = Device(host='192.168.32.2', user='automation', password='automation1' )

dev.open()

pprint(dev.facts)

dev.close()
```

NOTE By default, NETCONF over SSH listens on port 830. It's also possible to connect pass NETCONF to a device over port 22, but the argument `port=22` would need to be added to the `Device` call. For example:

```
dev = Device(host='192.168.32.2', user='automation', password='automation1',port='22' )
```

For this example, a vSRX was used, which yielded the following output:

```
$ python hello-world.py
{'2RE': False,
 'HOME': '/var/home/automation',
 'RE0': {'last_reboot_reason': '0x4000:VJUNOS reboot',
        'mastership_state': 'master',
        'model': 'VSRX-S',
        'status': 'OK',
        'up_time': '1 day, 11 hours, 5 minutes, 34 seconds'},
 'RE1': None,
 'RE_hw_mi': False,
 'current_re': ['master',
               'node',
               'fwdd',
               'member',
               'pfem',
               'fpc0',
               're0',
               'fpc0.pic0'],
 'domain': None,
 'fqdn': 'vsrx',
 'hostname': 'vsrx',
 'hostname_info': {'re0': 'vsrx'},
 'ifd_style': 'CLASSIC',
 'junos_info': {'re0': {'object': junos.version_info(major=(15, 1), type=X, minor=(49, 'D', 90),
build=7),
                      'text': '15.1X49-D90.7'}},
 'master': 'RE0',
 'model': 'VSRX',
 'model_info': {'re0': 'VSRX'},
 'personality': None,
 're_info': {'default': {'0': {'last_reboot_reason': '0x4000:VJUNOS reboot',
                             'mastership_state': 'master',
                             'model': 'VSRX-S',
                             'status': 'OK'},
                       'default': {'last_reboot_reason': '0x4000:VJUNOS '
                                                        'reboot',
                                  'mastership_state': 'master',
                                  'model': 'VSRX-S',
                                  'status': 'OK'}}},
 're_master': {'default': '0'},
 'serialnumber': '7B188C38E0D0',
 'srx_cluster': False,
 'srx_cluster_id': None,
 'srx_cluster_redundancy_group': None,
 'switch_style': 'VLAN_L2NG',
 'vc_capable': False,
 'vc_fabric': None,
 'vc_master': None,
 'vc_mode': None,
 'version': '15.1X49-D90.7',
 'version_RE0': '15.1X49-D90.7',
 'version_RE1': None,
 'version_info': junos.version_info(major=(15, 1), type=X, minor=(49, 'D', 90), build=7),
 'virtual': None}
```

More Detailed Explanation

The first two lines import the `pprint` library and the `Device` class of PyEZ, respectively:

```
from pprint import pprint
from jnpr.junos import Device
```

In the next line, you create a variable, `dev`, and assign it the `Device()` class:

```
dev = Device(host='192.168.32.2', user='automation', password='automation1' )
```

Next, you open the connection to the Juniper device using the `open()` function. This will open a connection to the device and, by default, gather some facts about that device:

```
dev.open()
```

While the `dev.open()` line opened the connection and gathered some basic facts, it doesn't display them unless you explicitly tell it to. This is what you do with the next line, using the `pprint()` function:

```
pprint(dev.facts)
```

Finally, you need to close the connection to the device, by using the `close()` function:

```
dev.close()
```

If at this point you're confused about what a function or library is, or how to assign and reference variables, you might pause this book and read the tutorials at https://www.learnpython.org/. If you're already familiar with Python, the PyEZ library is fully documented at http://junos-pyez.readthedocs.io.

MORE? If you experience any issues you can seek help either at the PyEZ GitHub project, https://github.com/Juniper/py-junos-eznc, or at the Juniper Automation Tech Wiki: http://forums.juniper.net/t5/Automation/tkb-p/Automation_Scripting.

Recipe 4 - PyEZ Connection Options

by Steve Steiner

In order to execute scripts against a device, a NETCONF session needs to be established. This recipe provides four methods for doing so.

Problem

I need to connect to my device(s) to run PyEZ scripts against them.

Solution

There are a few different ways to connect to your device(s) with PyEZ. Deciding which course of action is right really depends on your circumstances. This recipe demonstrates four connection methods and points out the benefits and limitations of each.

Direct Reference

The easiest connection method is to directly reference the connection parameters in the connection function, as discussed in “Recipe 03 – Hello World!”:

```
dev = Device(host=’192.168.32.2’, user=’automation’, password=’automation1’ )
```

Benefits	Simple Description
Drawbacks	Credentials are exposed in the code. Can only be used for one device.
Best use	Debugging. Python interpreter as a Junos power shell.

Interactive Input

Another way to pass the credentials to a device is to prompt for user input using `input()` for the hostname/IP and username, and `getpass` for the password:

```
import getpass
from jnpr.junos import Device

host = None
uname = None
pw = None

if host == None:
  host = input("Hostname or IP: ")

if uname == None:
   uname = input("Username: ")

if pw == None:
  pw = getpass.getpass()

dev = Device(host=host,user=uname,password=pw)
```

Benefits	Credentials are not in the source code, making this both secure and portable.
Drawbacks	Can only be used for one device.
Best use	Quick scripts that need only to connect to a single device.

SSH Public Keys

By far the easiest and most secure connection method is using SSH keys. First, ensure that the user account on the device has a corresponding SSH key configured:

```
automation@mx80> show configuration system login user automation
uid 2004;
class super-user;
authentication {
   ssh-rsa "ssh-rsa
…key data…"; ## SECRET-DATA
}
```

When using SSH key-based authentication, the only argument needed for `Device()` is `host`, so the code is greatly simplified:

```
from jnpr.junos import Device
host = None

if host == None:
  host = input("Hostname or IP: ")

dev = Device(host=host)
```

This code assumes that the user invoking the script has an account on the device and the private key in their default key agent. Alternatively, it’s possible to reference the private key file from within the script. If the private key is passphrase protected, the password argument in `Device()` is used to decrypt the private key:

```
import getpass
from jnpr.junos import Device
from pprint import pprint

host = None
uname = None
pw = None
key_file = ‘./id_rsa_mx80’

if host == None:
  host = input(“Hostname or IP: “)

if uname == None:
   uname = input(“Username: “)

if pw == None:
  pw = getpass.getpass(“SSH private key passphrase: “)

dev = Device(host=host,user=uname,password=pw,ssh_private_key_file=key_file)
```

In this example, there is a passphrase-protected private key located in the same directory as the script, called `id_rsa_mx80`. The public key was configured under the user ‘automation’ on the device. When the script is run, it will prompt for the hostname/IP, user name, and the passphrase to decrypt the private key.

Benefits	Credentials are not in the source code, making this both secure and portable.
Drawbacks	Can only be used for one device.
Best use	Quick scripts that need to only connect to a single device.

Extensibility

All the preceding examples were able connect to a single device. This limits the usefulness when it’s desirable to run the same script against multiple devices. The solution is to parse an input file and use those values to feed the variables for `Device()`:

```
import getpass
from jnpr.junos import Device
import sys
import argparse
import os

filename = None
host = None
uname = None
```

```
pw = None
key_file = './id_rsa_mx80'

if filename == None:
   filename = raw_input("Enter filename: ")

if uname == None:
   uname = raw_input("Username: ")

if pw == None:
   pw = getpass.getpass("SSH private key passphrase: ")

with open(filename) as f:
   for line in f:
      host = line.rstrip(os.linesep)

dev = Device(host=host,user=uname,password=pw,ssh_private_key_file=key_file)
```

Now the script prompts the user for a CSV list of hostnames or IP addresses. This list is iterated over, and each line is stored as, the host variable. This type of connection works well when multiple devices can all be connected using common credentials, which is customary in most networks.

Benefits	Credentials are not in the source code, making this both secure and portable. Able to connect to many devices.
Drawbacks	Assumes all devices use the same credentials.
Best use	Running the same script against many devices.

Recipe 5- Getting Started

by Jessica Garrison

- Python Version Used: 2.7
- PyEZ Version Used: 2.1.5
- Junos OS Used: 11.4
- Juniper Platforms General Applicability: (MX, EX, etc.)
- Example by PyEZ Developer Nitin Kumar: https://github.com/vnitinv/pyez-examples/blob/master/7_rollback_config.py

So how does one get started? If you have followed Recipes 1-4 in this cookbook, you have set up the server/VM/container with all the necessary software requirements to be the PyEZ server. Pair this with a Junos device running at least Junos OS 11.4 (Junos hardware or NFV) and you are good to go. Here is a super simple script by PyEZ Juniper expert, Nitin Kumar, that shows off the elegance of PyEZ.

Problem

An operator has been making configuration changes without committing the configurations. When a second operator makes a different configuration change and commits, that second operator will be committing both configurations. Problems ensue.

Solution

Run a job that does a rollback (see Figure 5.1) if there is a candidate configuration, preventing this issue from occurring.

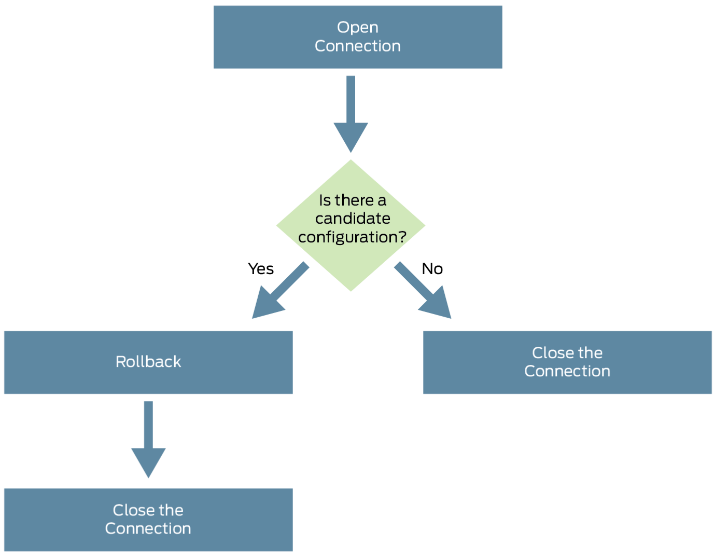


Figure 5-1 Recipe 5'sSolution Flowchart

Step-By-Step

Items needed: Hostname or IP, username, and password.

- Connect to the Junos OS device.
- Run a `show | compare` CLI command.
- If there is a candidate configuration, execute a `rollback`.
- Close the connection to the Junos OS device.

Now the code (nine lines):

```
from jnpr.junos import Device
from jnpr.junos.utils.config import Config

dev = Device(host='xxxx', user='demo', password='demo123', gather_facts=False)
dev.open()

cu = Config(dev)
diff = cu.diff()
if diff:
   cu.rollback()
dev.close()
```

Next the code broken down piece by piece. Let's begin with pre-flow, and import the Junos module in order to be able to use PyEZ on Junos OS devices:

```
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
```

Items needed: Hostname or IP, username, and password:

```
dev = Device(host='xxxx', user='demo', password='demo123', gather_facts=False)
```

Connect to the Junos OS device:

```
dev.open()
```

Enter configuration mode and you are now able to use the configuration utilities. `diff` and `rollback` are two examples of these utilities:

```
cu = Config(dev)
```

Run a `show |compare` command:

```
diff = cu.diff()
```

If there is a candidate configuration, execute a rollback:

```
if diff:
   cu.rollback()
```

Close the connection to the Junos OS device:

```
dev.close()
```

As you can see, this is "EZ." If one has network fundamentals, then PyEZ, is likely a great gateway method to learn Python. Automating networks provides the context that makes learning Python or improving your Python skills a gratifying experience.

Recipe 6 - Automating NETCONF Service Activation

by Ben Dale

- Python Version Used: 3.6
- PyEZ Version Used: N/A
- Junos OS Used: 12.1 -> 15.1
- Juniper Platforms General Applicability: All

Problem

In order for Juniper PyEZ to connect to your devices, you need to configure the NETCONF daemon in Junos to listen for external connections.

Enabling this functionality is simply a matter of applying the following configuration to your device(s):

```
set system services netconf ssh
```

If your network has a significant number of devices, however, logging into every box to enable NETCONF (and thus Junos PyEZ automation) sounds like a job for a lucky, enthusiastic, junior network engineer.

Surely we can automate this?!

This recipe explores the use of the `netmiko` library in Python to execute commands in Junos directly on the CLI and prepare our network for automation with Junos PyEZ.

Solution

First, let's install the `netmiko` module to get underway:

```
pip install netmiko
```

Now a little about `netmiko`. Where PyEZ interacts with Junos via NETCONF, `netmiko` operates by interacting with the Junos CLI (over SSH in this case) sending commands and waiting for a response.

This makes it an extremely useful tool for interacting with devices from vendors that don't provide NETCONF or other API mechanisms for programmatic access:

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

import time
from getpass import getpass
from netmiko import ConnectHandler
from netmiko.ssh_exception import NetMikoTimeoutException, NetMikoAuthenticationException

def enable_netconf(net_device):
  print ("{} Connecting to {}".format(time.asctime(), net_device['ip']))
  junos_device = ConnectHandler(**net_device)   #(5)
  configure = junos_device.config_mode()       #(6)
  print ("{} Applying configuration to {}".format(time.asctime(), net_device['ip']))
  setssns = junos_device.send_command("set system services netconf ssh")    #(7)
  print ("{} Committing configuration to {}".format(time.asctime(), net_device['ip']))
  junos_device.commit(comment='Enabled NETCONF service', and_quit=True)     #(8)
  print ("{} Closing connection to {}".format(time.asctime(), net_device['ip']))
  junos_device.disconnect()                    #(9)

def main():
  user_login = input('Username: ')              #(1)
  user_pass = getpass('Password: ')
  with open('inventory.txt') as f:              #(2)
    device_list = f.read().splitlines()
    for device in device_list:
      net_device = {
        'device_type': 'juniper',             #(3)
        'ip': device,
        'username': user_login,
        'password': user_pass,
      }
      enable_netconf(net_device)             #(4)

if __name__ == '__main__':
   main()
```

Let's walk through this simple script:

1. First, you interactively collect a login and password from the user to be used to connect to each device. You could hard code these credentials, but that would mean storing your password in plain-text inside your code, which is not recommended.

2. Next, the script opens the inventory.txt file. This is simply a list of IP addresses or hostnames of all the devices you wish to execute this script against – with one entry per line, for example:

```
192.168.50.254
192.168.51.254
bne-core-01
```

3. Next, the script iterates through each IP address or hostname, and creates a Python `dict` for each network device, containing the Device Type, IP Address, Username, and Password. The netmiko module will use this information when connecting to each device.

4. Execute the `enable_netconf` function against each device that is generated from our inventory.

5. Connect to the device via SSH.

6. Enter configuration mode within Junos.

7. And send the `set system services netconf ssh` command to the CLI.

8. Commit the configuration to the device.

9. Then close the session.

If no errors were encountered, you will now have a set of devices with NETCONF enabled, ready to be controlled with your first PyEZ script.

Discussion

Creating an inventory file and then enabling NETCONF across your fleet is the first step in your automation journey with PyEZ.

Depending on how many devices you have in your network, you will see that performing this task on each node, sequentially, will become very time consuming.

In fact, on lower-end products such as the SRX Branch, it can take up to 15-20 seconds to authenticate to a device over SSH, apply a configuration, commit the configuration and then close the connection gracefully.

This may not seem like a long time, compared to executing it manually, but imagine you are performing this operation on several hundred or several thousand devices (for example: a large branch network).

Since your Python script is idle most of the time, waiting for updates from your Junos device, surely it could go faster?

In *Recipe 9*, you'll explore the Python `multiprocessing` module and learn how to execute simple tasks such as this in parallel.

Once you complete this chapter, come back to this recipe and try re-writing it so that each call to `enable_netconf()` is executed in parallel.

Recipe 7 - PyEZ with Console: Learning Terminal Server Mapping

by Jessica Garrison

Python Version Used:	2.7
PyEZ Version Used:	2.1.5
NetConify Version Used:	1.0.2
Junos OS Used:	All supported Junos OS releases

- Juniper Platforms General Applicability: Tested on all hardware platforms running Junos OS.

This recipe shows an example of how to learn which Junos device is connected to each terminal server port.

Problem

This recipe could solve multiple issues:

- You need to map out your terminal server ports.
- You're looking for a particular device's console port because you want to zeroize the device and put a new configuration on the device.
- You lost management port connectivity to a device and you need to restore it.
- You are rebooting a device and you want to watch the console port during a reboot.

Solution

Using PyEZ over a console connection is not as fast using an RPC over the management IP address, but the script remains the same:

```
from jnpr.junos import Device
from lxml import etree

print ‘Terminal Server Inventory Report’
count = 7007
while (count < 7033):
   try:
      with Device(host=’10.164.1.233’, user=’root’, password=’Embe1mpls’, mode=’telnet’, port=count,
gather_facts=True) as dev:
         junosinfo = dev.facts
         print ‘Hostname:’ + junosinfo[‘hostname’] + ‘,’ + ‘Hardware:’ + junosinfo[‘model’] + ‘,’ +
‘Software:’ + junosinfo[‘version’] + ‘,’ + ‘TermServPort:’ + str( count )
   except:
      pass
   count = count + 1
```

Discussion

Additional information on the netconify utility can be found at its GitHub location: https://github.com/Juniper/py-junos-netconify.

Recipe 8 - PyEZ with Console: Pushing an Initial Configuration

by Jessica Garrison

■ Python Version Used:	2.7
■ PyEZ Version Used:	2.1.5
■ NetConify Version Used:	1.0.2
■ Junos OS Used:	All supported Junos OS releases

- Juniper Platforms General Applicability: Tested on all hardware platforms running Junos OS.

This recipe supplies an example of how to use PyEZ with a console connection to push a configuration that would provide SSH and NETCONF connectivity.

Problem

How do you push a configuration onto a Junos OS device that has come straight from the factory or has been zeroized?

Solution

Using PyEZ over a console connection is not as fast using an RPC over the management IP address, but the script remains the same:

```
ffrom jnpr.junos import Device
from lxml import etree

print ‘Terminal Server Inventory Report’
count = 7001
while (count < 7033):
   try:
      with Device(host='X.X.X.X', user='root', password='password', mode='telnet', port=count,
gather_facts=True) as dev:
         junosinfo = dev.facts
         print 'Hostname:' + junosinfo['hostname'] + ',' + 'Hardware:' + junosinfo['model'] + ',' +
'Software:' + junosinfo['version'] + ',' + 'TermServPort:' + str( count )
   except:
      pass
   count = count + 1
```

To further explain the Python script we are using a while loop, which increments "count" from 7001 to 7032. These are the 32 ports on the terminal server. Using host X.X.X.X and port 70XX (port = count) provide telnet connectivity via the console port:

```
count = 7001
while (count < 7033):
   count = count + 1
```

This particular script was built to be user friendly, beginner level, and run from the shell. That is why we print out some basic information for the user to see. In reality, the script could be expanded to pull from a csv file or insert into a SQL database.

```
junosinfo = dev.facts
print 'Hostname:' + junosinfo['hostname'] + ',' + 'Hardware:' + junosinfo['model'] + ',' + 'Software:'
+ junosinfo['version'] + ',' + 'TermServPort:' + str( count )
```

The "pass" was included to bypass non-Junos OS devices and not see the Exception handling. This script takes quite some time to run as using Netconify is not as fast as regular PyEZ.

Discussion

Additional information on the Netconify utility can be found at its GitHub location: https://github.com/Juniper/py-junos-netconify.

Recipe 9 - Checking File System Directory Usage on Multiple Devices in Parallel

by Peter Klimai

- Python Version Used: 3.6
- PyEZ Version Used: 2.1.5
- Junos OS Used: Multiple, including 17.1R2.7 and 12.1X47-D40.
- Juniper Platforms General Applicability: All

Performing file system maintenance tasks on many devices can be just plain boring and time-consuming. This recipe offers an example of how to automate work file system tasks using Junos PyEZ. It also uses Python's multiprocessing module to speed up the task execution.

Problem

You want to perform a directory usage check (similar to the standard UNIX `du` utility) on multiple Junos devices for a specific directory. You want to automate the process and get the data as soon as possible.

NOTE You can find additional discussion about, and examples of, multiprocessing with Python and Junos PyEZ in *Recipe 6*, by Ben Dale, in this cookbook.

Solution

Junos PyEZ has multiple useful utilities for working with the Junos device's file system. They are contained in `FS` class of `jnpr.junos.utils.fs` module. The `FS` class includes such methods as `cp()` (local file copy), `ls()` (return directory listing), `mkdir()` (create a directory), and many others. To solve the task of getting directory usage for a specific directory, use the `directory_usage()` method.

NOTE The `directory_usage()` method accepts two optional parameters: `path` (directory path; the default is current directory) and `depth` (default is 0, meaning do not walk the subdirectories). You can find all the details on library methods and their parameters in the PyEZ documentation.

To solve the task, create the automation script that you name `directory_usage_multiprocess.py`. Its source code is presented below, and the script's parts are numbered as `# (n)` for the explanation that follows:

```
#!/usr/bin/python3

from jnpr.junos import Device                                   # (1)
from jnpr.junos.utils.fs import FS
from jnpr.junos.exception import *
import multiprocessing
import time

NUM_PROCESSES = 1                                              # (2)
USER = “lab”
PASSWD = “lab123”
DEVICES = [
   “10.254.0.31”,
   “10.254.0.34”,
   “10.254.0.35”,
   “10.254.0.37”,
   “10.254.0.38”,
   “10.254.0.41”,
   “10.254.0.42”,
]
DIRECTORY = “/var/tmp/”

def check_directory_usage(host):                                 # (3)
   try:
      with Device(host=host, user=USER, password=PASSWD) as dev:
         fs = FS(dev)                                        # (4)
         print(“Checking %s: “ % host, end=””)
         print(fs.directory_usage(DIRECTORY))                  # (5)
   except ConnectRefusedError:                                  # (6)
      print(“%s: Error - Device connection refused!” % host)
   except ConnectTimeoutError:
      print(“%s: Error - Device connection timed out!” % host)
   except ConnectAuthError:
      print(“%s: Error - Authentication failure!” % host)

def main():                                                  # (7)
   time_start = time.time()
   with multiprocessing.Pool(processes=NUM_PROCESSES) as process_pool:  # (8)
      process_pool.map(check_directory_usage, DEVICES)      # (9)
      process_pool.close()                                 # (10)
      process_pool.join()
   print(“Finished in %f sec.” % (time.time() - time_start))  # (11)

if __name__ == “__main__”:                                      # (12)
   main()
```

Here is the explanation of what is happening in the script, using the numeral markers:

1. Import the PyEZ `Device` and `FS` classes, exceptions, and the `multiprocessing` and `time` modules from the standard Python library.

2. Introduce some "constants" including the number of parallel processes (`NUM_PROCESSES`), username, password, list of device IP addresses, and the directory name. Note: Consider replacing some of these hard-coded values with script parameters to make it more production ready.

3. The `check_directory_usage(host)` function will get and print the directory usage for the device with the address given by the `host`. The function begins with the `try` operator, followed by `with` (context manager) syntax for device connection.

4. Inside the context manager for the `Device` instance `dev`, you create an instance of file system (`FS`) class and pass `dev` to it as a parameter.

5. Call `directory_usage()` method providing `DIRECTORY` as a parameter. Print the result.

6. Catch some of the possible exception situations using the `except` operators.

7. The main function. Here, you first store current time to the `time_start` variable. This is just used for displaying information on how long the script ran to the end of execution.

8. Create a pool `process_pool` of `NUM_PROCESSES` processes using context manager syntax (the `with` operator).

9. Use `Pool.map()` method to apply a previously defined function `check_directory_usage()` to each of the entries in the `DEVICES` list. At this point, parallel execution starts (at least if `NUM_PROCESSES` is set to a value larger than 1 – see below).

10. Following best practice recommendations, use `close()` method to notify the process pool that no other work is going to be submitted to it. Use `join()` method to wait for all the worker processes to finish and terminate.

11. Print the time it took to execute the script.

12. Standard Python script "entry point".

Now, let's run the script to see how it works. At this point you have a setting of `NUM_PROCESSES = 1` in the script, which basically means devices are queried in sequence (no parallelization of tasks). You'll get the following results in your terminal:

```
user@ubuntu:~$ ./directory_usage_multiprocess.py
Checking 10.254.0.31: {'/var/tmp/': {'size': '84M', 'blocks': 171992, 'bytes': 88059904}}
Checking 10.254.0.34: {'/var/tmp/': {'size': '112K', 'blocks': 224, 'bytes': 114688}}
Checking 10.254.0.35: {'/var/tmp/': {'size': '223M', 'blocks': 456100, 'bytes': 233523200}}
Checking 10.254.0.37: {'/var/tmp/': {'size': '223M', 'blocks': 456052, 'bytes': 233498624}}
Checking 10.254.0.38: {‘/var/tmp/’: {‘size’: ‘223M’, ‘blocks’: 455952, ‘bytes’: 233447424}}
Checking 10.254.0.41: {‘/var/tmp/’: {‘size’: ‘20K’, ‘blocks’: 40, ‘bytes’: 20480}}
Checking 10.254.0.42: {‘/var/tmp/’: {‘size’: ‘16K’, ‘blocks’: 32, ‘bytes’: 16384}}
Finished in 12.057031 sec.
```

You get the data you needed for each device; directory usage for `/var/tmp` is given in units of bytes, blocks, and in human-readable form. However, in the lab, the script took around 12 seconds for seven devices to execute, but can we do it faster? Yes! Let’s just set `NUM_PROCESSES = 2`, save, and then re-run the script:

```
user@ubuntu:~$ ./directory_usage_multiprocess.py
Checking 10.254.0.34: {‘/var/tmp/’: {‘size’: ‘112K’, ‘blocks’: 224, ‘bytes’: 114688}}
Checking 10.254.0.31: {‘/var/tmp/’: {‘size’: ‘84M’, ‘blocks’: 171992, ‘bytes’: 88059904}}
Checking 10.254.0.35: {‘/var/tmp/’: {‘size’: ‘223M’, ‘blocks’: 456100, ‘bytes’: 233523200}}
Checking 10.254.0.37: {‘/var/tmp/’: {‘size’: ‘223M’, ‘blocks’: 456052, ‘bytes’: 233498624}}
Checking 10.254.0.38: {‘/var/tmp/’: {‘size’: ‘223M’, ‘blocks’: 455952, ‘bytes’: 233447424}}
Checking 10.254.0.41: {‘/var/tmp/’: {‘size’: ‘20K’, ‘blocks’: 40, ‘bytes’: 20480}}
Checking 10.254.0.42: {‘/var/tmp/’: {‘size’: ‘16K’, ‘blocks’: 32, ‘bytes’: 16384}}
Finished in 6.975386 sec.
```

You get the same data, but almost two times faster! This is because you used two independent threads of execution (also known as *workers*).

Now, let’s set `NUM_PROCESSES = 8` (so, with the number of hosts in the `DEVICES` list, no device has to wait in the queue for the others to finish):

```
user@ubuntu:~$ ./directory_usage_multiprocess.py
Checking 10.254.0.35: {‘/var/tmp/’: {‘size’: ‘223M’, ‘blocks’: 456100, ‘bytes’: 233523200}}
Checking 10.254.0.38: {‘/var/tmp/’: {‘size’: ‘223M’, ‘blocks’: 455952, ‘bytes’: 233447424}}
Checking 10.254.0.37: {‘/var/tmp/’: {‘size’: ‘223M’, ‘blocks’: 456052, ‘bytes’: 233498624}}
Checking 10.254.0.41: {‘/var/tmp/’: {‘size’: ‘20K’, ‘blocks’: 40, ‘bytes’: 20480}}
Checking 10.254.0.42: {‘/var/tmp/’: {‘size’: ‘16K’, ‘blocks’: 32, ‘bytes’: 16384}}
Checking 10.254.0.34: {‘/var/tmp/’: {‘size’: ‘112K’, ‘blocks’: 224, ‘bytes’: 114688}}
Checking 10.254.0.31: {‘/var/tmp/’: {‘size’: ‘84M’, ‘blocks’: 171992, ‘bytes’: 88059904}}
Finished in 3.511946 sec.
```

Again, significantly faster! And the speedy effect will become even more obvious if you have dozens, or even hundreds, of devices to query.

Discussion

You’ve seen a simple example of working with device file system utilities using Junos PyEZ. The script execution was parallelized using Python `multiprocessing` library. Generally, parallelization of automation tasks on multiple devices is a very natural idea. Many other scripts presented in this book can benefit from them being modified for multiprocessing in the same way as demonstrated here.

Recipe 10 - Configuring Devices Using Junos PyEZ and Jinja2 Templates

by Scott Ware

Python Version Used:	3.6
PyEZ Version Used:	2.1.5
Junos OS Used:	15.1X49-D100.6
Juniper Platforms General Applicability:	All

This recipe shows an example of how you can configure basic system settings on multiple devices using Junos PyEZ and Jinja2 templates.

Problem

You are tasked with configuring some basic settings, such as a DNS server, NTP server, and SNMP information on numerous firewalls throughout your organization. Some of these settings will be the same across all devices, and certain devices will require a different setting. How do you accomplish this in a timely fashion, so you don't have to log in to each firewall and manually make the changes?

This recipe aims to solve that exact problem by quickly allowing you to configure multiple devices, each with similar *and* device-specific configuration settings, without the need for manual intervention. And, you will be using Jinja2 for your configuration templates. Jinja2 is a powerful template engine that allows you to easily manipulate data in pre-defined templates, such as your configuration file.

Solution

To solve the problem of configuring devices with a same set of configuration values, you first want to create three files: `config.txt`, the configuration template, `firewalls.txt`, a list of devices (IP address or hostname), one per line that you want to configure, and the Python script named `config-devices.py`.

The `config.txt` contains the configuration commands that you want to apply. The template values will be passed from our `config-device.py` script and applied wherever the `{{ variable_name }}` sections are in `config.txt`. Below is what the config.txt file looks like:

```
set system name-server {{ dns_server }}
set system ntp server {{ ntp_server }}
set snmp location "{{ snmp_location }}"
set snmp contact "{{ snmp_contact }}"
set snmp community {{ snmp_community }} authorization read-write
set snmp trap-group snmp-traps targets {{ snmp_trap_recvr }}
```

NOTE For more information on, and examples of, using Jinja2 templates please refer to the Jinja2 documentation: http://jinja.pocoo.org/docs.

Below is the code for our script, `config-devices.py`. Each line in the script is numbered as `#(n)` for the explanation that follows:

```
#1          from jnpr.junos import Device
#2          from jnpr.junos.utils.config import Config
#3
#4          USER = "admin"
#5          PW = "starwars"
#6          CONFIG_FILE = 'config.txt'
#7          CONFIG_DATA = {
#8            'dns_server': '8.8.8.8',
#9            'ntp_server': '24.56.178.140',
#10           'snmp_location': 'Data center core rack',
#11           'snmp_contact': 'IT Security',
#12           'snmp_community': 'snmprw',
#13           'snmp_trap_recvr': '192.168.1.10'
#14         }
#15
#16         def config_devices(devices='firewalls.txt'):
#17           with open(devices, 'r') as f:
#18             firewalls = f.readlines()
#19             firewalls = [x.strip() for x in firewalls]
#20
#21             for firewall in firewalls:
#22               dev = Device(host=firewall, user=USER, password=PW).open()
#23               with Config(dev) as cu:
#24                 cu.load(template_path=CONFIG_FILE, template_vars=CONFIG_DATA, format='set',
merge=True)
#25                 cu.commit(timeout=30)
#26                 print("Committing the configuration on device: {}".format(firewall))
#27               dev.close()
#28
#29         if __name__ == "__main__":
#30           config_devices
```

Below is the explanation of what is happening in this script:

1. Line #1, 2: Import PyEZ `Device` and `Config` classes.

2. Line #4 – 14: Define a set of "constants" that will be used for logging into the devices, as well as configuration data: user, pw, config_file, config_data.

3. Line #16: The function `config_devices()` takes one parameter: `devices`. This is a file with the IP address or hostname of each device you wish to configure, one per line.

4. Line #17 – 19: Open our list of devices. The line immediately following, `firewalls = f.readlines()`, reads the file into a variable called `firewalls`. Line #19, `firewalls = [x.strip() for x in firewalls]`, strips any new line characters, which could pose a problem when connecting to each device.

5. Line #21: Iterate over each device that you want to configure.

6. Line #22 – 25: Open the connection to the device for configuration. The line immediately following: `with Config(dev) as cu:` passes the device connection to the Config class, which allows you to then load your Config in line #24, and commit your changes (line #25).

7. Line #26: Prints a status message to the console that a specific device has been configured.

8. Line #27: Closes our connection to the device.

9. Line #29 – 30: Standard Python script "entry point."

Now you can run the script manually to see if it works properly. You get the following results in the terminal, which tells you that your devices have been configured:

```
user@laptop:~$ python config-devices.py
Committing the configuration on device: corp-fw.company.com
Committing the configuration on device: branch-fw.company.com
```

Congratulations! You've just configured multiple devices with a few keystrokes.

Discussion

This script showed us how to configure multiple devices that share a common set of values. What if you need to configure specific values for each device, across multiple devices?

You can populate a CSV file with a header row that contains your variable names, and each subsequent row can include a device, and its specific settings. Here is a sample CSV file:

```
firewall,dns_server,ntp_server,snmp_location,snmp_contact,snmp_community,snmp_trap_recvr
corp-fw.company.com,8.8.8.8,24.56.178.140,Data center core rack,IT Security,snmprw,1.1.1.1
branch-fw.company.com,4.4.2.2,132.163.4.102,Wiring closet,Network operations,snmpro,2.2.2.2
```

The next function replaces our `config_devices()` function from the previous script. This will take the CSV file as the parameter, and for each row after the top/header, will configure each device with the values you specified in the file. Since we are using Python's CSV library, at the top of your script you need to add the following line: `import csv`, above or below the existing import statements:

```
def config_multi_devices(csv_file='config-data.csv'):
   with open(csv_file) as f:
      csvfile = csv.DictReader(f)

      for row in csvfile:
         firewall = row['firewall']
         values = {
            'dns_server': row['dns_server'],
            'ntp_server': row['ntp_server'],
            'snmp_location': row['snmp_location'],
            'snmp_contact': row['snmp_contact'],
            'snmp_community': row['snmp_community'],
            'snmp_trap_recvr': row['snmp_trap_recvr']
         }

         dev = Device(host=firewall, user=USER, password=PW).open()
         with Config(dev) as cu:
            cu.load(template_path=CONFIG_FILE, template_vars=values, format='set', merge=True)
            cu.commit(timeout=30)
                 print("Committing the configuration on device: {}".format(firewall))
         dev.close()
```

This alternative method provides more flexibility, as well as one central spot to keep track of the configuration data you wish to use. As you can see, using templates is very flexible, and hopefully it will simplify your configuration tasks!

Recipe 11 - Benefits of Taking a Snapshot on ACX Series Access Routers

by Ivan Del Rio Fernandez

Python Version Used:	2.7.3
PyEZ Version Used:	2.1.5
Junos OS Used:	15.1R6.7

- Juniper Platforms General Applicability: ACX Series platform (except the ACX5048 and ACX5096)

Learn why using PyEZ Junos to take a snapshot on the ACX Series Universal Access Routers can be beneficial by disabling or enabling module traces or syslog messages automatically, which may reduce the disk I/O operations during the procedure.

Platform

The ACX Series keeps the primary and backup Junos images in two independently bootable root partitions. *Dual-root* partitioning allows the ACX Series router to remain functional even if there is file system corruption, and facilitates easy recovery of the file system. Maintaining data consistency across router partitions means you won't encounter any unpleasant surprises if the active partition gets corrupted.

Problem

Before taking a snapshot on an ACX Series router, it is recommended to rotate log files and delete unused files, as well as keep traceoptions and syslog disabled to avoid any unnecessary I/O operations while the router copies the data to the alternate partition. Once the process is completed you will need to enable syslog, with or without traceoptions.

These extra tasks can be very time consuming and may add more difficulty to the snapshot procedure, particularly if you want to run them on several routers at the same time:

```
user@node01> request system snapshot slice alternate
System may go unstable if module traces or syslog mesages are enabled during snapshot. It is
recommended to disable all debug logging.
Do you wish to continue? [yes,no] (no) yes
```

NOTE ACX5048 and ACX5096 routers do not support dual-root partitioning. All other ACX routers run with dual-root partitioning.

Solution

Use PyEZ to handle all of these assignments automatically, in an orderly manner, and without an operator to monitor the entire process.

You only need to input the IP of the device that you want to snapshot. Once you launch the script, it will provide detailed information on every step and will abort the process if there are any errors or problems while performing the tasks:

```
user@dev:/home/user# python snapshot.py 10.1.196.77

####################
### Auto-snapshot v1.0 ###
####################

-> Connecting to 10.1.196.77
-> Device platform:  ACX1100
-> Performing system storage cleanup, please be patient
-> Disabling syslog + traceoptions (when enabled)
-> Snapshoting the device..
-> Re-enabling only syslog!

[Content of /etc/dumpdates]

/dev/da0s2a                  0 Thu Aug 31 16:39:33 2017
/dev/da0s2e                  0 Thu Aug 31 16:40:43 2017
```

Two files are used to inject the deactivate/delete/enable statements into the ACX router(s). This method provides flexibility if you want to add or remove any statements without changing code in the script:

```
disable_cmds.txt file:

deactivate system syslog
delete protocols rsvp traceoptions
delete protocols ospf traceoptions
delete protocols ldp traceoptions

enable_syslog.txt file:

activate system syslog
```

```
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
from jnpr.junos.utils.fs import FS
from jnpr.junos.rpcmeta import _RpcMetaExec
from jnpr.junos.exception import *
from lxml import etree
from pprint import pprint
import re
import os, errno
import os.path
import time
import sys
import socket


if len(sys.argv) > 2:
  print(“Please only call me with one parameter”)
  sys.exit(1)

device_ip = sys.argv[1]

try:
      socket.inet_aton(device_ip)
except Exception as err:
      print err
      sys.exit(1)

path = “/home/idelrio/scripts/snapshoter/”;

disable_cmds = path + “disable_cmds.txt”
enable_syslog = path + “enable_syslog.txt”

filename_snap = path +  “files/” + device_ip + “.snap”
filerun = path  + device_ip + “.run”

filelog = path + “/log/” + device_ip + “_” + time.strftime(“%Y-%m-%d_%H:%M:%S”) + “.log”

def close():

       file_log.close()
  os.remove(filename_snap)
  os.remove(filerun)
       j_device.close()

def log(text):

       file_log.write(text)


def my_commit(file_path,device):

       #load snippet configuration
       cfg = Config(device)

       #lock candidate configuration
       cfg.lock()

       #load configuration
```

```

    try:
         cfg.load(path=file_path, format=”set”, merge=True, ignore_warning=True)
    except ConfigLoadError as e:
        if e.rpc_error[‘severity’] == ‘warning’:
          pass
        else:
          raise


    #commit configuration

    if cfg.commit():
      pass
    else:
         print “Failed to commit configuration.Aborting!”
         log(“Failed to commit configuration.Aborting!”)
         close()
         sys.exit(1)

    #unlock configuration
    cfg.unlock()

    ######################
    ######## BEGIN ########
    ######################

    ### open log file ###

    file_log = open(filelog, “w”)

    print “”
    print “########################”
    print “### Auto-snapshot v1.0 ###”
    print “########################”
    print “”

    ### Is there any process that is targeting the same device already running? ###

    if os.path.exists(filerun):
   print “-> There is another script instance targeting same device:”,device_ip
        log(“-> There is another script instance targeting same device”)
        file_log.close()
   sys.exit(1)

    print “-> Connecting to “,device_ip

    # open a connection and establish a NETCONF session with the device
    j_device = Device(host=device_ip, user=’user’, password=’pwd’)
    try:
      j_device.open()
    except Exception as err:
    print “Cannot connect to device:”, err
      close()
      sys.exit(1)

    #increases device connection timeout
    j_device.timeout= 600

    ### Is the device platform ACX1100 ACX2200 ACX2200 ? ###

    if j_device.facts[‘model’] == “ACX2200” or j_device.facts[‘model’] == “ACX1100” or j_device.facts[‘model’] == “ACX2100”:
    print “-> Device platform: “,j_device.facts[‘model’]
    else:
      print “-> (i) Platform device is not ACX1100/ACX2100/ACX2200”
      log(“-> (i) Platform device is neither ACX1100 nor ACX2200 or ACX2100”)
      close()
      sys.exit(1)

    #Inform other scripts what device is the script targeting into.
    frun = open(filerun, “w”)

### Looks for input/output errors from previous snapshot ###

    text_file_snap = open(filename_snap, “w”)
    op = j_device.rpc.file_show(filename=’/var/log/snapshot’)
    text_file_snap.write(etree.tostring(op))
    text_file_snap.close()

    with open(filename_snap) as f:
    for line in f:
          if re.search(‘Input/output error’, line):
             print “-> (e) Device has Input/output errors.Aborting snapshot!”
          log(“-> (e) Device has Input/output error.Aborting snapshot!”)
          close()
          sys.exit(1)

    text_file_snap.close()

    ### request system storage cleanup ###

    print “-> Performing system storage cleanup, please be patient”

    fs = FS(j_device)
    fs.storage_cleanup()

    ### Disables traceoptions + syslog ###

    print “-> Disabling syslog + traceoptions (when enabled)”
    my_commit(disable_cmds,j_device)

    ### Take a snapshot ###

    print “-> Snapshoting the device..”

    try:
      rsp = j_device.rpc.request_snapshot(slice=’alternate’,dev_timeout=600)
    except Exception as err:
   print “--> (e) Error when snapshoting the device..”, err
   log(“--> (e) Error when snapshoting the device.”)

    print “-> Re-enabling only syslog!”
    my_commit(enable_syslog,j_device)

```

```
#179      ### shows contents of dumpdates file ###
#180
#181      dumpd_file = j_device.rpc.file_show(filename='/etc/dumpdates')
#182      dumpf = etree.tostring(dumpd_file, encoding='utf8', method='text')
#183      print ""
#184      print "    [Content of /etc/dumpdates]"
#185      print dumpf
#186      log(dumpf)
#187
#188      ### close ###
#189      close()
```

PyEZ Script Source Code Explanation

## Libraries

```
* Line #1 to #6   Imports PyEZ classes:
Device
Config
FS (Filesystem)
RpcMetaExec (Execute RPC commands)

* Line #16 to #18 Only expect one parameter (IP) when script launches. An error will occur and will
terminate the script execution if 2+ parameters are provided.

* Line #22 to #26 If the IPv4 address string passed to this function is invalid, socket.error will be
raised.

* Line #28 to #36 Define several set of variables such as file names and paths, etc..


      ### Functions ###

* Line #38 to #43 Function close () will terminate the NETCONF session and connection using
the close() method. It also will close all the file handlers used by snapshoter.py script.

* Line #45 to #47 Function log () saves on a text log the progress and return codes for the different
commands/functions used by the script.


* Line #50 to #79 my_commit () function handles several operations that interact with the candidate/
active configuration such as:
  - Lock the configuration
  - Load the configuration changes and handle any errors
  - Commit the configuration
  - Unlock the configuration


      ##### Begin #####

* Line #90 to #101  This is a code routine that checks if there is another script trying to take a
snapshot and/or performing some of the pre/post maintenance tasks. It uses a beacon file such as
10.1.2.3.run to notify other scripts that there is already a process targeting the same device.

* Line #117 to #123 Since you might have more than one platform running on the same network, you want to
```

```
make sure that the script targets only an ACX Router.

* Line #131 to #142 This is a code routine that reviews the file /var/log/snapshot looking for exit
codes of previous snapshots. Using RegEx, search for input/output errors on the nand drive and abort
the script execution if they are found.

* Line #148 to #149 This code frees up storage space on the ACX router by rotating log files and deleting
unused files.

* Line #171This command takes a snapshot of the device.

* Line #181 to #185 These commands show the content of /etc/dumpdates. This file contains useful
information about the system partitions and the last time a snapshot was taken.

[Content of /etc/dumpdates]

/dev/da0s2a                0 Thu Aug 31 16:39:33 2017
/dev/da0s2e                0 Thu Aug 31 16:40:43 2017

     ###### End ######
```

Discussion

This PyEZ script can be useful, particularly as your network expands and the need for automation is necessary because you do not always have enough manpower or time. You can also implement an interface with your network device database that will allow the script to take snapshots automatically, without the need for an operator, especially if there have been commit changes on the active configuration since the last snapshot was taken.

Recipe 12 - Extract Operational Data from Devices Running on a Bridge Domain Environment

by Ivan Del Rio Fernandez

- Python Version Used: 2.7.3
- PyEZ Version Used: 2.1.5
- Junos OS Used: 13.3R8.7
- Juniper Platforms General Applicability: MX240 / MX104

In this recipe, the network access layer is deployed across several locations where you cannot always depend on a reliable power source. In order to maintain continuous power to the equipment, Uninterruptible Power Supplies (UPS) can be utilized (as shown in Figure 1) to provide reliable power and monitoring via SNMP values such as power source quality, battery age, temperature, capacity, etc.

A bridge domain called UPS_Auto will allow the Layer 2 traffic flow between the UPS management card (attached to the edge device) and the DHCP server.

This recipe's script uses Tables and Views, along with other PyEZ functions, to extract operational information data from the MX Series routers. Afterwards, this information is used to create customizable configurations for each UPS that includes site address, edge device hostname, etc., by allowing a completely unmanaged provisioning process for every newly deployed or replaced UPS.

Problem

Every UPS that is deployed on a remote site requires a unique configuration that matches the location details of the edge device.

Collecting all this information becomes a tedious task and can be very time-consuming since the process is completely manual and requires the use of the CLI.

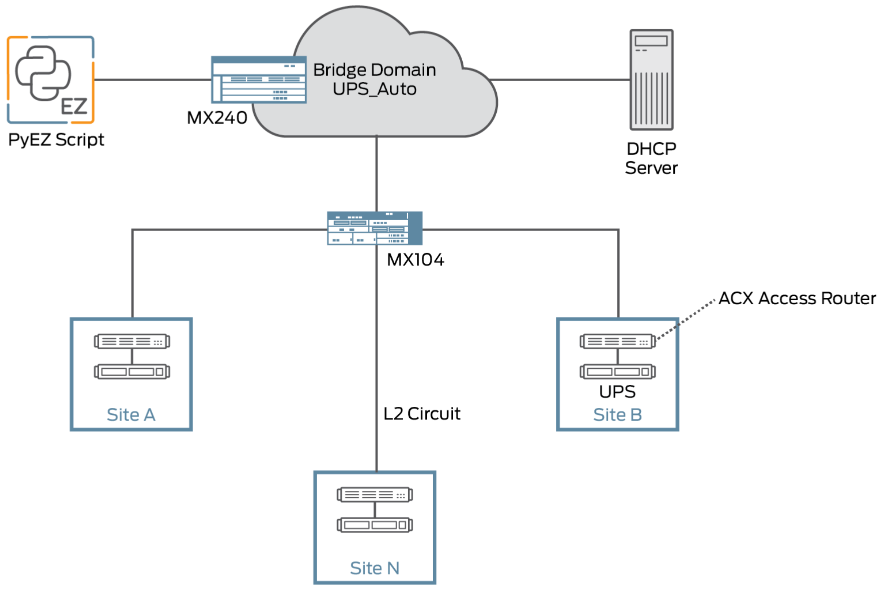


Figure 12.1 Recipe 12's Topology

Solution

Using PyEZ Tables and Views, you can extract operational data from the UPS_Auto bridge domain, which is hosted on the MX240, and the Layer 2 circuits (MX104) that provide Layer 2 connectivity with the edge devices.

A MySQL table will store the operational data and provide an interface that can later be utilized by other script(s) or monitoring tool(s).

Since the UPS does not have Layer 3 visibility of the edge device IP, by using PyEZ Tables and Views, you can extrapolate the VLAN ID from which the UPS MAC address is being learned, and then use this information to discover the Layer 2 circuit remote PE's IP address:

```
idelrio@node1:/home/idelrio# crontab –l
*/5 * * * * /usr/bin/python  /home/idelrio/get_edge_device_details.py 2>&1 >/dev/null
```

Tables and Views

Tables and Views provide a simple and efficient way to extract information from complex operational command output. A Table is associated with a View, which is used to access fields in the Table items. PyEZ Junos already provides predefined templates for different commands. For this recipe use:

- `L2circuit`
- `Ifdesc`
- `Bridge`

Tables and Views are defined using YAML language and can be found in:

```
ivan@node01:/usr/local/lib/python2.7/dist-packages/jnpr/junos/op#
```

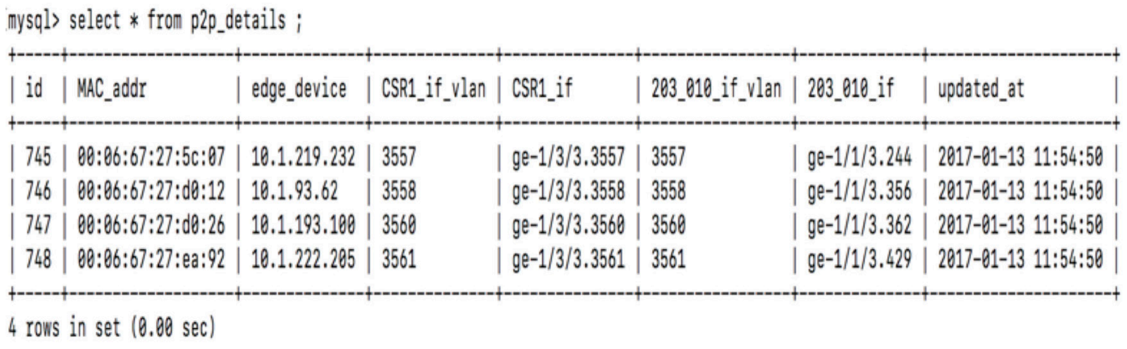
You can see the Tables in YAML format used for this script:

```
l2circuit.yml
---

### ---------------------------------------------------------
### show l2circuit connections
### ---------------------------------------------------------

L2CircuitConnectionTable:
  rpc: get-l2ckt-connection-information
  item: l2circuit-neighbor/connection
  key:
    - ancestor::l2circuit-neighbor/neighbor-address
    - connection-id
  view: L2CircuitConnectionView

L2CircuitConnectionView:
  fields:
    connection_id: connection-id
    connection_type: connection-type
    connection_status: connection-status
    remote_pe: remote-pe
    control_word: control-word
    inboud_label: inbound-label
    outbound_label: outbound-label
    pw_status_tlv: pw-status-tlv
    local_interface: local-interface/interface-name
    interface_status: local-interface/interface-status
    interface_encapsulation: local-interface/interface-encapsulation
    interface_description: local-interface/interface-description

ifdesc.yml
---
IfdescTable:
  rpc: get-interface-information
  args:
    brief: True
    interface_name: '[afgx]e*'
  args_key: interface_name
  item: physical-interface/logical-interface
  view: IfdescView

IfdescView:
  fields:
    name: name
    vlan: link-address
    description: description

bridge.yml
---
BridgeTable:
  rpc: get-bridge-mac-table
  item: l2ald-mac-entry
  key: l2-mac-address
  view: BridgeView

BridgeView:
  fields:
    vlan: l2-bridge-vlan
    mac: l2-mac-address
    interface: l2-mac-logical-interface
    domain: l2-mac-bridging-domain
```

MySQL

This recipe uses a table called `p2p_details` to store all the operational data extracted from the Bridge Domain/Sub-Interfaces/L2circuits:

- `MAC_addr`: UPS Management card Layer 2 address
- `Edge_device`: ACX Router IP (Remote PE)



```
mysql> select * from p2p_details ;
+-----+-------------------+---------------+-------------+---------------+-----------------+---------------+---------------------+
| id  | MAC_addr          | edge_device   | CSR1_if_vlan | CSR1_if      | 203_010_if_vlan | 203_010_if    | updated_at          |
+-----+-------------------+---------------+-------------+---------------+-----------------+---------------+---------------------+
| 745 | 00:06:67:27:5c:07 | 10.1.219.232 | 3557         | ge-1/3/3.3557 | 3557            | ge-1/1/3.244 | 2017-01-13 11:54:50 |
| 746 | 00:06:67:27:d0:12 | 10.1.93.62   | 3558         | ge-1/3/3.3558 | 3558            | ge-1/1/3.356 | 2017-01-13 11:54:50 |
| 747 | 00:06:67:27:d0:26 | 10.1.193.100 | 3560         | ge-1/3/3.3560 | 3560            | ge-1/1/3.362 | 2017-01-13 11:54:50 |
| 748 | 00:06:67:27:ea:92 | 10.1.222.205 | 3561         | ge-1/3/3.3561 | 3561            | ge-1/1/3.429 | 2017-01-13 11:54:50 |
+-----+-------------------+---------------+-------------+---------------+-----------------+---------------+---------------------+
4 rows in set (0.00 sec)
```

Figure 12.2 Table p2p_details

PyEZ script source code

```
#################
### Libraries ###
#################

from jnpr.junos import Device
from jnpr.junos.op.bridge import *
from jnpr.junos.op.ifdesc import *
from jnpr.junos.op.l2circuit import *
from pprint import pprint
import MySQLdb
import MySQLdb.cursors
import os, errno
import re


#######################
### MySQL connector ###
#######################

connx = MySQLdb.Connect(
     host=’10.1.254.124’, user=’ups’,
      passwd=’ups_pwd’, db=’ups’,compress=1,
     cursorclass=MySQLdb.cursors.DictCursor)


#################################################################
### Saves operational commands from the MX240 on MySQL table. ###
#################################################################

def save_csr1_db(mac,interface,vlan,connx):

       query = “insert into p2p_details (MAC_addr,CSR1_if,CSR1_if_vlan) values (‘%s’,’%s’,’%s’)” % (mac,interface,vlan)
       cursor_call(query,connx)


#################################################################
### Saves the preferred output from the MX104 on MySQL table. ###
#################################################################

def save_mx104_db(interface,vlan,rpe,connx):

       query = “update p2p_details set edge_device=’%s’,203_010_if_vlan=’%s’,203_010_if=’%s’ where CSR1_if_vlan=’%s’” % (rpe,vlan,interface,vlan)
       cursor_call(query,connx)


###############################################################
### SQL command to clear the contents of table p2p_details. ###
###############################################################

def drop_table_db(connx):

     query = “delete from p2p_details”
         cursor_call(query,connx)

```

```
####################
### MySQL cursor ###
####################

def cursor_call(query,connx):

       cursorx = connx.cursor()
   cursorx.execute(query)
   connx.commit()
   cursorx.close()


########################
### Close connectors ###
########################

def close():

       dev.close()
       dev1.close()
       connx.close()



#############
### BEGIN ###
#############

dev = Device(host='10.1.2.3', user='user', password='pwd')
dev1 = Device(host='10.1.2.4', user='user', password='pwd' )

       try:
         dev.open()
         dev1.open()

       except ConnectError as err:

         print err
         print err._orig

########################
### Tables and Views ###
#######################

bdomain = BridgeTable(dev);
opt = bdomain.get()

Ifdetails_csr1 = IfdescTable(dev);
opt_interface_csr1 = Ifdetails_csr1.get(interface_name='ge-1/3/3')

Ifdetails_mx104 = IfdescTable(dev1);
opt_interface_mx104 = Ifdetails_mx104.get(interface_name='ge-1/1/3')

l2c = L2CircuitConnectionTable(dev1);
opt_l2c = l2c.get()


#########################################################################
```

```
### Cleans the p2p_details table before proceeding to store new data. ###
##########################################################################

#68 drop_table_db(connx)
#69
```

After you retrieve the Table items, you can treat them like a Python dictionary, which enables you to use methods in the standard Python library to access and manipulate the items.

```
#####################################
### Iterating through the Tables ###
#####################################

#70 for item in opt:
#71
#72        if ( item.domain == “UPS_Auto” ):
#73
#74            for mac,interface in zip(item.mac,item.interface):
#75
#76          if re.match(‘ge-1/3/3’, interface) is not None:
#77
#78            print mac
#79            print interface
#80
#81            for item in opt_interface_csr1:
#82
#83              if re.match(‘.*’ + interface, item.name) is not None:
#84
#85                vlan = re.search(‘Out\(swap \.(.*?)\)’, item.vlan).group(1)
#86                save_csr1_db(mac,interface,vlan,connx)
#87
#88        for item in opt_interface_mx104:
#89
#90        logical_if = item.name;
#91        desc_if = item.description;

### ..[ 0x8100.3555 ] In(pop) Out(push 0x8100.3555) .. ###

#92        if re.match(‘.*0x8100.*’, item.vlan) is not None:
#93          vlan = re.search(‘Out\(push 0x8100\.(.*?)\)’, item.vlan).group(1)
#94          for item in opt_l2c:
#95            print “remote_pe”, item.remote_pe
#96            print “local_interface”, item.local_interface
#97            if ( item.local_interface == logical_if ):
#98                print item.remote_pe;
#99           print item.local_interface;
#100                 save_mx104_db(logical_if,vlan,item.remote_pe,connx)
#101
#102 close()


###########
### END ###
###########
```

PyEZ Script Source Code Explanation

The MX240 (10.1.2.3) hosts the bridge domain that provides Layer 2 connectivity between all the devices connected to the same broadcast domain such as the DHCP server and the UPS appliances.

The MX104 (10.1.2.4) manages the `l2circuits` and provides Layer 2 connectivity from the UPS management cards (connected into the remote PE) to the MX240.The link between the MXs uses sub-interfaces that are VLAN tagged for each `l2circuit` connection.

Line #01 to #04: PyEZ Functions including Tables.

Line #11 to #14: MySQL connector. Provides an interface with the MySQL database where you have provisioned your table that you will use to store the results of the Views.

Line #16 to #19: The save_mx240_db(mac,interface,vlan,connx) function will store the UPS MAC address, and the sub-interface/VLAN Id (MX240) into the database.

Line #21 to #24: Function save_mx104_db(interface,vlan,rpe,connx) saves Items Sub-interface Id, VLAN Id and remote PE IP obtained from Iterating the Tables L2CircuitConnectionTable, IfdescTable and BridgeTable into the database.

Line #31 to #36: In function cursor_call(query,connx) cursor() method is used to instantiate a MySQLCursor object.

Line #56 to #57: Method that will extract operational data from the bridge domain provisioned on the MX240, such as MAC address, Sub-interface Id, VLAN, etc.

Line #72 to #79: Iterate BridgeTable looking for the UPS_Auto bridge domain. Inside this Table you only want to match the information contained on interface ge-1/3/3, which is the one connecting with the MX104/Layer 2 circuits. This object contains the UPS MAC addresses and Sub-interfaces/ VLAN ID among other data.

```
idelrio@mx240> show bridge mac-table bridge-domain UPS_Auto | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/13.3R8/junos">
   <l2ald-rtb-macdb UPS_Auto>
      <l2ald-mac-entry junos:style="brief-rtb">
         <l2-mac-routing-instance>default-switch</l2-mac-routing-instance>
         <l2-mac-bridging-domain>UPS_Auto</l2-mac-bridging-domain>
         <l2-bridge-vlan>2999</l2-bridge-vlan>
         <l2-mac-address>00:00:68:1c:34:98</l2-mac-address>
         <l2-mac-flags>D</l2-mac-flags>
```

Line #81 to #86: This will iterate IfdescTable on the MX240.The VLAN ID obtained here will be used later to match the VLAN ID on the MX104 side of the link. The logical interface ID can change but the VLAN will remain the same on both sides of the link.

```
idelrio@mx240> show interfaces ge-1/3/3 | display xml
<logical-interface>
            <name>ge-1/3/3.3589</name>
            <local-index>937</local-index>
            <snmp-index>1255</snmp-index>
            <description>Auto UPS</description>
            <if-config-flags>
               <iff-up/>
               <iff-snmp-traps/>
               <internal-flags>0x20004000</internal-flags>
            </if-config-flags>
            <link-address junos:format="VLAN-Tag [ 0x8100.3589 ] In(swap .2999) Out(swap .3589) ">[ 
0x8100.3589 ] In(swap .2999) Out(swap .3589) </link-address>
            <encapsulation>VLAN-Bridge</encapsulation>
            <traffic-statistics junos:style="brief">
               <input-packets>6001</input-packets>
               <output-packets>1117349</output-packets>
            </traffic-statistics>
            <filter-information>
            </filter-information>
            <address-family>
               <address-family-name>bridge</address-family-name>
               <mtu>2000</mtu>
               <address-family-flags>
                  <internal-flags>0x4000000</internal-flags>
               </address-family-flags>
            </address-family>
         </logical-interface>
```

Line #88 to #93: Now iterate the IfdescTable on the (MX104 only search for values contained on interface ge-1/1/3 such as VLAN IDs.

```
idelrio@mx104> show interfaces ge-1/1/3  | display xml
<logical-interface>
            <name>ge-1/1/3.734</name>
            <local-index>855</local-index>
            <snmp-index>3181</snmp-index>
            <description>7340425.mn_mgmt_ups_p2p_234_135_V3588</description>
            <if-config-flags>
               <iff-up/>
               <internal-flags>0x0</internal-flags>
            </if-config-flags>
            <link-address junos:format="VLAN-Tag [ 0x8100.3588 ] In(pop) Out(push 0x8100.3588) ">[ 
0x8100.3588 ] In(pop) Out(push 0x8100.3588) </link-address>
            <encapsulation>VLAN-CCC</encapsulation>
            <traffic-statistics junos:style="brief">
               <input-packets>690714</input-packets>
               <output-packets>6233</output-packets>
            </traffic-statistics>
            <filter-information>
            </filter-information>
            <address-family>
               <address-family-name>ccc</address-family-name>
               <mtu>2000</mtu>
               <address-family-flags>
                  <internal-flags>0x402</internal-flags>
               </address-family-flags>
            </address-family>
         </logical-interface>
```

Line #94 to #100: You iterate the L2CircuitConnectionTable Table on the MX104.

Once the local-interface (L2CircuitConnectionTable) value is the same as the logical if (IfdescTable), you save these items the database among the Remote_PE IP using the VLAN variable as a unique ID.

```
idelrio@mx104 > show l2circuit connections | display xml
..
<l2circuit-neighbor>
      <neighbor-address>10.1.28.69</neighbor-address>
      <connection heading="Interface                 Type  St     Time last up          # Up trans">
         <connection-id>ge-1/1/3.506(vc 15011196)</connection-id>
         <connection-type>rmt</connection-type>
         <connection-status>Up</connection-status>
         <last-change>Sep  7 06:22:04 2017
         </last-change>
         <up-transitions>1</up-transitions>
         <remote-pe>10.1.28.69</remote-pe>
         <control-word>No</control-word>
         <inbound-label>449200</inbound-label>
         <outbound-label>364737</outbound-label>
         <pw-status-tlv>No</pw-status-tlv>
         <local-interface>
            <interface-name>ge-1/1/3.506</interface-name>
            <interface-status>Up</interface-status>
            <interface-encapsulation>ETHERNET</interface-encapsulation>
            <interface-description>15733455</interface-description>
         </local-interface>
         <connection-bandwidth heading="VC bandwidth: ">
            <bandwidth>128kbps</bandwidth>
         </connection-bandwidth>
      </connection>
    </l2circuit-neighbor>
```

Discussion

Instead of manually collecting the information required to provision the UPS appliances, using the PyEZ script will provide benefits such as not having to set the UPS IP statically, since you will use the remote PE IP address instead, and this value shouldn't change over time. So you can leave the UPS IP as DHCP-enabled, avoiding the need of pre-staging the UPS before deploying into the field.

One valid scenario is to use the MySQL table as an inter-process communication. Using this method as an interface will allow different tools such as monitoring tools, provisioning tools, and others to access the information.

Recipe 13 - Bandwidth Reservation for MPLS Access Rings

by Paul McGinn

- Python Version Used: 2.7.3
- PyEZ Version Used: 2.1.5
- Junos OS Used: 12.3x52
- Juniper Platforms General Applicability: ACX platform

Capacity planning, and ensuring over utilization does not impede the provisioning of new services, can be a tedious task. This recipe provides reports of the current RSVP reservation for each interface across a predetermined segment of the network. Typically, it would be predominately used for Metro Access Rings.

Problem

MPLS protection mechanisms for LSPs create inconsistencies that previous technologies in this environment did not present. Previously, carriers could reserve bandwidth for each pseudowire on both the main and protection path. MPLS will reserve bandwidth for both primary and secondary paths. Adding to this reservation, depending on your protection method, are bypass and detour tunnels. Getting a clear picture of where RSVP bottlenecks are can help with planning capacity increases.

Solution

Utilizing a custom YAML Table and View in PyEZ will allow users to pull RSVP interface information directly from a system, or from multiple systems, with ease.

Our goal is to automate the collection of data, so this recipe focuses on RSVP interface values, but the example can be used as a template to gather any information you may need with a quick easy adjustment to the script. Let's first `show rsvp interface`:

```
user@acx> show rsvp interface
RSVP interface: 4 active
              Active Subscr- Static     Available  Reserved   Highwater
Interface   State resv   iption  BW        BW        BW        mark
ge-0/1/2.0  Down      0   200%  1000Mbps   2Gbps     0bps      0bps
ge-0/1/3.0  Down      0   200%  1000Mbps   2Gbps     0bps      0bps
xe-0/3/0.0  Up       39   200%  1000Mbps   857.088Mbps 1.14291Gbps 1.64236Gbps
xe-0/3/1.0  Up       49   200%  1000Mbps   680.48Mbps  1.31952Gbps 1.59949Gbps
```

The next step in gathering information is to run the `display xml rpc` command, and look for the XML command to pull the RSVP interface table via NETCONF:

```
user@acx> show rsvp interface | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.3X52/junos">
   <rpc>
      <get-rsvp-interface-information>
      </get-rsvp-interface-information>
   </rpc>
   <cli>
      <banner></banner>
   </cli>
</rpc-reply>
```

With the `>show rsvp interface` command in the CLI we are given the following output. To find the correct XML table to extract the data via NETCONF, utilize a `display xml` flag:

```
user@acx> show rsvp interface | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.3X52/junos">
   <rsvp-interface-information xmlns="http://xml.juniper.net/junos/12.3X52/junos-routing">
      <active-count>4</active-count>
      <rsvp-interface junos:style="brief">
         <interface-name>ge-0/1/2.0</interface-name>
         <index>-1610691704</index>
         <rsvp-status>Down</rsvp-status>
         <rsvp-telink>
            <active-reservation>0</active-reservation>
            <subscription>200</subscription>
            <static-bandwidth>1000Mbps</static-bandwidth>
            <available-bandwidth>2Gbps</available-bandwidth>
            <total-reserved-bandwidth>0bps</total-reserved-bandwidth>
            <high-watermark>0bps</high-watermark>
         </rsvp-telink>
      </rsvp-interface>
      <rsvp-interface junos:style="brief">
         <interface-name>ge-0/1/3.0</interface-name>
         <index>-1610691704</index>
         <rsvp-status>Down</rsvp-status>
         <rsvp-telink>
```

```
            <active-reservation>0</active-reservation>
            <subscription>200</subscription>
            <static-bandwidth>1000Mbps</static-bandwidth>
            <available-bandwidth>2Gbps</available-bandwidth>
            <total-reserved-bandwidth>0bps</total-reserved-bandwidth>
            <high-watermark>0bps</high-watermark>
         </rsvp-telink>
      </rsvp-interface>
      <rsvp-interface junos:style="brief">
         <interface-name>xe-0/3/0.0</interface-name>
         <index>-1610691704</index>
         <rsvp-status>Up</rsvp-status>
         <rsvp-telink>
            <active-reservation>39</active-reservation>
            <subscription>200</subscription>
            <static-bandwidth>1000Mbps</static-bandwidth>
            <available-bandwidth>857.088Mbps</available-bandwidth>
            <total-reserved-bandwidth>1.14291Gbps</total-reserved-bandwidth>
            <high-watermark>1.64236Gbps</high-watermark>
         </rsvp-telink>
      </rsvp-interface>
      <rsvp-interface junos:style="brief">
         <interface-name>xe-0/3/1.0</interface-name>
         <index>-1610691704</index>
         <rsvp-status>Up</rsvp-status>
         <rsvp-telink>
            <active-reservation>49</active-reservation>
            <subscription>200</subscription>
            <static-bandwidth>1000Mbps</static-bandwidth>
            <available-bandwidth>680.48Mbps</available-bandwidth>
            <total-reserved-bandwidth>1.31952Gbps</total-reserved-bandwidth>
            <high-watermark>1.59949Gbps</high-watermark>
         </rsvp-telink>
      </rsvp-interface>
   </rsvp-interface-information>
   <cli>
      <banner></banner>
   </cli>
</rpc-reply>
```

Each value in the XML is available for extraction, but for this recipe you should be concerned with pulling only a few values: *Interface Name*, *Status*, *Active Reservation*, *Subscription*, and *Available Bandwidth*. This is where you benefit from utilizing PyEZ because you can extract and report the values you are looking for without the need to parse through all the output data.

Mapping values is a straightforward process. Available bandwidth in XML is a value under `rsvp-interface/ rsvp=telink/ available-bandwidth`. Translating this to YAML is `rsvp_telink_available_bandwidth`. All hypens, "-" as reported in XML output are converted to underscores "_":

```
1. Rsvp_int_Table:
2.  rpc: get-rsvp-interface-information
3.  item: rsvp-interface
4.  key: interface-name
5.  view: RsvpIntView
6. RsvpIntView:
7.  fields:
8.   interface_name: interface-name
9.   rsvp_status: rsvp-status
10.  rsvp_telink_active_reservation: rsvp-telink/active-reservation
11.  rsvp_telink_subscription: rsvp-telink/subscription
12.  rsvp_telink_available_bandwidth: rsvp-telink/available-bandwidth
13.  rsvp_telink_total_reserved_bandwidth: rsvp-telink/total-reserved-bandwidth
14.  rsvp_telink_high_watermark: rsvp-telink/high-watermark
```

Let's break down the this YAML for further clarification. The most important variables that you are concerned with are 1, 2, and 6–14:

1. Table name.
2. RPC, earlier the XML `rpc` flag was used to give us the appropriate command to pull the RSVP interface table.
6. RsvpIntView: User defined.
8-14. These fields are the data that you would like to extract for use. They can be added to, or subtracted from, based on your use case.

Once the YAML definition is completed, you set up a file for data collection:

```
# Setup Output File

_rsvpout = "/var/tmp/rsvp.out"

def log(text):

      file_log.write(text)

file_log = open(_rsvpout, "w")
```

Now create the function to connect and pull the desired data:

```
def get_rsvp(_IP1):
           try:
             _dev = Device(host=_IP1,user=_uname,password=_upass)
             _dev.open()
           except Exception as err:
             print "Can not Connect" + _IP1, err
             return
             sys.exit(1)

           rsvp_int_T = Rsvp_int_Table(_dev)
           rsvp_int_T.get()
           for rsvp in rsvp_int_T:
             file_log.write(_IP1 + ";" + rsvp.interface_name + ";" + rsvp.rsvp_status + ";" + rsvp.
rsvp_telink_active_reservation + ";" + rsvp.rsvp_telink_subscription + ";" + rsvp.rsvp_telink_
available_bandwidth + '\n')
             #print _IP1 + ";" + rsvp.interface_name + ";" + rsvp.rsvp_status + ";" + rsvp.rsvp_telink_
```

```
active_reservation + “;” + rsvp.rsvp_telink_subscription + “;” + rsvp.rsvp_telink_available_bandwidth
+ “\n”

          _dev.close()
```

To collect data across a network segment, use a simple text file of IP addresses:

```
directory = os.path.normpath(“/path_to_text_file/IP_Addresses_In_Segment”)
```

And finally, walk the directory for any text file, allowing you to report several segments at a time:

```
for subdir, dirs, files in os.walk(_directory):
          for file in files:
            if file.endswith(“.txt”):
              with open(os.path.join(subdir, file),’r’) as _IP1:
                for line in _IP1:
                  line = line.strip()
                  get_rsvp(line)
```

```
#!/usr/bin/python

import os, sys, yaml, glob, errno
from pprint import pprint
from jnpr.junos import Device
from jnpr.junos.factory.factory_loader import FactoryLoader
from jnpr.junos.op.ethport import EthPortTable
from jnpr.junos.rpcmeta import _RpcMetaExec
from lxml import etree

# U&P
_uname = “user”
_upass = “pass”

# YAML table for RSVP
RSVP_INT_yml = ‘’’
---
Rsvp_int_Table:
 rpc: get-rsvp-interface-information
 item: rsvp-interface
 key: interface-name
 view: RsvpIntView
RsvpIntView:
 fields:
  interface_name: interface-name
  rsvp_status: rsvp-status
  rsvp_telink_active_reservation: rsvp-telink/active-reservation
  rsvp_telink_subscription: rsvp-telink/subscription
  rsvp_telink_available_bandwidth: rsvp-telink/available-bandwidth
  rsvp_telink_total_reserved_bandwidth: rsvp-telink/total-reserved-bandwidth
  rsvp_telink_high_watermark: rsvp-telink/high-watermark
‘’’

globals().update(FactoryLoader().load(yaml.load(RSVP_INT_yml)))
```

```
# Setup Output File

_rsvpout = “/var/tmp/rsvp.out”

def log(text):

      file_log.write(text)

file_log = open(_rsvpout, “w”)

def get_rsvp(_IP1):
          try:
            _dev = Device(host=_IP1,user=_uname,password=_upass)
            _dev.open()
          except Exception as err:
            print “Can not Connect” + _IP1, err
            return
            sys.exit(1)

          rsvp_int_T = Rsvp_int_Table(_dev)
          rsvp_int_T.get()
          for rsvp in rsvp_int_T:
            file_log.write(_IP1 + “;” + rsvp.interface_name + “;” + rsvp.rsvp_status + “;” + rsvp.
rsvp_telink_active_reservation + “;” + rsvp.rsvp_telink_subscription + “;” + rsvp.rsvp_telink_
available_bandwidth + ‘\n’)
            #print _IP1 + “;” + rsvp.interface_name + “;” + rsvp.rsvp_status + “;” + rsvp.rsvp_telink_
active_reservation + “;” + rsvp.rsvp_telink_subscription + “;” + rsvp.rsvp_telink_available_bandwidth
+ “\n”

          _dev.close()

directory = os.path.normpath(“/path_to_text_file/IP_Addresses_In_Segment”)

for subdir, dirs, files in os.walk(_directory):
          for file in files:
            if file.endswith(“.txt”):
              with open(os.path.join(subdir, file),’r’) as _IP1:
                for line in _IP1:
                  line = line.strip()
                  get_rsvp(line)
```

Conclusion

PyEZ allows quick, customizable data collection. The ability to extract specific information from a system creates a simplified scripting environment, providing you with live network resource reporting with minimal effort. Cool.

Recipe 14 - Adding a Graphical Interface to the PyEZ Script

by Peter Klimai

Python Version Used:	3.5
PyEZ Version Used:	2.1.4
Junos OS Used:	12.1X47-D40
Juniper Platforms General Applicability:	All

Running your Junos PyEZ automation scripts from the command line is a rather common practice. However, sometimes you want to supply your scripts with a graphical interface for additional convenience, visibility, and usability. The aim of this recipe is to illustrate that this is really not that hard to do.

Problem

You want to be able to collect different sorts of information from your Junos devices, such as:

- Output of PyEZ device "facts"
- BGP summary information
- Interfaces' state information.

And you want to do it easily, with a single click of the mouse button. The graphical user interface (GUI) of the script should assume some default values for device management IP, login, and password – but also give you the flexibility to change that on the fly, too.

Solution

To solve the problem of this recipe, you use the Python `tkinter` package. Tkinter is a part of Python's standard library and, as Python, it is cross-platform – the same GUI script will work on Linux, Mac, Windows, etc.

NOTE For more information on Tkinter, consult its documentation at: https://docs.python.org/3/library/tk.html.

The script that solves the task is given below, and the script's parts are marked with numerals `# (n)` for the explanation that follows:

```
#!/usr/bin/python3

from tkinter import *                                # (1)
from jnpr.junos import Device
from jnpr.junos.exception import *
from pprint import pformat

USER = "lab"                                        # (2)
PASSWD = "lab123"
DEVICE_IP = "10.254.0.35"

def output(st):                                      # (3)
    text.insert(END, chars=st)
    text.see(END)

def read_and_display(message, function):                  # (4)
    output(message)
    try:
        with Device(host=entry_dev.get(), user=entry_user.get(),
                    password=entry_pw.get()) as dev:
            res = function(dev)
    except ConnectRefusedError:
        print("\nError: Connection refused!\n")
    except ConnectTimeoutError:
        output("\nConnection timeout error!\n")
    except ConnectUnknownHostError:
        output("\nError: Connection attempt to unknown host.\n")
    except ConnectionError:
        output("\nConnection error!\n")
    except ConnectAuthError:
        output("\nConnection authentication error!\n")
    else:
        output(res)

def print_facts():                                    # (5)
    read_and_display("\nDevice facts:\n", lambda dev: pformat(dev.facts))

def show_bgp():                                      # (6)
    read_and_display("\nBGP summary information:",
                     lambda dev: dev.rpc.get_bgp_summary_information({"format": "text"}).text)

def show_intf():                                     # (7)
    read_and_display("\nInterface information:",
                     lambda dev: dev.rpc.get_interface_information(
            {“format”: “text”}, terse=True).text)

def main():                                         # (8)
    global entry_dev, entry_user, entry_pw, text
    root = Tk()                                      # (9)

    Frame(root, height=10).grid(row=0)                    # (10)

    Label(root, text=”Device address:”).grid(row=1, column=0)  # (11)
    entry_dev = Entry(root)                               # (12)
    entry_dev.grid(row=1, column=1)
    entry_dev.insert(END, DEVICE_IP)

    Label(root, text=”Login:”).grid(row=2, column=0)         # (13)
    entry_user = Entry(root)
    entry_user.grid(row=2, column=1)
    entry_user.insert(END, USER)

    Label(root, text=”Password:”).grid(row=3, column=0)      # (14)
    entry_pw = Entry(root, show=”*”)
    entry_pw.grid(row=3, column=1)
    entry_pw.insert(END, PASSWD)

    Frame(root, height=10).grid(row=4)                       # (15)
    Button(root, text=”Read facts!”, command=print_facts).grid(row=5, column=0)
    Button(root, text=”Show interfaces!”, command=show_intf).grid(row=5, column=1)
    Button(root, text=”Show BGP!”, command=show_bgp).grid(row=5, column=2)
    Frame(root, height=10).grid(row=6)

    frame = Frame(root, width=800, height=700)              # (16)
    frame.grid(row=7, column=0, columnspan=4)
    frame.grid_propagate(False)
    frame.grid_rowconfigure(0, weight=1)
    frame.grid_columnconfigure(0, weight=1)
    text = Text(frame, borderwidth=3)
    text.config(font=(“courier”, 11), wrap=’none’)
    text.grid(row=0, column=0, sticky=”nsew”, padx=2, pady=2)

    scrollbarY = Scrollbar(frame, command=text.yview)        # (17)
    scrollbarY.grid(row=0, column=1, sticky=’nsew’)
    text[‘yscrollcommand’] = scrollbarY.set
    scrollbarX = Scrollbar(frame, orient=HORIZONTAL, command=text.xview)
    scrollbarX.grid(row=1, column=0, sticky=’nsew’)
    text[‘xscrollcommand’] = scrollbarX.set

    root.mainloop()                                      # (18)

if __name__ == “__main__”:                                 # (19)
    main()
```

Here is an explanation of what is happening in the script:

1. Import all `tkinter` objects, such as `Tk`, `Frame`, `Label`, etc. They are used below. Also, import Junos PyEZ `Device` class, and all PyEZ exceptions. Additionally, import `pformat` function.

2. Username, password, and device IP address that will be used as default values when the script starts (can be changed by user at the script run-time).

3. The `output()` function is basically used to output some data in the script's text field (the `text` object). The `insert()` method adds characters to the end of the output and the `see()` method scrolls the output to the end.

4. The `read_and_display()` function opens the connection to the Junos device using IP address, username, and password as specified by the corresponding GUI entry fields. It then performs actions specified by function `function`, that is passed as a parameter, and outputs the information received. It tries to catch common errors using `try/except` operators.

5. The `print_facts()` is used to collect and output Junos PyEZ device facts. It will be called once the corresponding GUI button is clicked. Note how the `lambda` (anonymous function) syntax is used here.

6. The `show_bgp()` function is similar to the previous one, but is used to collect text output of `show bgp summary` command from the device.

7. The `show_intf()` function is similar and returns text output of the `show interfaces terse` command.

8. The `main()` function starts with the definition of some global variables. Here, `entry_dev`, `entry_user`, and `entry_pw`, will represent user entry text fields, and `text` is the output text object.

9. The `root` object is going to be the main window, everything else is placed inside it.

10. Here, inside `root`, you create an unnamed frame and put it to the grid in the 0-th row (at this point, start thinking columns and rows of the imaginary grid inside a main window). The frame at this point is created just to have some spare space between window edge and other elements that are added later.

11. Create a `Label` (with output text "`Device address:`") and put it into the grid in row 1, column 0.

12. Create a user entry field named `entry_dev` to be used for entering the device IP address. Put it on the grid and use `DEVICE_IP` as a default value.

13. Create a label and an entry field for the username.

14. Create a label and an entry field for the password. Argument `show="*"` ensures that password is not displayed.

15. Create a couple of empty frames and three buttons, and put all that into the grid. Each button uses a different `command` parameter, which will determine what happens when the button is pressed (one of the corresponding functions, defined previously, is called).

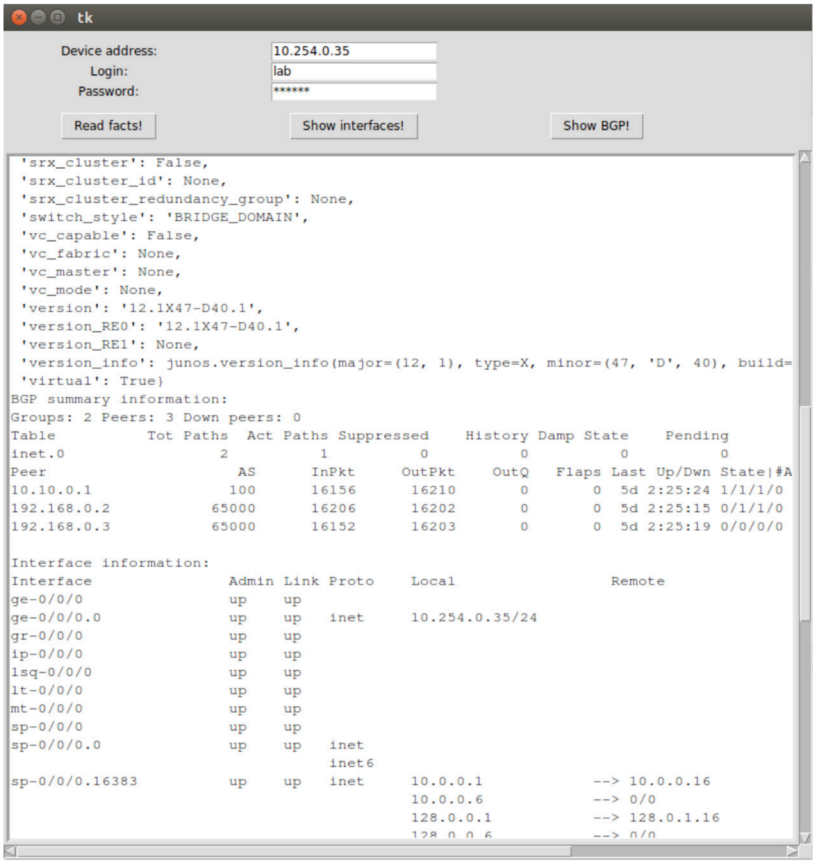
16. Create another frame with a specified size, this time giving it a name (`frame`) and putting a text field `text` inside it.

17. Attach X and Y scrollbars to the text field.

18. Display the main window by calling the `mainloop()` method.

19. Standard Python script "`entry point`".

When you run the script, you will see a window similar to one shown in Figure 14.1. Clicking the buttons provides you with the information gathered from the device, as requested by the task.



Device address: 10.254.0.35
Login: lab
Password: ******

```
 'srx_cluster': False,
 'srx_cluster_id': None,
 'srx_cluster_redundancy_group': None,
 'switch_style': 'BRIDGE_DOMAIN',
 'vc_capable': False,
 'vc_fabric': None,
 'vc_master': None,
 'vc_mode': None,
 'version': '12.1X47-D40.1',
 'version_RE0': '12.1X47-D40.1',
 'version_RE1': None,
 'version_info': junos.version_info(major=(12, 1), type=X, minor=(47, 'D', 40), build=
 'virtual': True}
BGP summary information:
Groups: 2 Peers: 3 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0                 2          1          0          0          0          0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State|#A
10.10.0.1               100      16156      16210       0       0  5d 2:25:24 1/1/1/0
192.168.0.2           65000      16206      16202       0       0  5d 2:25:15 0/1/1/0
192.168.0.3           65000      16152      16203       0       0  5d 2:25:19 0/0/0/0

Interface information:
Interface               Admin Link Proto    Local                 Remote
ge-0/0/0                up    up
ge-0/0/0.0              up    up   inet     10.254.0.35/24
gr-0/0/0                up    up
ip-0/0/0                up    up
lsq-0/0/0               up    up
lt-0/0/0                up    up
mt-0/0/0                up    up
sp-0/0/0                up    up
sp-0/0/0.0              up    up   inet
                                   inet6
sp-0/0/0.16383          up    up   inet     10.0.0.1            --> 10.0.0.16
                                            10.0.0.6            --> 0/0
                                            128.0.0.1           --> 128.0.1.16
                                            128.0.0.6           --> 0/0
```

Figure 14.1 The GUI Example Result of Running the Script from Recipe 14

Discussion

As you can see from Figure 15.1, building a simple GUI for your PyEZ script is not a very complex task. The example presented here can easily be extended to add more functions, including configuration changes, etc.

This recipe is demonstrated using Tkinter, which is Python's *standard* GUI library. Multiple alternatives to Tkinter exist (see https://wiki.python.org/moin/GuiProgramming for review). Also, a script could be called from a web page, if a web server is properly configured – but in the approach demonstrated here, no web server was necessary.

Recipe 15 - Monitoring IPSEC Tunnels

by Michel Tepper & Jac Backus

Python Version Used:	3.5
PyEZ Version Used:	2.1.4
Junos OS Used:	12.1X47-D20.7

- Juniper Platforms General Applicability: SRX, vSRX

Problem

You have a customer with IPsec VPNs to “other vendors.” Sometimes the other side screws up the tunnel, and on the SRX Series side a “clear security like security-association” is necessary. The customer doesn’t want to log in to the SRX, and to be honest, neither do you.

Solution

The automation goal is to have a list of active tunnels in a GUI and select one to reset.

This problem was broken into sub-problems and it was decided the first task was to get a list of active tunnels shown by a Python script. The first part of the script gives it the most important aspect of scripting: reuse of other scripts already invented:

```
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
from lxml import etree
```

```
import dill
import xmltodict
from io import StringIO
from contextlib import redirect_stdout
from collections import defaultdict
from operator import itemgetter
```

The second step sets the variables used to connect to the device:

```
host = ‘172.27.75.5’
user=”root”
passwd=”root123”
```

In a live solution, you don’t want to use the `root` account, of course, so the best thing to do is define a class with just enough permissions to perform your task, assign this class to the user, and then use this user.

CAUTION Safety first, especially when you list the password in the script as done here. A more secure method would be to ask the user to enter the password.

Since this script is going to be expanded in the future, the first step (listing the tunnels) is defined in the function main:

```
def main():
            dev = Device(host=host, user=user, passwd=passwd )
            # open a connection with the device and start a NETCONF session
            try:
                        dev.open()
                        dev.timeout = 300
            except Exception as err:
                        print (“Cannot connect to device:”, err)
                        return
```

If everything is working, and the SRX at IP 172.27.75.5 accepts the netconf-shh session, an open connection is established. The next step for this script is to recursively load active IPsec V tunnels into memory. For this, let’s make the IKE Phase I and Phase II information available, done with library calls:

```
     ikepeers = dev.rpc.get_ike_security_associations_information(detail=True)
     ipsecpeers = dev.rpc.get_security_associations_information(detail=True)
     iketree = etree.ElementTree(ikepeers)
     ipsectree = etree.ElementTree(ipsecpeers)
     ikeroot=iketree.getroot()
     ipsecroot=ipsectree.getroot()
     ikeactions = iketree.findall(‘.//ike-security-associations-block’)
     ikeparsed = [{field.tag: field.text for field in action} for action in ikeactions]
     ipsecactions = ipsectree.findall(‘.//ipsec-security-associations-block’)
     ipsecparsed = [{field.tag: field.text for field in action} for action in ipsecactions]
     d = defaultdict(dict)
```

Now that it’s available, time to parse and print it:

```
            for elem in ipsecparsed:
                        d[elem[‘sa-tunnel-index’]].update(elem)
            l3 = sorted(d.values(), key=itemgetter('sa-vpn-name'))
            for elem in ikeparsed:
                        for elem1 in l3:
                                    if elem1['sa-remote-gateway'] == elem['ike-sa-remote-address']:
                                                elem1.update(elem)
            for idx, tunnel in enumerate (l3):
                        try:
                                    print ((idx + 1),' - ',tunnel['sa-vpn-name'],' - ',tunnel['sa-
remote-gateway'],'- ',tunnel['ike-sa-index'],'- ',tunnel['sa-tunnel-index'],' - ',
"\n\t",tunnel['sa-local-identity'],' - ',tunnel['sa-remote-identity'],'\n' )
                        except KeyError:
                                    print ((idx + 1),' - ',tunnel['sa-vpn-name'],' - ',tunnel['sa-
remote-gateway'],'- ',tunnel['sa-tunnel-index'],' - ', '\n\t',tunnel['sa-local-identity'],' -
',tunnel['sa-remote-identity'],'\n' )
```

The `for` loop repeats this for the number of tunnels found earlier in the script. The print lines may look difficult, but they are mostly formatting. Please be aware that in the script the whole print statement is one line, it just doesn't fit on a page when printing it.

So, what's left to the solution is to close the connection to the device and invoke the `main` function, previously defined, the last two lines:

```
# End the NETCONF session and close the connection
            dev.close()

main()
```

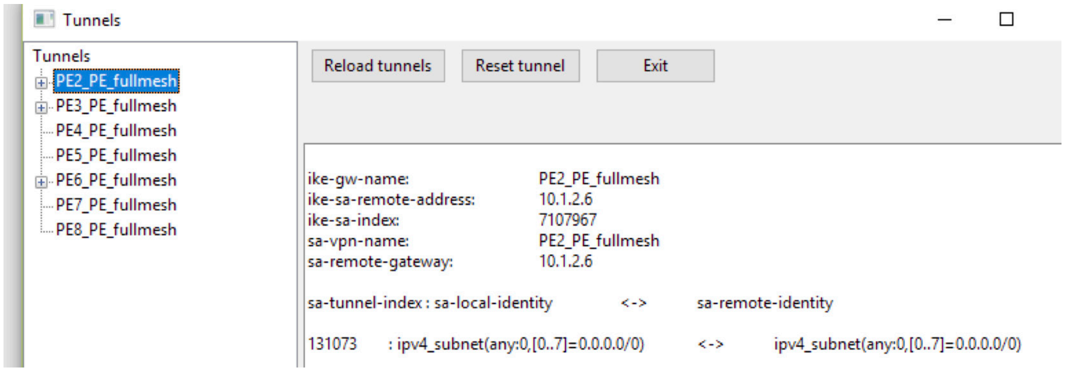
Let's have a look at the outcome of this script when running. It's used to connect to a vSRX on IP address 172.27.75.5, as you saw. This vSRX has IPsec tunnels to seven other vSRXs (PE2 to PE8, this one is PE1), but not all are up. It's part of a full mesh VPN between those eight devices. The goal was only to show the active tunnels; here we go:

```
C:\Python36>py ike-tunnels.py
1  -  PE2_PE_fullmesh  -  10.1.2.6 -  7107967 -  131073  -
       ipv4_subnet(any:0,[0..7]=0.0.0.0/0)  -  ipv4_subnet(any:0,[0..7]=0.0.0.0/0)

2  -  PE4_PE_fullmesh  -  10.1.2.22 -  7108049 -  131075  -
       ipv4_subnet(any:0,[0..7]=0.0.0.0/0)  -  ipv4_subnet(any:0,[0..7]=0.0.0.0/0)

3  -  PE5_PE_fullmesh  -  10.1.2.34 -  7108053 -  131076  -
       ipv4_subnet(any:0,[0..7]=0.0.0.0/0)  -  ipv4_subnet(any:0,[0..7]=0.0.0.0/0)

4  -  PE7_PE_fullmesh  -  10.1.2.50 -  7108052 -  131078  -
       ipv4_subnet(any:0,[0..7]=0.0.0.0/0)  -  ipv4_subnet(any:0,[0..7]=0.0.0.0/0)
```

You can see the gateway name, remote IP address, IK security association, and IPsec security association. The second line gives the proxy IDs, but hey,—those are route-based VPNs—so all zeros are expected.

We achieved our first goal in the lab. And how you want to build your GUI is up to you. The GUI in Figure 15.1 used *wxWidgets* to create a Windows interface around the first step. It was another 450 more lines of code, beyond the scope of this recipe, but this is how it can look when you are building your own GUI.

Pretty nice, all Python scripts with libraries.



```
ike-gw-name:               PE2_PE_fullmesh
ike-sa-remote-address:     10.1.2.6
ike-sa-index:              7107967
sa-vpn-name:               PE2_PE_fullmesh
sa-remote-gateway:         10.1.2.6

sa-tunnel-index : sa-local-identity        <->      sa-remote-identity

131073     : ipv4_subnet(any:0,[0..7]=0.0.0.0/0)        <->        ipv4_subnet(any:0,[0..7]=0.0.0.0/0)
```

Figure 15.1 WxWidgets Used to Create a GUI in Recipe 15

Recipe 16 - Working with Junos Enhanced Auto-Provision Process (JEAP)

by Michelle Zhang

- Python Version Used: 2.7.12
- PyEZ Version Used: 2.1.5
- Junos OS Used: 15.1X49-D60.7
- Juniper Platforms General Applicability: Branch SRX Series and EX Series

Traditionally, the deployment of equipment is a large part of the initial operational spend (OPEX). With JEAP, you can not only reduce the complexity of deployment, you can also virtually eliminate the need for installation services for CPE devices at branch locations.

Problem

When deploying a handful of new devices, manually configuring each device is both time-consuming and repetitive, most importantly, it can cost a lot of money. The larger the network, the more of a problem it becomes.

Solution

JEAP allows Juniper SRX firewalls and EX switches to be shipped directly to customer sites with a factory defaulted configuration. Upon the arrival of the Juniper SRX and EX devices, you simply need to unbox the device, power it on, and plug it in to the existing network. The devices will automatically be configured by a local server with a specific Junos image and configuration.

NOTE JEAP requires a local DHCP server with administrator access and a file transfer server.

JEAP is a ZTP (Zero Touch Provisioning) tool designed for branch offices. It's initially based on the autoinstallation feature on branch SRXs, and it was expanded to the EX series with a bit modification. PyEZ scripts come into play after the autoinstallation process is done – it will reach out to each SRX and EX, do image upgrades or downgrades when necessary, and push certain configuration files.

For the autoinstallation function to work, you need to set up a local DHCP server and a file transfer server, and choices can be a TFTP (Trivial File Transfer Server), FTP (File Transfer Server), or an HTTP (Hypertext Transfer Protocol) server. (More details can be found here on the file transfer server: https://www.juniper.net/documentation/en_US/junos-space-apps/connectivity-services-director1.0/topics/topic-map/autoinstallation-acx-srx-series-using-junos-space.html.) Below is a snippet of the key configuration file for DHCP server used in this example. (Modifications have been made to cover EX Series switches based on their standard ZTP configuration). Figure 16.1 shows the JEAP workflow including the DHCP server.

NOTE The Junos EX image configurations for ZTP have been commented out in order to match the auto-installation feature on the SRX.

CAUTION Due to the differences in their configuration files, you need to be able to tell the SRX and EX devices apart. In this case, you can take advantage of the vendor information contained in the EX's initial broadcast message (the SRX doesn't contain any extra information in its message).

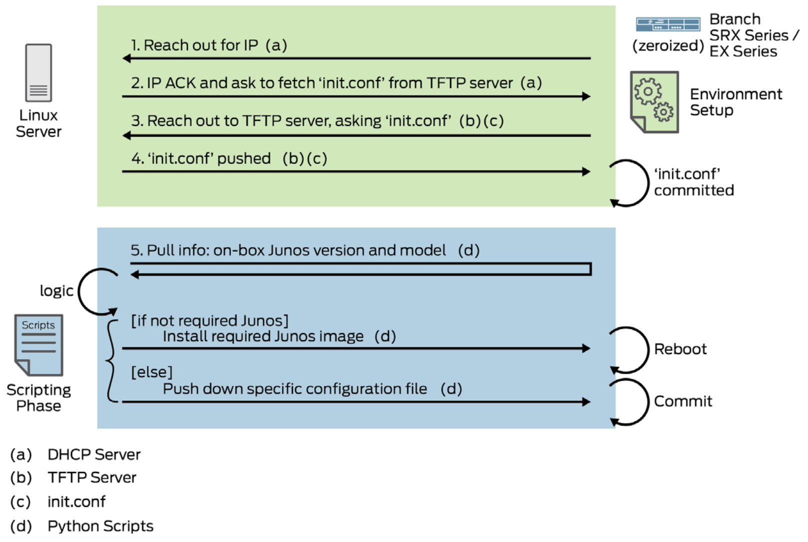

Linux Server
1. Reach out for IP (a)
2. IP ACK and ask to fetch 'init.conf' from TFTP server (a)
3. Reach out to TFTP server, asking 'init.conf' (b)(c)
4. 'init.conf' pushed (b)(c)
Branch SRX Series / EX Series
(zeroized)
Environment Setup
'init.conf' committed
5. Pull info: on-box Junos version and model (d)
logic
Scripts
Scripting Phase
[if not required Junos]
Install required Junos image (d)
[else]
Push down specific configuration file (d)
Reboot
Commit
(a) DHCP Server
(b) TFTP Server
(c) init.conf
(d) Python Scripts

Figure 16.1 JEAP Workflow

```
#### For EX ZTP   ####
```

```
option space ZTP_OP;

option ZTP_OP.config-file-name code 1 = text;
option ZTP_OP.transfer-mode code 3 = text;

option ZTP_OP-encapsulation code 43 = encapsulate ZTP_OP;
option option-150 code 150 = ip-address;

subnet 192.168.222.0 netmask 255.255.255.0 {
}

##########################
### For Class Division ###
##########################

class "EX2200"{
  match if substring (option vendor-class-identifier,0,14) = "Juniper-ex2200";
}

subnet 10.0.1.0 netmask 255.255.255.0 {
           option broadcast-address 10.0.1.255;
           option routers 10.0.1.1;
  pool{
           allow members of "EX2200";
           range 10.0.1.10 10.0.1.20;
           option option-150 10.0.1.1;
           option ZTP_OP.config-file-name "init_EX2200.conf";
#          option ZTP_OP.transfer-mode "ftp";
#          option ZTP_OP.image-file-name  "pub/jinstall-ex-2200-14.1X53-D26.2-domestic-signed.tgz"
#          option ZTP_OP.image-file-type "symlink";
     }
  pool{
           allow unknown-clients;
           range 10.0.1.50 10.0.1.60;
           next-server 10.0.1.1;
           filename "init_SRX.conf";
     }
}
```

By the time the auto installation process is done, Juniper devices will have obtained an IP address and have NETCONF and ssh enabled, under [systems services], for later PyEZ accessibility (if NETCONF and SSH do not come with factory default settings, you need to make sure they are included in the initial configuration file pushed down through the auto-installation function).

CAUTION For the SRX you need to enable NETCONF and SSH in the host-inbound-traffic, too, with the related security zone to which the interface connected to the DHCP server belongs.

Now the JEAP scripts come into play. They read a lease file in the DHCP server, where all the devices that have dynamically obtained IP addresses (from the DHCP server) have been recorded, and they reach out to the Juniper devices (other products won't respond to PyEz scripts) to check their on-box Junos image, verify if that is the needed version, do upgrades or downgrades as necessary, and finally push down a customized configuration file for that specific device.

Now with a decent understanding of JEAP, let's take a look at the code.

Here we use two scripts for better maintenance purposes, one called jeap.py, which is the main workflow of this project. The other is called tools.py, which serves as a tools library for jeap.py to call upon and use. Let's take a closer look at the PyEZ and the Juniper technology related codes, rather than just the normal Python codes here.

In jeap.py, it first imports the necessary libraries and the tools.py script, and initiates variables for later use:

```
import time
import logging
from lxml import etree
from tools import Tools
from jnpr.junos import exception
from jnpr.junos.utils.config import Config

if __name__ == "__main__":
   """
   Aim to do simple logic here
   And call modules to do specific function
   """
   tools = Tools()

   known_hosts = []       # hosts finished provisioning
   new_hosts = []        # hosts to-be provisioned
tmp_hosts = []        # hosts read from leases file
   # library for customer required to-be provisioned boxes
box_to_configure = tools.customer_requirements
```

Then, jeap.py reads from the dhcpd.leases file, filters out IP and its associated MAC address, and then returns them in pairs. If nothing shows up in the leases file, just wait 15 more seconds, then try it again:

```
while True:
   try:
     # tmp_hosts: list of dictionaries with ip and mac
     tmp_hosts = tools.lease_read("/var/lib/dhcp/dhcpd.leases")
  #return would be tmp_hosts=[{"ip":"10.0.1.20","mac":"aa:aa:aa:aa:aa:aa"}]
  if not tmp_hosts:
        print("don't have any DHCP client now, please wait.")
        time.sleep(15)
     else:
        break
   except IOError:
     print("Please restart your dhcp service, dhcpd.leases file have been removed somehow")
time.sleep(30)
```

You should have another datasheet at hand before configuring each box, a datasheet that specifies your *preferred* ID parameter of the device (like MAC address or serial number), the hostname, or any key parameters you want to change, say (`“ntp-server”: “xx.xx.xx.xx”`), the Junos version. Take the snippet below, for example:

```
{“Products”:[
   {“mac”:”aa:aa:aa:aa:aa:aa”, “model”:”EX2200”, “junos”:”12.8R12”, “hostname”: “EX2200-1”},
   {“mac”:”bb:bb:bb:bb:bb:bb”, “model”:”SRX320”, “junos”:”17.3”, “hostname”: “JEAP1”}
]}
```

Now, filter out the IP and MAC pairs that haven’t been provisioned yet. Based on MAC address or your *preferred* ID parameter, you can merge the datasheet information and ip-mac pairs together before saving them to list `new_hosts`:

```
for tester in tmp_hosts:
   # first filter by list of seen hosts in the past
   if tester not in known_hosts:
     # then filter by MAC from customer document
     for item in box_to_configure:
       if tools.mac_compare(tester[“mac”], item[“mac”]):
        # merge two info source together
        # thus don’t need to lookup IP every time
        item[“ip”] = tester[“ip”]
        new_hosts.append(item)
        break
```

New_hosts should be a list like this:

```
[{u’junos’: u’15.1X49-D60.7’, ‘ip’: ‘10.0.1.21’, u’mac’: bb:bb:bb:bb:bb:bb’, u’hostname’: u’JEAP1’,
u’model’: u’srx320-poe’}]
```

If new_hosts is not null, pop out each individual node in this list and reach out to each associated device one by one for further operation:

```
while new_hosts:
  # call another function to do that
  sample = new_hosts.pop(0)
  print(“\nTarget hosts:\n\t” + str(sample[“model”]) + “ @ “ + str(sample[“ip”]))
  # reach out to srx
  dev = tools.device_conn(sample[“ip”])
  cu = Config(dev)
  print(“Connecting ...”)
  try:
    dev.open()
    print(“Connected”)
  except exception.ConnectAuthError:
    print(“Username and password doesn’t match”)
    print(“Please double check your credentials and try again later.”)
    continue
  except exception.ConnectTimeoutError:
    print(“Connection timeout, will try again later”)
    continue
  except:
    print(“Cannot reach out to the device now, will try again later”)
continue

# check on-box Junos
# upgrade/downgrade if needed
# push down new config file
# …
```

The `device_conn` function is a few simple lines of PyEZ code, for reuse purposes:

```
def device_conn(self, host_ip):
   user = "demo"
password = "demo123"
Device.auto_probe = 10
return Device(host=host_ip, user=user, password=password, port="22", auto_probe=True)
```

Next you need to check the on-box Junos version, and compare to see if that's the one you want:

```
# pull info from box current settings
sample_on_box_model = dev.facts["model"]
sample_on_box_version = dev.facts["version"]
print("\nCurrent Config:")
print("\t" + sample_on_box_model + ":  " + sample_on_box_version)

# version check
feedback = tools.junos_version_compare(sample_on_box_version, sample["junos"])
print feedback
```

The `junos_version_compare` function compares the on-box Junos version to the desired Junos version specified in the customer datasheet, digit by digit, and returns 0 if they are same, "-1" if on-box version is older, and "1" if it's newer:

```
def junos_version_compare(self, sample_version, target_version):
  sample_series = self.junos_version_serial_analyze(sample_version)
  target_series = self.junos_version_serial_analyze(target_version)
  count = min(len(sample_series), len(target_series))
  for x in range(count):
    if sample_series[x] != target_series[x]:
      if sample_series[x] > target_series[x]:
        # print("current one has newer version @" + str(x+1) + " digit")
        return 1
      elif sample_series[x] < target_series[x]:
        # print("current one has older version")
        return -1
      # if compare every digit and they are the same till the end
      # it indicates that they are the same
      # just of different length
      return 0

def junos_version_serial_analyze(self, sample):
  holder = 0
  result = []
  counter = 0
  for x in sample:
  counter += 1
  if x.isdigit():
    holder = holder * 10 + int(x)
  if counter == len(sample):
    result.append(holder)
  else:
    if holder != 0:
      result.append(holder)
      holder = 0
    if x.isalpha():
      result.append(str(x))
    else:
      continue
  return result
```

Based on the return value of function junos_version_compare, you can decide if an Junos image update is needed.

```
if feedback >= 0:
  print("On-box Junos version at " + sample["ip"] + "(" + sample_on_box_model + ")" + "is NOT OLDER than 
customer required")
  print("No need to upgrade\n")
else:
  print("On-box Junos version at " + sample["ip"] + "(" + sample_on_box_model + ")" + "is OLDER than 
customer required")
  print("Need to upgrade\n")
  # first check if we store that junos file locally
  filename = tools.local_junos_dir_check(sample["model"], sample["junos"])
  if filename:
    # after find needed image file locally, call PyEz to install that
dev = tools.junos_auto_install(str(sample["ip"]), "Junos/" + filename, dev)
  else:
# if we don't have file locally, print out warning
print("Failed to find customer required image locally, please download in Junos folder before we can 
perform system upgrade")

def junos_auto_install(self,host_ip, path, device):
"""
Call PyEz to secure install new junos version
to remote srx host
"""

sw = SW(device)
path = os.path.join(os.getcwd(),path)
#print path,type(path)
try:
  ok = sw.install(package=path, progress=install_progress)
except Exception as err:
  print("Install error")
  return False

print(ok)
try:
  rsp = sw.reboot()
  print(rsp)
except exception.ConnectClosedError:
  print("About to loose connection ..")
finally:
  print("Please wait for the box to wake-up!")
  time.sleep(20)
  dev = self.device_conn(host_ip)
  print("Initiating connection to box ...")
  while True:
    try:
      dev.open()
      break
    except:
      print “Box is not ready yet, try again in next 20s...”
      time.sleep(20)
  print(“Connected”)
  print(“New version:”)
  print(“\t” + dev.facts[“model”] + “:  “ + dev.facts[“version”])
  return dev

def install_progress(dev, msg):
  print(“{}:{}”.format(dev.hostname, msg))
```

Last but not least, push down a customized configuration file for that specific device. And append this device to the known_hosts list, so we don’t re-configure it next time. Don’t forget to read the dhcpd.leases file again at the end to keep the loop running.

```
cu = Config(dev)
print(“\n\nPushing down another configuration file now “)
tools.config_composer(sample[“model”], sample[“hostname”], sample_on_box_version)
cu.load(path=”Config_History/” + sample[“hostname”] + “.set”)
try:
  cu.commit(timeout=180)
except exception.RpcTimeoutError:
  print(“Need longer time to commit, please adjust commit timeout value ...”)
  time.sleep(5)
print(“Configuration Updated”)
tmp = {}
tmp[“ip”] = sample[“ip”]
tmp [“mac”]= tools.mac_return(sample[“model”], str(sample[“mac”]))
known_hosts.append(tmp)
dev.close()

print(“\nRead leases file again:”)
tmp_hosts = tools.lease_read(“/var/lib/dhcp/dhcpd.leases”)
time.sleep(10)
```

The `config_composer` function is based on Jinja2 templates and yaml files, that you probably already learned them from other recipes in this book. Specifically, you need the device’s model information here because there are at least two different Jinja2 templates dedicated to the SRX and the EX due to the difference in their configuration. Configuration files are named after the hostname specified in the customer datasheet and saved locally. Configuration files can be text-based, xml-based, or set-command-based. In this demo, the set-command-based configuration file is used:

```
def config_composer(self, model, hostname, junos_on_box_version):
  “””
  Using Yaml and Jinja2 generate dynamic templates
  “””
  if “SRX” in model:
    template_filename = “SRX_template.j2”
    network_parameter_filename = “SRX_networkParameters.yaml”
  elif “EX” in model:
    template_filename = “EX_template.j2”
    network_parameter_filename = “EX_networkParameters.yaml”
  complete_path = os.path.join(os.getcwd(), ‘Config’)
  ENV = jinja2.Environment(loader=jinja2.FileSystemLoader(complete_path))
  template = ENV.get_template(template_filename)
  with open(complete_path + “/” + network_parameter_filename) as yamlfile:
    dict = yaml.load(yamlfile)  # yaml file is loaded as a dictionary with key value pairs
  addition = {“hostname”: hostname, “version”: junos_on_box_version}
  dict.update(addition)
  content = template.render(dict)
  target = open(“path_to_save_config_file/” + hostname + “.set”, ‘w’)
  target.write(content)
  target.close()
  return content
```

Here is the screen output when you don’t need to update the Junos image:

```
demo@ubuntu:~/Documents/JEAP$ python jeap.py
Below are new hosts from lease file:
[{u’junos’: u’15.1X49-D60.7’, ‘ip’: ‘10.0.1.21’, u’mac’: u’bb:bb:bb:bb:bb:bb’, u’hostname’: u’JEAP1’,
u’model’: u’srx320-poe’}]

Target hosts:
          srx320-poe @ 10.0.1.21
Connecting ...
Connected

Current Config:
          SRX320-POE:  15.1X49-D60.7
[15, 1, ‘X’, 49, ‘D’, 60, 7]
[15, 1, ‘X’, 49, ‘D’, 60, 7]
0
On-box Junos version at 10.0.1.21(SRX320-POE)is NOT OLDER than customer required
No need to upgrade

Pushing down another configuration file now
Configuration Updated
Remaining hosts to configure this round:
[]
```

On the SRX CLI, it would look like this (hostname change from JEAP to JEAP1):

```
[edit]
root@JEAP#

[edit]
root@JEAP1#
```

And the screen output when the Junos image update *is* needed:

```
Below are new hosts from lease file:
[{u’junos’: u’17.3’, ‘ip’: ‘10.0.1.21’, u’mac’: u’bb:bb:bb:bb:bb:bb’, u’hostname’: u’JEAP2’, u’model’:
u’SRX320-poe’}]

Target hosts:
          SRX320-poe @ 10.0.1.21
Connecting ...
Connected

Current Config:
```

```
          SRX320-POE:  15.1X49-D60.7
[15, 1, ‘X’, 49, ‘D’, 60, 7]
[17, 3]
-1
On-box Junos version at 10.0.1.21(SRX320-POE)is OLDER than customer required
Need to upgrade

Local junos directory check for version 17.3 ...
Found proper image ...
10.0.1.21:computing checksum on local package:
10.0.1.21:cleaning filesystem ...
10.0.1.21:before copy, computing checksum on remote package: /var/tmp/junos-srxsme-17.3R1.10.tgz
10.0.1.21:junos-srxsme-17.3R1.10.tgz: 29589504 / 295871546 (10%)
10.0.1.21:junos-srxsme-17.3R1.10.tgz: 59179008 / 295871546 (20%)
10.0.1.21:junos-srxsme-17.3R1.10.tgz: 88768512 / 295871546 (30%)
10.0.1.21:junos-srxsme-17.3R1.10.tgz: 118358016 / 295871546 (40%)
10.0.1.21:junos-srxsme-17.3R1.10.tgz: 147947520 / 295871546 (50%)
10.0.1.21:junos-srxsme-17.3R1.10.tgz: 177537024 / 295871546 (60%)
10.0.1.21:junos-srxsme-17.3R1.10.tgz: 207110144 / 295871546 (70%)
10.0.1.21:junos-srxsme-17.3R1.10.tgz: 236699648 / 295871546 (80%)
10.0.1.21:junos-srxsme-17.3R1.10.tgz: 266289152 / 295871546 (90%)
10.0.1.21:junos-srxsme-17.3R1.10.tgz: 295871546 / 295871546 (100%)
10.0.1.21:after copy, computing checksum on remote package: /var/tmp/junos-srxsme-17.3R1.10.tgz
10.0.1.21:checksum check passed.
10.0.1.21:installing software ... please be patient ...
10.0.1.21:software pkgadd package-result: 0
Output:
Formatting alternate root (/dev/da0s2a)...
/dev/da0s2a: 2518.0MB (5156848 sectors) block size 16384, fragment size 2048
          using 14 cylinder groups of 183.62MB, 11752 blks, 23552 inodes.
super-block backups (for fsck -b #) at:
 32, 376096, 752160, 1128224, 1504288, 1880352, 2256416, 2632480, 3008544,
 3384608, 3760672, 4136736, 4512800, 4888864
saving package file in /var/sw/pkg ...
Installing package ‘/altroot/cf/packages/install-tmp/junos-17.3R1.10’ ...
Verified junos-boot-srxsme-17.3R1.10.tgz signed by PackageProductionEc_2017 method ECDSA256+SHA256
Verified junos-srxsme-17.3R1.10-domestic signed by PackageProductionEc_2017 method ECDSA256+SHA256
JUNOS 17.3R1.10 will become active at next reboot
WARNING: A reboot is required to load this software correctly
WARNING:    Use the ‘request system reboot’ command
WARNING:        when software installation is complete
cp: cannot overwrite directory /altroot/cf/etc/ssh with non-directory /cf/etc/ssh
Saving state for rollback ...

Software installation succeeded
Shutdown NOW!
[pid 2634]
Please wait for the box to wake-up!

Connecting to box now ...
Connected
New version:
SRX320-POE:   17.3R1.10

Pushing down another configuration file now
Configuration Updated
Remaining hosts to configure this round:
[]
```

```
Read leases file again:
Don’t have any new host come online
```

The CLI on the SRX would look like this (Junos version upgrading from 15.1 to 17.3, and hostname changing from JEAP1 to JEAP2):

```
[edit]
root@JEAP1#

*** FINAL System shutdown message from root@JEAP1 ***

System going down IMMEDIATELY


SWaiting (max 60 seconds) for system process `vnlru’ to stop...done
Waiting (max 60 seconds) for system process `vnlru_mem’ to stop...Ignoring watchdog timeout during 
boot/reboot
done
Waiting (max 60 seconds) for system process `bufdaemon’ to stop...done
Waiting (max 60 seconds) for system process `syncer’ to stop...
Syncing disks, vnodes remaining...3 3 0 0 0 0 done
                …..

Fri Sep  1 00:10:29 UTC 2017

JEAP1 (ttyu0)

login: root
Password:

--- JUNOS 17.3R1.10 built 2017-08-23 06:40:27 UTC
root@JEAP1%
root@JEAP1%
root@JEAP1% cli
root@JEAP2>
```

Discussion

Once you master ZTP on the SRX using PyEZ, try applying this to the EX, or even expanding it to the MX Series and QFX Series, or try using the serial number as the ID instead of the MAC address.

Also, you can have a lot of fun with Jinja2 and YAML, and there are way more interesting things you can do than just changing the hostname!

Recipe 17 - PyEZ for On-Box Scripts

by Peter Klimai

- Python Version Used: 2.7
- PyEZ Version Used: 1.3.1 (on-box PyEZ version for Junos 17.1R2)
- Junos OS Used: 17.1R2
- Juniper Platforms General Applicability: MX, EX, ACX, QFX, PTX, T

For many years, operators were able to create Junos OS commit, op, and event automation scripts using SLAX and XSLT programming languages. Starting with Junos OS Release 16.1, you can use Python for the same task. This recipe presents an example of an operational (op) script that uses Python and PyEZ and runs on-box – without the need for an external server. The script solves the classic task of bouncing the interface specified as its parameter.

Problem

As an old proverb says, "seven problems – one reset button." For the network engineer, it is no surprise that troubleshooting the network often requires restarting protocols, processes, different hardware components, bouncing the interfaces, and more. Although these actions might not be required in a perfect world, the reality is that we're required to do such things every so often. With respect to interfaces, a standard method of bouncing them in Junos is disabling, commiting, enabling back, and then commiting again. This sounds simple, and it is – if you only have to do it once or twice. But what if you have to perform this, or a similar task, over and over again? The answer is clear: *let's create an automation script!*

NOTE Other recipes in this book assume Python scripts running remotely on a dedicated management server. In this recipe, you want the operator to be able to easily bounce the interface when working with the device CLI, hence an on-box op script is required.

Solution

Python on a Junos OS device has many modules available including `jinja2`, `lxml`, `and paramiko`, and it also includes Junos PyEZ (`jnpr.junos`). You can consult Juniper technical documentation for a complete list of supported modules. The solution script in this recipe will use a standard `Device` class from `jnpr.junos` module for connection and configuration of the local device, in a way that is very similar to how you would do it with an off-box script (that is, one running remotely on a management server).

NOTE On-box Python currently requires you to use Python 2.7 and also an earlier version of PyEZ. Check for recent versions of Python and PyEZ that might be supported in future Junos releases.

To create this recipe, you first create the automation script `interface_bounce.py`. Its source code is presented below (the script's parts are numbered `# (n)` for the explanation that follows):

```
from jnpr.junos import Device                # (1)
from jnpr.junos.utils.config import Config
from jnpr.junos.exception import *
import argparse
from time import sleep

arguments = {                              # (2)
   “interface”: “Name of the interface to disable/enable”,
   “delay”: “Time to wait before enabling the interface (seconds)”,
}

def config_xml(interface_name, disable_attributes):     # (3)
   return “””
      <configuration>
         <interfaces>
            <interface>
               <name>{0}</name>
               <disable {1}/>
            </interface>
         </interfaces>
      </configuration>
   “””.format(interface_name, disable_attributes)

def change_config(dev_cfg, delta_config, log_message):      # (4)
   print “%s: Locking the configuration” % log_message
   try:
      dev_cfg.lock()
```

```
    except LockError:
        print “Error: Unable to lock configuration”
        return False

    print “%s: Loading configuration changes” % log_message
    try:
        dev_cfg.load(delta_config, format=”xml”, merge=True)
    except ConfigLoadError as err:
        print “Unable to load configuration changes: \n” + err
        print “Unlocking the configuration”
        try:
            dev_cfg.unlock()
        except UnlockError:
            print “Error: Unable to unlock configuration”
        return False

    print “%s: Committing the configuration” % log_message
    try:
        dev_cfg.commit()
    except CommitError:
        print “Error: Unable to commit configuration”
        print “Unlocking the configuration”
        try:
            dev_cfg.unlock()
        except UnlockError:
            print “Error: Unable to unlock configuration”
        return False

    print “%s: Unlocking the configuration” % log_message
    try:
        dev_cfg.unlock()
    except UnlockError:
        print “Error: Unable to unlock configuration”
        return False

    return True

def main():                                   # (5)
    parser = argparse.ArgumentParser()         # (6)
    for key in arguments:
        parser.add_argument((‘-’ + key), required=True, help=arguments[key])
    args = parser.parse_args()

    with Device() as dev:                     # (7)
        dev.bind( cu=Config )                 # (8)
        if change_config(dev.cu, config_xml(args.interface, “”),    # (9)
                “Disabling interface”):
            print “Waiting %s seconds...” % args.delay
            sleep(float(args.delay))                        # (10)
            if change_config(dev.cu, config_xml(args.interface, “delete=’delete’”),
                    “Enabling interface”):                  # (11)
                print “Interface bounce script finished successfully.”
            else:
                print “Error enabling the interface, it will remain disabled.”

if __name__ == “__main__”:                                  # (12)
    main()
```

Let's see what is happening in the script:

1. Import standard PyEZ classes `Device`, `Config`, and PyEZ exceptions. Also import `argparse` module and `sleep()` function that will be used in the script.

2. Define the `arguments` dictionary, which is a reserved name that tells the Junos OS which CLI parameters the script will work with. Using this variable allows using context sensitive help (question mark) when working with scripts. The present script uses `interface` and `delay` arguments (also known as *parameters*).

3. The function `config_xml()` returns part of the Junos configuration that is responsible for disabling (or enabling) the specific interface in XML form. The interface name must be passed as a parameter `interface_name`, and the second parameter `disable_attributes` defines the possible attributes for the `<disable>` XML element.

4. The `change_config()` function locks, commits, and unlocks the specified `delta_config` configuration on the particular device (identified by corresponding configuration object `dev_cfg`). It also prints the status and tries to catch possible exceptions, including `LockError`, `ConfigLoadError`, and some others. The function returns a Boolean value, with `True` meaning successful configuration load.

5. The main function.

6. Use an instance of `ArgumentParser()` class to get arguments provided to the script, and put them in the `args` object.

7. Create an instance of `Device()` class called `dev`. The context manager syntax (`with` operator) is used to make sure `open()` and `close()` methods are called automatically as needed (and so you do not need to issue them explicitly).

8. Bind the configuration object to the device instance. This allows the performance of configuration tasks.

9. The call `change_config()` function was defined previously, disabling the interface. Proceed only if it returns `True`.

10. Sleep the specified number of seconds.

11. Call the `change_config()` function again, this time providing `"delete='delete'"` as a `disable_attributes` parameter to `config_xml()` function. The intention is to enable the interface, deleting the `disable` option from interface configuration. Also, print the message depending on the function return value.

12. The standard Python script "entry point."

NEXT

Now that you developed the script, in order to run it on the Junos box you have to:

- Copy the `interface_bounce.py` script to the `/var/db/scripts/op/` directory on the Junos device. Use any convenient tool and protocol for that, such as SCP. Make sure that the script owner is a user of a super-user class and that this user has write permissions for the script file. Make sure other users only have read permissions for the same file.
- Enter configuration mode and configure Python as a scripting language for Junos:

```
lab@vMX-1> configure
Entering configuration mode
[edit]
lab@vMX-1# set system scripts language python
```

- Configure `interface_bounce.py` script as an op script and commit:

```
[edit]
lab@vMX-1# set system scripts op file interface_bounce.py
[edit]
lab@vMX-1# commit and-quit
commit complete
Exiting configuration mode

lab@vMX-1>
```

- Finally, you can run the script from the Junos CLI using the `op` command, providing, in this example, the interface name of ge-0/0/0 and value of 10 seconds as a delay:

```
lab@vMX-1> op interface_bounce.py interface ge-0/0/0 delay 10
Disabling interface: Locking the configuration
Disabling interface: Loading configuration changes
Disabling interface: Committing the configuration
Disabling interface: Unlocking the configuration
Waiting 10 seconds...
Enabling interface: Locking the configuration
Enabling interface: Loading configuration changes
Enabling interface: Committing the configuration
Enabling interface: Unlocking the configuration
Interface bounce script finished successfully.
```

Discussion

Among the multiple ways of checking to see that the script actually worked, you can view the configuration and interface operational state before, during, and after script execution. Here, you just check the log messages to make sure the interface ge-0/0/0 was down for nearly 10 seconds:

```
lab@vMX-1> show log messages | match SNMP_TRAP_LINK_ | last
Jul 22 21:49:20  vMX-1 mib2d[4219]: SNMP_TRAP_LINK_DOWN: ifIndex 518, ifAdminStatus down(2),
ifOperStatus down(2), ifName ge-0/0/0
Jul 22 21:49:31  vMX-1 mib2d[4219]: SNMP_TRAP_LINK_UP: ifIndex 518, ifAdminStatus up(1), ifOperStatus
up(1), ifName ge-0/0/0
[ ... ]
```

The messages log confirms that the script worked as desired.

As you can see, Junos on-box scripts written in Python are very similar to off-box PyEZ automation scripts (see other recipes in this book!). Junos OS not only allows for multiple types of automation, it comes with consistency: on-box and off-box automation can use similar approaches, the same language, and the same library modules!

Recipe 18 - Automated Network Testing with Junos PyEZ

by Peter Klimai

- Python Version Used: 3.5
- PyEZ Version Used: 2.1.4
- Junos OS Used: 12.1X47-D40
- Juniper Platforms General Applicability: All

You can make sure your network is working well by spotting problems before users notice them. The good news is that automation can help with this task. This recipe provides an example of how you can automate network tests with Junos PyEZ.

Problem

Typically, to make sure your network is operating normally, you log in to the devices and check multiple command outputs making sure everything is “Full” and “Established.” Alternatively, you might look at graphs that represent different counters and values, as provided by your monitoring system. Today you can use a different approach: create a script that does the job for you.

NOTE This recipe’s approach is similar to a common software engineer’s practice of writing *unit tests* – small pieces of code that test different parts of the main program. Here, we’re testing the network instead!

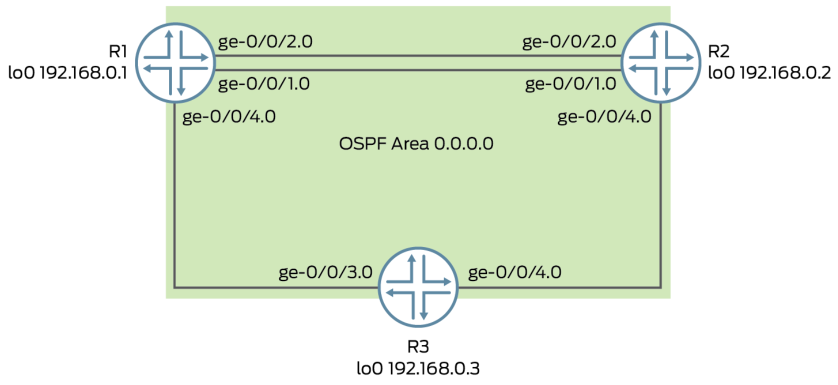


Figure 18.1 Network Topology Used in Recipe 18

The exact problem is making sure the OSPF network shown in Figure 18.1 is working fine. In particular, you need to make sure all adjacencies are in the Full state. You should also send a short email with test results to the administrator and schedule the script execution on your Linux server.

Solution

To solve the specific task of OSPF adjacency testing automation, you must first create the automation script, here named `monitor_ospf.py`. Its source code is presented below, and the script's parts are numbered with numerals `# (n)` for the explanation that follows the script:

```
from jnpr.junos import Device                              # (1)
from jnpr.junos.op.ospf import OspfNeighborTable
import smtplib

MAIL_LOGIN = “user@example.com”                             # (2)
MAIL_PW = “xxx”
TO_ADDR = “admin@example.com”
MAIL_SERVER = “smtp.example.com”
SMTP_SSL_PORT = 587
SUBJ = “OSPF adjacency test results”

USER = “lab”                                             # (3)
PASSWD = “lab123”
R1 = “10.254.0.35”
R2 = “10.254.0.37”
R3 = “10.254.0.38”

def check_ospf_full_adjacencies(dev, neighbor_count):     # (4)
   ospf_table = OspfNeighborTable(dev)    # Create an instance of the Table
   ospf_table.get()                     # Populate the Table
   if len(ospf_table) != neighbor_count:
      return False
```

```
    for neighbor in ospf_table:
        if neighbor["ospf_neighbor_state"] != "Full":
            return False
    return True

def str_result(test_result):                          # (5)
    return "Success" if test_result else "Fail"

def main():                                         # (6)
    with Device(host=R1, user=USER, password=PASSWD) as dev:
        result1 = check_ospf_full_adjacencies(dev, 3)
        print("Test OSPF adjacencies on R1: " + str_result(result1))

    with Device(host=R2, user=USER, password=PASSWD) as dev:
        result2 = check_ospf_full_adjacencies(dev, 3)
        print("Test OSPF adjacencies on R2: " + str_result(result2))

    with Device(host=R3, user=USER, password=PASSWD) as dev:
        result3 = check_ospf_full_adjacencies(dev, 2)
        print("Test OSPF adjacencies on R3: " + str_result(result3))

    print("Sending email.")                         # (7)

    body_msg = "Test results: %s, %s, %s\n" % (str_result(result1),
                                   str_result(result2),
                                   str_result(result3))
    msg = "From: %s\nTo: %s\nSubject: %s\n\n%s\n" % (MAIL_LOGIN,
                                        TO_ADDR, SUBJ, body_msg)
    mailserver = smtplib.SMTP(MAIL_SERVER, SMTP_SSL_PORT)
    mailserver.starttls()
    mailserver.login(MAIL_LOGIN, MAIL_PW)
    mailserver.sendmail(MAIL_LOGIN, TO_ADDR, msg)
    mailserver.quit()

if __name__ == "__main__":                          # (8)
    main()
```

Here is what's happening in the script:

1. Import PyEZ `Device` and `OspfNeighborTable` classes, and the `smtplib` module from the standard Python library.

2. Define a set of "constants" that will be used for sending email from the script: login, password, email-to address, server address, server port, and email subject.

3. Login and password for device connection, and IP addresses of R1, R2, and R3 routers.

4. The function `check_ospf_full_adjacencies()` takes two parameters: device instance `dev` and expected OSPF neighbor count `neighbor_count`. It then uses the built-in PyEZ functionality of operational tables; the `ospf_table` variable, using the `OspfNeighborTable` class, is populated with the device's OSPF adjacency data. You can then check the length of the table (it should be equal to `neighbor_count`) and the adjacency state (each should be Full). The function returns `True` only if all OSPF adjacencies are fine.

5. The `str_result()` function performs an auxiliary task of converting the Boolean result to a string representation.

6. In the `main()` function, for each device, open a connection and call the `check_ospf_full_adjacencies()` function. As you can see from Figure 1, you can expect three Full OSPF adjacencies for R1 and R2, and two adjacencies for R3, and these values are provided to the function using parameters. Also, the success or failure result of the test is printed to the screen.

7. Here you form an email and send it using smtplib. The syntax is mostly self-explanatory. As with other parts of the script, exception checking is left out, after the fact, to reduce the length of the example script. This means that if exception happens (such as a mail server being unreachable, etc.), the script will be terminated and you will get a standard error stack trace in the terminal.

8. Standard Python script entry point.

NEXT

Now run the script manually to see if it works properly. You should get the following results in the terminal, which tells you that OSPF is currently working fine:

```
lab@ubuntu:~$ python monitor_ospf.py
Test OSPF adjacencies on R1: Success
Test OSPF adjacencies on R2: Success
Test OSPF adjacencies on R3: Success
Sending email.
```

Additionally, after the script runs, you'll receive an email with the following content (both here and in the script above, the lab device names and addresses were changed):

```
From: user@example.com
To: admin@example.com
Subject: OSPF adjacency test results

Test results: Success, Success, Success
```

Very good to receive such notices! The final step will be to schedule the periodic script execution using cron. Here, you do it on your Ubuntu Linux server just by editing the `/etc/crontab` file and adding this line to it to get your emails at the start of every hour:

```
00 *      * * *  root /usr/bin/python3.5 /home/lab/monitor_ospf.py
```

Discussion

In the same way that this script worked, you can write many other tests to make sure functionality of your network did not degrade. You've seen how Junos PyEZ Tables and Views are used for this purpose, but there are other possible approaches to creating tests, and here, briefly, let's show you two alternative ways to write the same `check_ospf_full_adjacencies()` function.

The first alternative is to use the direct processing of XML RPC responses using methods defined by `lxml` library, such as `find()`, `findall()`, or `xpath()`:

```
def check_ospf_full_adjacencies(dev, neighbor_count):
   full_count = 0
   rpc_result = dev.rpc.get_ospf_neighbor_information()
   for element in rpc_result.findall("ospf-neighbor"):
      if element.findtext("ospf-neighbor-state") == "Full":
         full_count += 1
      else:
         return False
   return full_count == neighbor_count
```

This approach of working with XML is extremely flexible, but might not be so simple to use. The second possible alternative is to use a Python package called `jxmlease` (https://github.com/Juniper/jxmlease). Its methods allow you to to convert XML to native Python data structures (lists, dictionaries, and their combinations) for easier access to XML elements, as demonstrated below:

```
import jxmlease
def check_ospf_full_adjacencies(dev, neighbor_count):
   full_count = 0
   parser = jxmlease.EtreeParser()
   res = parser(dev.rpc.get_ospf_neighbor_information())
   for neighbor_data in res["ospf-neighbor-information"]["ospf-neighbor"]:
      if neighbor_data["ospf-neighbor-state"] == "Full":
         full_count += 1
      else:
         return False
   return full_count == neighbor_count
```

These two different definitions of `check_ospf_full_adjacencies()` work exactly the same as our first definition in the main script – so for your tests, you can choose the approach you like best, or have each script do a different thing and email.

MORE? For more test examples and some additional discussion on the subject please refer to *Day One: Juniper Ambassadors' Cookbook 2017*, available here: https://www.juniper.net/us/en/training/jnbooks/day-one/networking-technologies-series/cookbook-2017/, and the corresponding GitHub repository: https://github.com/pklimai/pyez-network-testing.

MORE? For an example of a more advanced and high-level testing framework that does not actually require you to write scripts, check out JSNAPy tool available at GitHub under this URL: https://github.com/Juniper/jsnapy. An excellent *Day One* book covers JSNAPy here: http://www.juniper.net/us/en/training/jnbooks/day-one/automation-series/jsnapy/.

Recipe 19 - Menu Script for Address Book Editing

by Peter Klimai

- Python Version Used: 3.5
- PyEZ Version Used: 2.1.4
- Junos OS Used: 15.1X49-D70
- Juniper Platforms General Applicability: SRX Series

There are many simple tasks that administrators have to perform often, and automating them can free up a lot of time. Editing address black- or whitelists, for example, is a manipulation that can easily be required multiple times a day.

Additionally, sometimes personnel are not keen on working with CLI or Web UI directly, but must still perform network device configuration changes. In this case, a simple multiple-choice style menu script can do the job, with minimal education and stress required.

This recipe gives you an example of how you can create a menu script to edit the address book for the Juniper SRX Series.

Problem

Let's assume you have an SRX Series that makes use of a set of security policies, and some of the policies reference an address-set named *ALLOWED-IN*. This address-set contains IPv4 addresses and the IPv4 subnets must be changed frequently. Also, to aid automation, let's use a standard name convention for the address object, deriving it from the IP address or subnet (if you want a more unique name you can modify the solution script accordingly, as an exercise).

The particular commands that you have to manually enter to add a record to the address-set are similar to these (assuming that you use a global address book):

```
lab@SRX# set security address-book global address CIDR-10.1.1.0/24 10.1.1.0/24
lab@SRX# set security address-book global address-set ALLOWED-IN address CIDR-10.1.1.0/24
```

Here, in the first line, you created an address object named *CIDR-10.1.1.0/24,* which is actually just a 10.1.1.0/24 subnet, and in the second line you put it in the `address-set`. This is not too complex, but what actually changes for every new address or subnet that you add? Correct, only the IP and subnet mask. You will have to type everything else again and again – so let's automate the process to avoid that.

Solution

You can solve the task using `address_edit.py` script presented below. The script's parts are numbered as `# (n)` for the explanation that follows:

```
from jnpr.junos import Device                            # (1)
from jnpr.junos.utils.config import Config
from jnpr.junos.exception import *
from lxml import etree
import ipaddress

USER = “lab”                                          # (2)
PASSWD = “lab123”
DEVICE_IP = “10.254.0.35”

ADDR_BOOK_NAME = “global”                                # (3)
ADDR_SET_NAME = “ALLOWED-IN”
ADDR_NAME_PREFIX = “CIDR-”

                                                    # (4)
STR_INVITE = “””
Address-set editor script for Juniper SRX. Device: {0}
Address book to edit: {1}   Address-set to edit: {2}
  r - READ and show addresses
  a - ADD address (IP or subnet)
  d - DELETE particular address
  q - QUIT script
Choice >>> “””

STR_QUITING = “Goodbye!”
STR_READING = “\nReading and displaying address book entries:”
STR_UNKNOWN_INPUT = “Unknown input, please repeat.”
STR_INCONSISTENT_ERROR = “Error: Address book on the device \
requires manual fix before this script can be used.”
STR_ENTER_IP_ADD = “Enter IP or IP/mask to add >>> “
STR_ADDRESS_ADDED = “Address added successfully.”
STR_INVALID_IP = “Invalid entry, please repeat.”
STR_ENTER_IP_DEL = “Enter IP/mask to delete >>> “
STR_ADDRESS_DELETED = “Address deleted successfully.”
STR_PDIFF_BANNER = “\nConfig diff on the device:”
STR_CONFIG_CHANGED = “Configuration change committed.”
```

```
                                                  # (5)
STR_GET_CONFIG = “””<configuration>
                   <security>
                      <address-book>
                         <name>{0}</name>
                      </address-book>
                   </security>
                </configuration>”””

STR_SET_CONFIG = “””set security address-book {0} address {1}{2} {2}
set security address-book global address-set ALLOWED-IN address {1}{2}”””

STR_DELETE_CONFIG = “””delete security address-book {0} address {1}{2} {2}
delete security address-book global address-set ALLOWED-IN address {1}{2}”””

class InconsistentConfigException(Exception):           # (6)
   pass


def read_addresses():                              # (7)
   addr_book_prefixes = set()
   address_set_prefixes = set()   # empty sets so far
   try:                                                     # (8)
      with Device(host=DEVICE_IP, user=USER, password=PASSWD) as dev:
         resp = dev.rpc.get_config(
               filter_xml=etree.XML(STR_GET_CONFIG.format(ADDR_BOOK_NAME)),
               options={‘inherit’: ‘inherit’,
                      ‘database’: ‘committed’,
                      ‘format’: ‘XML’})
         if resp is not None:                               # (9)
            for address_element in resp.findall(“security/address-book/address”):
               name = address_element.findtext(“name”)
               ip_prefix = address_element.findtext(“ip-prefix”)
               if name is not None and ip_prefix is not None:
                  if name == ADDR_NAME_PREFIX + ip_prefix:
                     if sanitize_ip(ip_prefix) is not None:
                        addr_book_prefixes.add(ip_prefix)
                  else:
                     pass # this entry was not added by script - ignore

            for address_set_element in resp.findall(              # (10)
                  “security/address-book/address-set[name=’{0}’]/address”
                        .format(ADDR_SET_NAME)):
               ab_name = address_set_element.findtext(“name”)
               if ab_name.startswith(ADDR_NAME_PREFIX):
                  test_ip = ab_name[len(ADDR_NAME_PREFIX):]
                  if test_ip in addr_book_prefixes:
                     address_set_prefixes.add(test_ip)
                  else:
                     # All addresses in address set that start with prefix
                     # ADDR_NAME_PREFIX, must be of ‘standard’
                     # form ADDR_NAME_PREFIX + IP/mask
                     raise InconsistentConfigException(
                        “Inconsistent entry in address book”)

   except ConnectRefusedError:                        # (11)
      print(“\n\nError: Connection refused!”)
   except ConnectTimeoutError:
        print(“\n\nError: Device connection timed out!”)
    except ConnectAuthError:
        print(“\n\nError: Authentication failure!”)

    return address_set_prefixes


def display_addresses(addrs):                         # (12)
    for addr in sorted(addrs):
        print(addr)


def sanitize_ip(address_entered):                      # (13)
    result = address_entered
    if “/” not in result: result += “/32”
    try:
        ip = ipaddress.IPv4Network(result)
    except:
        return None
    return result


def change_config_with_set_commands(set_commands):       # (14)
    try:
        with Device(host=DEVICE_IP, user=USER, password=PASSWD) as dev:
            # open and close is done automatically by context manager
            with Config(dev, mode=”exclusive”) as conf:
                # exclusive locks are treated automatically by context manager
                conf.load(set_commands, format=”set”)
                print(STR_PDIFF_BANNER)
                conf.pdiff()
                conf.commit()
    except LockError:
        print(“\n\nError applying config: configuration was locked!”)
    except ConnectRefusedError:
        print(“\n\nError: Device connection refused!”)
    except ConnectTimeoutError:
        print(“\n\nError: Device connection timed out!”)
    except ConnectAuthError:
        print(“\n\nError: Authentication failure!”)
    except ConfigLoadError as ex:
        print(“\n\nError: “ + str(ex))
    else:
        print(STR_CONFIG_CHANGED)


def add_address(address_sanitized):                    # (15)
    change_config_with_set_commands(STR_SET_CONFIG.format(
                ADDR_BOOK_NAME, ADDR_NAME_PREFIX, address_sanitized))


def del_address(address_sanitized):                    # (16)
    change_config_with_set_commands(STR_DELETE_CONFIG.format(
        ADDR_BOOK_NAME, ADDR_NAME_PREFIX, address_sanitized))


def main():                                           # (17)
```

```
    while True:
        print(STR_INVITE.format(DEVICE_IP, ADDR_BOOK_NAME, ADDR_SET_NAME), end="")
        choice = input().lower()
        if choice == "q":                              # (18)
            print(STR_QUITING)
            break
        elif choice == "r":                            # (19)
            print(STR_READING)
            try:
                addrs = read_addresses()
            except InconsistentConfigException:
                print(STR_INCONSISTENT_ERROR)
                break
            display_addresses(addrs)
        elif choice == "a":                            # (20)
            print(STR_ENTER_IP_ADD, end="")
            address_entered = input()
            address_sanitized = sanitize_ip(address_entered)
            if address_sanitized is None:
                print(STR_INVALID_IP)
            else:
                add_address(address_sanitized)
        elif choice == "d":                            # (21)
            print(STR_ENTER_IP_DEL, end="")
            address_entered = input()
            address_sanitized = sanitize_ip(address_entered)
            if address_sanitized is None:
                print(STR_INVALID_IP)
            else:
                del_address(address_sanitized)
        else:
            print(STR_UNKNOWN_INPUT)

if __name__ == "__main__":                             # (22)
    main()
```

Here is the explanation of what is happening in the script:

1. Import PyEZ `Device` and `Config` classes, all PyEZ exception definitions, the `etree` module from `lxml` and the `ipaddress` module from the standard Python library.

2. Login, password, and IP addresses for device connection – note that best practice is actually to not hardcode the login credentials in the script.

3. Set of "constants" for address book name, address-set name, and address name prefix.

4. STR_INVITE is the string that will be used as a main menu prompt. Note that {0}, {1}, and {2} will be substituted with particular values using the `format()` method. Some other string "constants" follow.

5. STR_GET_CONFIG is a small XML document that will be used below in a get configuration operation. STR_SET_CONFIG and STR_DELETE_CONFIG will be used for changing the configuration.

6. Here, you create a custom exception type, `InconsistentConfigException`, for use below.

7. The function `read_addresses()` is used to read the address book from a Junos device. You start by creating two empty sets (in a mathematical sense): addr_book_prefixes and address_set_prefixes. Then addr_book_prefixes will contain IP prefixes from the address book, for which the address name starts with ADDR_NAME_PREFIX. The address_set_prefixes, on the other hand, contain addresses from the address-set ADDR_SET_NAME, that conforms to this convention. They must also be present in the address book – otherwise a custom exception is generated.

8. Use the `try` syntax to catch possible exceptions in the nested block. Use context manager (`with` operator) to create a `Device` instance and open or close the NETCONF connection to it automatically. Load the address book configuration using the `rpc.get_config()` method.

9. Start analyzing the address book configuration. For each address element (with configuration contained in security/address-book/address XML XPath) extract its name and IP-prefix. Add ip-prefix to addr_book_prefixes set, but only if the address element conforms to naming convention (name starts with ADDR_NAME_PREFIX following actual IP prefix). Ignore other addresses that might have been added to the address book manually.

10. Proceed analyzing the address book configuration and analyze all addresses in the address-set named ADDR_SET_NAME. For addresses that start with ADDR_NAME_PREFIX make sure the actual IP subnet is contained in addr_book_prefixes before adding it to address_set_prefixes.

11. Process any possible exceptions, and return address_set_prefixes (empty set is returned in the case of device connection or authentication errors).

12. This is an auxiliary function that prints an address set. Note that addresses are sorted lexicographically before printing.

13. The function `sanitize_ip()` in the script is used to check if an argument is a correct IPv4 address or subnet. It returns `None` if not, otherwise it returns the IPv4 subnet string. A built-in Python module `ipaddress` is used here for simplicity. An alternative implementation could be based on straightforward regular expressions.

14. The `change_config_with_set_commands()` function is used to apply configuration changes to the device. Note the two nested context managers: the first one is used to create a `Device` instance and open/close connection, while the second (inner) one creates a `Config` object instance and locks or unlocks the configuration database. Inside the context managers you only have to load and commit configuration – and print the delta config output for user's convenience (`pdiff()` method).

15. The `add_address()` function calls the function defined previously, namely `change_config_with_set_commands()`, to add an address to the address book. STR_SET_CONFIG is used as a template.

16. The `del_address()` function works very similarly, but for the set command template it uses STR_DELETE_CONFIG instead.

17. In the main function you have an infinite “while True” loop (to abandon such a loop use the `break` operator). In the loop, you print the prompt and ask for the user to input instruction.

18. Entering “q” (or “Q”) finishes the script.

19. Entering “r” (or “R”) reads and prints the address list from a device. In case “inconsistent” configuration is detected, the script finishes.

20. Entering “a” (or “A”) asks the user to enter an IP address or subnet, and then adds it to the address book using `add_address()` function. Function `sanitize_ip()` is used to make sure user input is valid.

21. Entering “d” (or “D”) also asks the user to enter IP address or subnet, and then calls `del_address()` function.

22. Standard Python script “entry point.”

NEXT

Now, let’s test the script! In the example below you start with an empty address book on the SRX device (that is, no address book ADDR_BOOK_NAME is defined in the configuration). Then you add 10.1.1.0/24 and 10.2.2.2/32 addresses with a script, and then delete them just to check that everything works fine:

```
lab@host$ python3 address_edit.py
Address-set editor script for Juniper SRX. Device: 10.254.0.35
Address book to edit: global   Address-set to edit: ALLOWED-IN
  r - READ and show addresses
  a - ADD address (IP or subnet)
  d - DELETE particular address
  q - QUIT script
Choice >>> a
Enter IP or IP/mask to add >>> 10.1.1.0/24

Config diff on the device:

[edit security]
+  address-book {
+     global {
+        address CIDR-10.1.1.0/24 10.1.1.0/24;
+        address-set ALLOWED-IN {
+           address CIDR-10.1.1.0/24;
+        }
+     }
+  }
```

```
Configuration change committed.

Address-set editor script for Juniper SRX. Device: 10.254.0.35
Address book to edit: global   Address-set to edit: ALLOWED-IN
  r - READ and show addresses
  a - ADD address (IP or subnet)
  d - DELETE particular address
  q - QUIT script
Choice >>> a
Enter IP or IP/mask to add >>> 10.2.2.2

Config diff on the device:

[edit security address-book global]
     address CIDR-10.1.1.0/24 { ... }
+    address CIDR-10.2.2.2/32 10.2.2.2/32;
[edit security address-book global address-set ALLOWED-IN]
      address CIDR-10.1.1.0/24 { ... }
+     address CIDR-10.2.2.2/32;

Configuration change committed.

Address-set editor script for Juniper SRX. Device: 10.254.0.35
Address book to edit: global   Address-set to edit: ALLOWED-IN
  r - READ and show addresses
  a - ADD address (IP or subnet)
  d - DELETE particular address
  q - QUIT script
Choice >>> r

Reading and displaying address book entries:
10.1.1.0/24
10.2.2.2/32

Address-set editor script for Juniper SRX. Device: 10.254.0.35
Address book to edit: global   Address-set to edit: ALLOWED-IN
  r - READ and show addresses
  a - ADD address (IP or subnet)
  d - DELETE particular address
  q - QUIT script
Choice >>> d
Enter IP/mask to delete >>> 10.1.1.0/24

Config diff on the device:

[edit security address-book global]
-    address CIDR-10.1.1.0/24 10.1.1.0/24;
[edit security address-book global address-set ALLOWED-IN]
-     address CIDR-10.1.1.0/24;

Configuration change committed.
```

```
Address-set editor script for Juniper SRX. Device: 10.254.0.35
Address book to edit: global   Address-set to edit: ALLOWED-IN
  r - READ and show addresses
  a - ADD address (IP or subnet)
  d - DELETE particular address
  q - QUIT script
Choice >>> r

Reading and displaying address book entries:
10.2.2.2/32

Address-set editor script for Juniper SRX. Device: 10.254.0.35
Address book to edit: global   Address-set to edit: ALLOWED-IN
  r - READ and show addresses
  a - ADD address (IP or subnet)
  d - DELETE particular address
  q - QUIT script
Choice >>> d
Enter IP/mask to delete >>> 10.2.2.2

Config diff on the device:

[edit security]
-  address-book {
-     global {
-         address CIDR-10.2.2.2/32 10.2.2.2/32;
-         address-set ALLOWED-IN {
-             address CIDR-10.2.2.2/32;
-         }
-     }
-  }

Configuration change committed.

Address-set editor script for Juniper SRX. Device: 10.254.0.35
Address book to edit: global   Address-set to edit: ALLOWED-IN
  r - READ and show addresses
  a - ADD address (IP or subnet)
  d - DELETE particular address
  q - QUIT script
Choice >>> q
Goodbye!
```

Looks good so far. Checking the configuration on the SRX device also confirms that the configuration has been added and deleted successfully – the checks are straightforward and this recipe omits those outputs for the sake of brevity. Try it in your own lab to see.

Discussion

There are a couple more things you need to check before calling it a day. What if the SRX device connection is down? What if NETCONF is not enabled on the device? What if someone edited the configuration manually in an improper way?

First, let's see what happens to the script session after NETCONF was purposefully disabled on the SRX device:

```
lab@host$ python3 address_edit.py
Address-set editor script for Juniper SRX. Device: 10.254.0.35
Address book to edit: global   Address-set to edit: ALLOWED-IN
  r - READ and show addresses
  a - ADD address (IP or subnet)
  d - DELETE particular address
  q - QUIT script
Choice >>> r

Reading and displaying address book entries:


Error: Connection refused!

Address-set editor script for Juniper SRX. Device: 10.254.0.35
Address book to edit: global   Address-set to edit: ALLOWED-IN
  r - READ and show addresses
  a - ADD address (IP or subnet)
  d - DELETE particular address
  q - QUIT script
Choice >>>
```

As you can see, the ConnectRefusedError exception was properly processed by the change_config_with_set_commands() function. The connection timeout error (ConnectTimeoutError exception) and the user authentication error (ConnectAuthError exception) will be processed similarly.

What about configuration lock? Let's say you have some candidate configuration changes on your SRX device that have not been applied yet:

```
[edit]
lab@SRX# show | compare
[edit interfaces ge-0/0/0]
+   description “Test lock”;
```

Reading the address book will work fine, but adding or deleting entries should fail as you require an exclusive lock to perform changes from the script. Let's see:

```
lab@host$ python3 address_edit.py
Address-set editor script for Juniper SRX. Device: 10.254.0.35
Address book to edit: global   Address-set to edit: ALLOWED-IN
  r - READ and show addresses
  a - ADD address (IP or subnet)
  d - DELETE particular address
  q - QUIT script
Choice >>> a
```

```
Enter IP or IP/mask to add >>> 10.3.3.0/24


Error applying config: configuration was locked!

Address-set editor script for Juniper SRX. Device: 10.254.0.35
Address book to edit: global   Address-set to edit: ALLOWED-IN
  r - READ and show addresses
  a - ADD address (IP or subnet)
  d - DELETE particular address
  q - QUIT script
Choice >>>
```

As expected, the LockError exception was again processed properly. You should start realizing the usefulness of the `try` and `except` operators if you have not done so yet!

Finally, let's see what happens if someone modifies the configuration to make it "inconsistent" from the point of view of our script. The following is an example of such configuration:

```
 [edit]
lab@SRX# show security address-book
global {
    address CIDR-WRONGIP 10.10.10.0/24;
    address-set ALLOWED-IN {
        address CIDR-WRONGIP;
    }
}
```

And here is the result of running the script:

```
lab@host$ python3 address_edit.py
Address-set editor script for Juniper SRX. Device: 10.254.0.35
Address book to edit: global   Address-set to edit: ALLOWED-IN
  r - READ and show addresses
  a - ADD address (IP or subnet)
  d - DELETE particular address
  q - QUIT script
Choice >>> r

Reading and displaying address book entries:
Error: Address book on the device requires manual fix before this script can be used.
```

This is again expected – because "WRONGIP" is not actually an IP address or subnet, and the script asks the operator to sanitize the configuration first.

Recipe 20 - Provisioning L3VPN Services on PE Routers

by Peter Klimai

- Python Version Used: 3.6
- PyEZ Version Used: 2.1.5
- Junos OS Used: 17.1R2.7
- Juniper Platforms General Applicability: MX/PTX/QFX/ACX Series

Manually provisioning network services can be painful. Automating this process improves speed of service delivery and reduces errors. This recipe shows how L3VPN services can be configured on MPLS PE routers using Junos PyEZ.

Problem

You have a set of provider edge (PE) routers and each PE generally has multiple connected L3VPN customers. You want to provision the corresponding configuration automatically using a script.

The example topology used in this recipe is shown in Figure 20.1. The IP/MPLS backbone is preconfigured and not managed by the script. Edge customer-facing interfaces and L3VPN VRF instances must be provisioned automatically by a Python script, according to data specified in a separate file.

Additional points to consider are:

The configuration used for each customer must be standardized using a configuration template. This recipe will use a Jinja2 template engine.

All variable parameters such as customers' AS numbers and IP addresses, as well as mappings of customers to PE devices, must be separated from the template and stored in the YAML file.

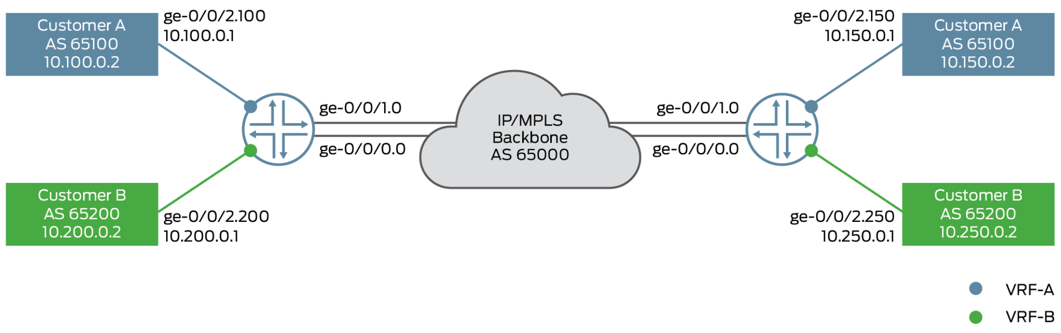


Figure 20.1 Network Topology Used in Recipe 20

For simplicity, assume that each customer has no more than one Layer 3 connection to each of the PEs.

The script must allow for easily adding and removing customers from PE devices, as well as modifying any service parameters.

Solution

The solution includes three files:

- a Python provisioning script,
- a Jinja2 configuration template,
- and a YAML file with variable parameters.

The first two files are created once and are not supposed to be modified unless you are implementing some new functionality (for example, adding configuration options to the template). On the other hand, the YAML file is changed every time you want to add, remove, or modify services. You will generally want to keep such files in a version control system repository such as Git.

Provisioning the Script

The code of the L3VPN service provisioning script named `provision_l3vpn.py` is presented below. The script's parts are numbered as `# (n)` for reference in the explanation that follows:

```
#!/usr/bin/python3

from jnpr.junos import Device                          # (1)
from jnpr.junos.utils.config import Config
from jnpr.junos.exception import *
import jinja2
import os
import yaml

USER = “lab”                                    # (2)
PASSWD = “lab123”

def render(full_filename, context):                 # (3)
   path, filename = os.path.split(full_filename)
   template = jinja2.Environment(
      loader=jinja2.FileSystemLoader(path or ‘./’)).get_template(filename)
   return template.render(context)

def main():                                     # (4)
   with open(“l3vpn-data.yaml”) as var_file:
      l3vpn_data = yaml.load(var_file)

   for PE in l3vpn_data[“PEs”]:                        # (5)
      print(“Working on device %s” % PE)
      vars = l3vpn_data[“PEs”][PE].copy()                 # (6)
      vars.update({“customers”: l3vpn_data[“customers”]})
      result_conf = render(“l3vpn_config.jinja2”, vars)      # (7)

      try:                                       # (8)
         with Device(host=l3vpn_data[“PEs”][PE][“management_ip”],
                  user=USER, password=PASSWD) as dev:
            # open and close is done automatically by context manager
            with Config(dev, mode=”exclusive”) as conf:
               # exclusive locks are treated automatically by context manager
               conf.load(result_conf, format=”text”)
               diff = conf.diff()                     # (9)
               if diff is None:
                  print(“Configuration is up to date.”)
               else:
                  print(“Config diff to be committed on device:”)
                  print(diff)
                  conf.commit()                      # (10)
      except LockError:
         print(“\nError applying config: configuration was locked!”)
      except ConnectRefusedError:
         print(“\nError: Device connection refused!”)
      except ConnectTimeoutError:
         print(“\nError: Device connection timed out!”)
      except ConnectAuthError:
         print(“\nError: Authentication failure!”)
      except ConfigLoadError as ex:
         print(“\nError: “ + str(ex))
      else:                                       # (11)
         if diff is not None:
            print(“Config committed successfully!”)

if __name__ == “__main__”:                            # (12)
   main()
```

And here is the explanation of what is happening in the script:

1. Import the required Junos PyEZ classes, as well as the `jinja2`, `os` and `yaml` packages.

2. Login and password for PE devices connection. In a production environment, you should consider using SSH keys instead.

3. The function `render()` takes two parameters: template filename and context (variable data that will be substituted into the template). It uses the `jinja2` library and returns the rendered template (part of the device configuration in this case).

4. The main function starts with opening a YAML file named `l3vpn-data.yaml` and reading the data from it. The result is placed in the `l3vpn_data` variable.

5. The `l3vpn_data` variable has a complex structure, which is demonstrated in a section below. At the top level, it is a dictionary with one of the keys named "*PEs*." The value corresponding to this key is another dictionary. Here you iterate over this inner dictionary's keys (basically, over a set of PE devices).

6. The `vars` variable is our context that will be passed to `render()` function. Here you put in `vars,` the data for the current PE, and also add the data contained in the "customers" key. Looking at the YAML example below should make it clearer.

7. Render the template stored in `l3vpn_config.jinja2` file and store the result in `result_conf` variable.

8. Try/except block. Inside the block is where you open the PE device connection and start configuring the device using exclusive mode (note two nested context managers). Load the configuration from `result_conf`.

9. Get the configuration difference (basically a `show | compare`) and store to the `diff` variable.

10. Commit, but only if `diff` is *not None* – otherwise no commit is needed. Process possible exceptions.

11. The `else` block of the try/except operator is only executed if no exceptions were raised.

12. Standard Python script "entry point."

Configuration Template

Okay, to create a configuration template easily, the most natural approach is to start with a configuration piece that you want to get from that template. In this case, you want to get something like this – but for each of the customers on each PE:

```
interfaces {
   ge-0/0/2 {
      unit 100 {
         vlan-id 100;
         family inet {
            address 10.100.0.1/24;
         }
```

```
        }
    }
}
routing-instances {
    Cust_A {
        instance-type vrf;
        interface ge-0/0/2.100;
        vrf-target target:65000:1;
        vrf-table-label;
        protocols {
            bgp {
                group EBGP-Cust_A {
                    family inet {
                        unicast {
                            prefix-limit {
                                maximum 10;
                                teardown;
                            }
                        }
                    }
                    peer-as 65100;
                    as-override;
                    neighbor 10.100.0.2;
                }
            }
        }
    }
```

All the parameters here, such as interface names, AS numbers, and more, must be replaced with variables passed to the template from the outside. So the Jinja2 configuration template will include logical interfaces and VRF configurations for your connected customers, but it will also have some operators (such as `if` and `for`) and references to these variables. Using the above example, you should come up with something akin to the following template:

```
groups {
    replace:
    L3VPN-SCRIPT {
        {% if VPN_data %}
        interfaces {
        {% for VPN_entry in VPN_data %}
            {{ VPN_entry.interface_name }} {
                unit {{ VPN_entry.unit }} {
                    vlan-id {{ VPN_entry.vlan_id }};
                    family inet {
                        address {{ VPN_entry.ip_mask }};
                    }
                }
            }
        {% endfor %}
        }
        routing-instances {
        {% for VPN_entry in VPN_data %}
            {{ VPN_entry.customer_id }} {
                instance-type vrf;
                vrf-table-label;
                interface {{ VPN_entry.interface_name }}.{{ VPN_entry.unit }};
                vrf-target {{ customers[VPN_entry.customer_id].vrf_target }};
                protocols {
                    bgp {
                        group EBGP-{{ VPN_entry.customer_id }} {
                            family inet {
                                unicast {
                                    prefix-limit {
                                        maximum {{ VPN_entry.prefix_limit }};
                                        teardown;
                                    }
                                }
                            }
                            peer-as {{ customers[VPN_entry.customer_id].AS }};
                            as-override;
                            neighbor {{ VPN_entry.customer_ip }};
                        }
                    }
                }
            }
        {% endfor %}
        }
        {% endif %}
    }
}
apply-groups L3VPN-SCRIPT;
```

Several points to mention:

- All template configuration is put inside the Junos configuration group named `L3VPN-SCRIPT` so that it is easy to see which part of the configuration was actually generated by the script. It also simplifies configuration modification or removal by the same script later.
- The `replace:` tag ensures that previous content of `L3VPN-SCRIPT` group is overwritten.
- The `if` operator is checking if `VPN_data` is empty or not. If it is empty, just the empty configuration group is created.
- The `for` operators are used to loop over multiple entries, corresponding to each of the customers connected to a given PE.
- Note how customer data is put inside the `customers` dictionary. You do not iterate over it in the template – but query the dictionary as needed while iterating over `VPN_entry`.

NOTE For more information on Jinja2 template engine and available operators visit: http://jinja.pocoo.org.

The Data (YAML) File

The example for the content of `l3vpn-data.yaml` file, corresponding to network topology shown in Figure 20.1, is as follows:

```
---
customers:
  Cust_A:
    vrf_target: “target:65000:1”
    AS: 65100
  Cust_B:
    vrf_target: “target:65000:2”
    AS: 65200

PEs:
  PE1:
    management_ip: “10.254.0.41”
    VPN_data:
      - customer_id: Cust_A
        interface_name: ge-0/0/2
        unit: 100
        vlan_id: 100
        ip_mask: 10.100.0.1/24
        customer_ip: 10.100.0.2
        prefix_limit: 10
      - customer_id: Cust_B
        interface_name: ge-0/0/2
        unit: 200
        vlan_id: 200
        ip_mask: 10.200.0.1/24
        customer_ip: 10.200.0.2
        prefix_limit: 15
  PE2:
    management_ip: “10.254.0.42”
    VPN_data:
      - customer_id: Cust_A
        interface_name: ge-0/0/2
        unit: 150
        vlan_id: 150
        ip_mask: 10.150.0.1/24
        customer_ip: 10.150.0.2
        prefix_limit: 10
      - customer_id: Cust_B
        interface_name: ge-0/0/2
        unit: 250
        vlan_id: 250
        ip_mask: 10.250.0.1/24
        customer_ip: 10.250.0.2
        prefix_limit: 15
```

When the script reads this YAML data, the result is put into the `l3vpn_data` variable. If you printed out the contents of this variable (using the `pprint()` function from the `pprint` module) it would look like this:

```
{‘PEs’: {‘PE1’: {‘VPN_data’: [{‘customer_id’: ‘Cust_A’,
                              ‘customer_ip’: ‘10.100.0.2’,
                              ‘interface_name’: ‘ge-0/0/2’,
                              'ip_mask': '10.100.0.1/24',
                              'prefix_limit': 10,
                              'unit': 100,
                              'vlan_id': 100},
                             {'customer_id': 'Cust_B',
                              'customer_ip': '10.200.0.2',
                              'interface_name': 'ge-0/0/2',
                              'ip_mask': '10.200.0.1/24',
                              'prefix_limit': 15,
                              'unit': 200,
                              'vlan_id': 200}],
                   'management_ip': '10.254.0.41'},
            'PE2': {'VPN_data': [{'customer_id': 'Cust_A',
                              'customer_ip': '10.150.0.2',
                              'interface_name': 'ge-0/0/2',
                              'ip_mask': '10.150.0.1/24',
                              'prefix_limit': 10,
                              'unit': 150,
                              'vlan_id': 150},
                             {'customer_id': 'Cust_B',
                              'customer_ip': '10.250.0.2',
                              'interface_name': 'ge-0/0/2',
                              'ip_mask': '10.250.0.1/24',
                              'prefix_limit': 15,
                              'unit': 250,
                              'vlan_id': 250}],
                   'management_ip': '10.254.0.42'}},
     'customers': {'Cust_A': {'AS': 65100, 'vrf_target': 'target:65000:1'},
                  'Cust_B': {'AS': 65200, 'vrf_target': 'target:65000:2'}}}
```

REMEMBER This output should be helpful to analyze how the main script works. Keep in mind that dictionaries use curly braces while lists use square brackets.

Note the two dictionary keys at the top level in the output: "PEs" and "customers". The meaning should be clear: "PEs" contain data for the PE devices, including L3VPN services that must be provisioned, as well as management IP data. The "customers" refers to a nested dictionary storing important information about each of the customers – AS number and VRF target. Customer names ("Cust_A", "Cust_B") are used as keys for this nested dictionary. Note also how using a separate "customers" dictionary allows you to not duplicate the same customer information for each PE.

Running the Script for Provisioning L3VPN Services

The configuration on PE devices at this point includes:

- Full configuration of core-facing interfaces (family inet and MPLS);
- Standard OSPF, LDP, and IBGP (with `family inet-vpn unicast`) configuration for the IP/MPLS backbone;

- Only physical parameters for customer-facing interfaces (ge-0/0/2) are configured – namely, `flexible-vlan-tagging` and `encapsulation flexible-ethernet-services` are configured. No units are configured on these interfaces – the script must do that;
- No VRF (L3VPN) instances are configured for the customers – again, the script must do that;
- The `route-distinguisher-id` is configured in `routing-options` hierarchy on both PEs, so manual configuration for route-distinguisher in VRFs is not needed.

Let's begin by running the script with a version of `l3vpn-data.yaml` YAML file containing only Customer-A data, namely:

```
---
customers:
  Cust_A:
    vrf_target: "target:65000:1"
    AS: 65100

PEs:
  PE1:
    management_ip: "10.254.0.41"
    VPN_data:
      - customer_id: Cust_A
        interface_name: ge-0/0/2
        unit: 100
        vlan_id: 100
        ip_mask: 10.100.0.1/24
        customer_ip: 10.100.0.2
        prefix_limit: 10
  PE2:
    management_ip: "10.254.0.42"
    VPN_data:
      - customer_id: Cust_A
        interface_name: ge-0/0/2
        unit: 150
        vlan_id: 150
        ip_mask: 10.150.0.1/24
        customer_ip: 10.150.0.2
        prefix_limit: 10
```

The resulting output after running the script looks like this:

```
lab@host:~$ python3 provision_l3vpn.py
Working on device PE1
Config diff to be committed on device:

[edit]
+ groups {
+     L3VPN-SCRIPT {
+         interfaces {
+             ge-0/0/2 {
```

```
+                unit 100 {
+                    vlan-id 100;
+                    family inet {
+                        address 10.100.0.1/24;
+                    }
+                }
+            }
+        }
+        routing-instances {
+            Cust_A {
+                instance-type vrf;
+                interface ge-0/0/2.100;
+                vrf-target target:65000:1;
+                vrf-table-label;
+                protocols {
+                    bgp {
+                        group EBGP-Cust_A {
+                            family inet {
+                                unicast {
+                                    prefix-limit {
+                                        maximum 10;
+                                        teardown;
+                                    }
+                                }
+                            }
+                            peer-as 65100;
+                            as-override;
+                            neighbor 10.100.0.2;
+                        }
+                    }
+                }
+            }
+        }
+    }
+ }
+ apply-groups L3VPN-SCRIPT;

Config committed successfully!
Working on device PE2
Config diff to be committed on device:

[edit]
+ groups {
+    L3VPN-SCRIPT {
+        interfaces {
+            ge-0/0/2 {
+                unit 150 {
+                    vlan-id 150;
+                    family inet {
+                        address 10.150.0.1/24;
+                    }
+                }
+            }
+        }
+        routing-instances {
+            Cust_A {
+                instance-type vrf;
+                interface ge-0/0/2.150;
+               vrf-target target:65000:1;
+               vrf-table-label;
+               protocols {
+                   bgp {
+                       group EBGP-Cust_A {
+                           family inet {
+                               unicast {
+                                   prefix-limit {
+                                       maximum 10;
+                                       teardown;
+                                   }
+                               }
+                           }
+                           peer-as 65100;
+                           as-override;
+                           neighbor 10.150.0.2;
+                       }
+                   }
+               }
+           }
+       }
+   }
+ }
+ apply-groups L3VPN-SCRIPT;

Config committed successfully!
```

Looks good so far! Let's check out the configuration on PE1:

```
[edit]
lab@PE-1# show
## Last changed: 2017-08-17 13:08:16 UTC
version 17.1R2.7;
groups {
    L3VPN-SCRIPT {
        interfaces {
            ge-0/0/2 {
                unit 100 {
                    vlan-id 100;
                    family inet {
                        address 10.100.0.1/24;
                    }
                }
            }
        }
        routing-instances {
            Cust_A {
                instance-type vrf;
                interface ge-0/0/2.100;
                vrf-target target:65000:1;
                vrf-table-label;
                protocols {
                    bgp {
                        group EBGP-Cust_A {
                            family inet {
                                unicast {
                                    prefix-limit {
                                        maximum 10;
```

```
                              teardown;
                           }
                        }
                     }
                     peer-as 65100;
                     as-override;
                     neighbor 10.100.0.2;
                  }
               }
            }
         }
      }
   }
}
apply-groups L3VPN-SCRIPT;
system {
   host-name PE-1;

[ ... ]


[edit]
lab@PE-1# show routing-instances | display inheritance
##
## 'Cust_A' was inherited from group 'L3VPN-SCRIPT'
##
Cust_A {
   ##
   ## 'vrf' was inherited from group 'L3VPN-SCRIPT'
   ##
   instance-type vrf;
   ##
   ## 'ge-0/0/2.100' was inherited from group 'L3VPN-SCRIPT'
   ##
   interface ge-0/0/2.100;

[ ... ]
```

Also, let's verify the routes in the Cust_A VRF instance on PE-1:

```
lab@PE-1> show route table Cust_A.inet.0

Cust_A.inet.0: 3 destinations, 3 routes (3 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

10.100.0.0/24      *[Direct/0] 00:15:39
                 > via ge-0/0/2.100
10.100.0.1/32      *[Local/0] 00:15:39
                   Local via ge-0/0/2.100
10.150.0.0/24      *[BGP/170] 00:15:30, localpref 100, from 192.168.0.2
                   AS path: I, validation-state: unverified
                   to 10.0.0.222 via ge-0/0/0.0, Push 16
                 > to 10.0.1.222 via ge-0/0/1.0, Push 16
```

The route to the remote network is there. You can also check that a ping between remote Customer-A instances also works as it should (the vr-A here is a virtual router-instance created manually on PE-1 just for testing purposes – it emulates

Customer-A's CE device):

```
lab@PE-1> **ping 10.150.0.2 routing-instance vr-A source 10.100.0.2 rapid**
PING 10.150.0.2 (10.150.0.2): 56 data bytes
!!!!!
--- 10.150.0.2 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 2.669/2.895/3.303/0.220 ms
```

Now let's add Customer-B to the `l3vpn-data.yaml` YAML file, making it look exactly as shown in the previous section, and then re-run the script:

```
lab@host:~$ python3 provision_l3vpn.py
Working on device PE1
Config diff to be committed on device:

[edit groups L3VPN-SCRIPT interfaces ge-0/0/2]
+    unit 200 {
+       vlan-id 200;
+       family inet {
+          address 10.200.0.1/24;
+       }
+    }
[edit groups L3VPN-SCRIPT routing-instances]
+   Cust_B {
+      instance-type vrf;
+      interface ge-0/0/2.200;
+      vrf-target target:65000:2;
+      vrf-table-label;
+      protocols {
+         bgp {
+            group EBGP-Cust_B {
+               family inet {
+                  unicast {
+                     prefix-limit {
+                        maximum 15;
+                        teardown;
+                     }
+                  }
+               }
+               peer-as 65200;
+               as-override;
+               neighbor 10.200.0.2;
+            }
+         }
+      }
+   }

Config committed successfully!
Working on device PE2
Config diff to be committed on device:

[edit groups L3VPN-SCRIPT interfaces ge-0/0/2]
+    unit 250 {
+       vlan-id 250;
+       family inet {
```

```
+          address 10.250.0.1/24;
+       }
+    }
[edit groups L3VPN-SCRIPT routing-instances]
+   Cust_B {
+       instance-type vrf;
+       interface ge-0/0/2.250;
+       vrf-target target:65000:2;
+       vrf-table-label;
+       protocols {
+           bgp {
+               group EBGP-Cust_B {
+                   family inet {
+                       unicast {
+                           prefix-limit {
+                               maximum 15;
+                               teardown;
+                           }
+                       }
+                   }
+                   peer-as 65200;
+                   as-override;
+                   neighbor 10.250.0.2;
+               }
+           }
+       }
+   }

Config committed successfully!
```

Thus provisioning of additional customers is successful (to save book space, the outputs that further demonstrate this are omitted).

Modifying Service on the Router

It is extremely simple to modify the service parameters using this recipe's approach – just modify the YAML file and re-run the script!

For example, let's say you want to change Customer-B's AS number from 65200 to 65300. You can change this in the "customers" section of `l3vpn-data.yaml` file and run the script again:

```
lab@host:~$ python3 provision_l3vpn.py
Working on device PE1
Config diff to be committed on device:

[edit groups L3VPN-SCRIPT routing-instances Cust_B protocols bgp group EBGP-Cust_B]
-      peer-as 65200;
+      peer-as 65300;

Config committed successfully!
Working on device PE2
Config diff to be committed on device:
```

```
[edit groups L3VPN-SCRIPT routing-instances Cust_B protocols bgp group EBGP-Cust_B]
-      peer-as 65200;
+      peer-as 65300;

Config committed successfully!
```

Just what was required!

Removing Service from the Router

Now let's say you want to delete all the VPN services from PE-2. In this case, just edit the `l3vpn-data.yaml` file so that the corresponding section looks like this:

```
 [...]

  PE2:
    management_ip: "10.254.0.42"
    VPN_data:
```

There's nothing in VPN_data, making it an empty dictionary, so run the script:

```
lab@host:~$ python3 provision_l3vpn.py
Working on device PE1
Configuration is up to date.
Working on device PE2
Config diff to be committed on device:

[edit groups L3VPN-SCRIPT]
-   interfaces {
-      ge-0/0/2 {
-         unit 150 {
-            vlan-id 150;
-            family inet {
-               address 10.150.0.1/24;
-            }
-         }
-         unit 250 {
-            vlan-id 250;
-            family inet {
-               address 10.250.0.1/24;
-            }
-         }
-      }
-   }
-   routing-instances {
-      Cust_A {
-         instance-type vrf;
-         interface ge-0/0/2.150;
-         vrf-target target:65000:1;
-         vrf-table-label;
-         protocols {
-            bgp {
-               group EBGP-Cust_A {
-                  family inet {
```

```
-                       unicast {
-                           prefix-limit {
-                               maximum 10;
-                               teardown;
-                           }
-                       }
-                   }
-                   peer-as 65100;
-                   as-override;
-                   neighbor 10.150.0.2;
-               }
-           }
-       }
-   }
-   Cust_B {
-       instance-type vrf;
-       interface ge-0/0/2.250;
-       vrf-target target:65000:2;
-       vrf-table-label;
-       protocols {
-           bgp {
-               group EBGP-Cust_B {
-                   family inet {
-                       unicast {
-                           prefix-limit {
-                               maximum 15;
-                               teardown;
-                           }
-                       }
-                   }
-                   peer-as 65300;
-                   as-override;
-                   neighbor 10.250.0.2;
-               }
-           }
-       }
-   }
- }

Config committed successfully!
```

The services are removed from PE-2.

Note that because configuration for PE1 did not change in the above script run, no commit was even required (and this was properly detected by the script). Thus, it does not hurt to perform additional runs – such property of our provisioning script is called *idempotence* – check out Wikipedia if you are interested.

Discussion

In Recipe 20, the L3VPN configuration is regenerated from the template and YAML file data every time the script runs. Then it is uploaded to the PE device and committed, reporting the configuration difference. Thus, the YAML file turns out to be our "system of record" – an authoritative source of L3VPN customer data. At large scale, you might want to consider using a database (such as SQL database) instead of the single YAML file.

All configuration provisioned by a script was put in a separate configuration group. Such a method has several advantages, including consistency and visibility into what the automation system is doing. Also, changing and removing of services with such an approach is really easy.

One can object, noting that regenerating the whole service configuration every time might be resource consuming. That is a valid concern and it is possible to include optimizations to the deployment script that will only upload the delta-configuration to the device. However, as with any optimization, you first want to make sure you actually have the bottleneck (and locate it). In the recipe's lab environment, we have confirmed experimentally that even for 1000 VRFs per PE, the script presented here still provisions the configuration in less than 30 seconds. So, unless you have a really large-scale deployment, optimization may just not be needed.

This recipe focused on the deployment of L3VPN services – but of course, multiple other services and configurations can be provisioned similarly, as long as you can come up with a standardized template for them.

Recipe 21 - Identifying and Disabling Unused Interfaces with Ansible

by Sean Sawtell

- Python Version Used: 2.7
- PyEZ Version Used: 2.1.5
- Junos OS Used: mixed
- Juniper Platforms General Applicability: MX, EX, SRX

If you are using Ansible and PyEZ to automate network operations, then this recipe will show you how to create an Ansible playbook and a custom Ansible module that can disable unused Ethernet interfaces on Junos devices by using PyEZ operational Tables and Views.

Problem

You wish to disable unused Ethernet interfaces on your network devices. Some network monitoring systems report status on all enabled or "admin-up" interfaces, and an enabled but unused interface will cause the monitoring system to generate "interface down" messages for those interfaces – but disabling the unused interfaces can avoid such false alarms. Disabling unused interfaces, particularly on switches, may also help prevent unauthorized devices from connecting to the network, inadvertent or malicious connections between different subnets, and other similar problems.

How do we know if an interface is unused? For purposes of this recipe, an "unused" interface will be one with no link and no subinterfaces (units). We will check for subinterfaces because it is easier than checking the device's configuration to see if the interface is configured. There are a number of places in the configuration where an interface may be configured, each having a slightly different syntax. However, each of the various interface configuration methods results in one or more subinterfaces being associated with the interface.

Automation with PyEZ can determine which interfaces are unused. We will write a custom Ansible module using PyEZ that will determine which interfaces are unused and return that data to the calling Ansible playbook.

We will also write an Ansible playbook and supporting files to change a device's configuration based on the results returned by the module.

NOTE While this recipe assumes you are using Ansible, the code in the custom module could be easily modified to run as a stand-alone program and extended to make the configuration change using PyEZ.

NOTE This recipe assumes you have a working Ansible configuration, including your device inventory and the Juniper.junos galaxy modules. The discussion about the solution focuses on the custom module and its use of PyEZ. The author assumes you are familiar enough with Ansible to understand the playbook and related files with minimal explanation.

Solution

The Ansible playbook for this recipe needs to accomplish three major tasks for each device:

- Call the custom module that will determine which interfaces are unused, and capture that list of interfaces in a variable.
- Use a Jinja2 template to create a Junos configuration file that disables the unused interfaces.
- Push the configuration change to the device.

However, there are a few additional tasks that will make the playbook easier and safer to use, and easier to troubleshoot, so the final playbook will do the following:

- Test NETCONF connectivity to the device with a five second timeout. This quickly stops processing for a device that is unreachable, rather than waiting 30 seconds for the next step to time out. (See lines 19-23.)
- Call the custom module that will determine which interfaces are unused, and capture that list of interfaces in variable `j_ints`. (See lines 25-29.)
- Display the unused interface list if the playbook is run with `–v` (verbose mode). (See lines 31-34.)

- Create the directory that will hold the configuration files, if it does not already exist. (See lines 36-39.)
- Generate the filename for the configuration file. Because the filename will be used in two of the following tasks, we should create it once and store it in a variable. (See lines 41-43.)
- Use a Jinja2 template to create a Junos configuration file that disables the unused interfaces. (See lines 45-48.)
- Push the configuration change to the device using `commit confirmed` so that the device can roll back the change should connectivity be lost. (See lines 50-61.)
- Confirm the `commit`. When the playbook connects to the device to complete this task, it provides a quick check to ensure that disabling the interfaces did not break the device. (See lines 69-74.)
- Delete the configuration file (clean up). (See lines 63-66.)

This is the complete playbook *disable-unused-interfaces.yaml*, with line numbers added for easy reference:

```
 1|---
 2|# Query a device to find out which Ethernet interfaces are unused
 3|# (have no link or logical interface) and disable those interfaces.
 4|
 5|- name: Disable unused Ethernet interfaces
 6|  hosts:
 7|    - all
 8|  roles:
 9|    - Juniper.junos
10|  connection: local
11|  gather_facts: no
12|
13|  vars:
14|    config_dir: ‘configs’
15|    netconf_port: 830
16|    template_dir: ‘templates’
17|
18|  tasks:
19|    - name: check netconf connectivity
20|      wait_for:
21|        host: “{{ ansible_host }}”
22|        port: “{{ netconf_port }}”
23|        timeout: 5
24|
25|    - name: get unused interfaces
26|      junos_unused_interfaces:
27|        host: “{{ ansible_host }}”
28|        port: “{{ netconf_port }}”
29|      register: j_ints
30|
31|    - name: display unused interfaces in verbose mode
32|      debug:
33|        var: j_ints
```

```
34|       verbosity: 1
35|
36|   - name: confirm/create configs directory
37|     file:
38|       path: “{{ config_dir }}”
39|       state: directory
40|
41|   - name: generate filename for config file
42|     set_fact:
43|       config_file: “{{ config_dir }}/unused-ints-{{ inventory_hostname }}.conf”
44|
45|   - name: generate configuration file
46|     template:
47|       src: “{{ template_dir }}/unused-interfaces.j2”
48|       dest: “{{ config_file }}”
49|
50|   - name: push configuration change to device
51|     junos_install_config:
52|       host: “{{ ansible_host }}”
53|       file: “{{ config_file }}”
54|       port: “{{ netconf_port }}”
55|       timeout: 120
56|       comment: “playbook disable-unused-interfaces, commit confirmed 10”
57|       confirm: 10
58|       replace: true
59|       overwrite: false
60|     notify:
61|       - confirm config commit
62|
63|   - name: delete configuration file
64|     file:
65|       path: “{{ config_file }}”
66|       state: absent
67|
68| handlers:
69|   - name: confirm config commit
70|     junos_commit:
71|       host: “{{ ansible_host }}”
72|       port: “{{ netconf_port }}”
73|       timeout: 120
74|       comment: “playbook disable-unused-interfaces, confirm previous commit”
```

The playbook assumes that Jinja2 templates are in the subdirectory *templates* within the playbook directory, and that the device configuration files generated by the playbook will be stored in subdirectory *configs* within the playbook directory. If your Ansible environment has different requirements, adjust the appropriate variables near the top of the playbook (lines 14 or 16).

Let’s use an `interface-range` to disable the unused interfaces. In the Junos `interfaces` configuration hierarchy, the result will look similar to this:

```
/* administratively disable unused interfaces */
interface-range unused {
   member ge-0/0/2;
   member ge-0/0/3;
   member ge-0/0/4;
```

```
    member ge-0/0/5;
    member ge-0/0/6;
    member ge-0/0/7;
    description unused;
    disable;
}
```

The Jinja2 template that will create the configuration is:

```
#jinja2: lstrip_blocks: True
interfaces {
  {% if j_ints.interfaces.unused %}
    /* administratively disable unused interfaces */
    replace:
    interface-range unused {
        description "unused";
        disable;
        {% for interface in j_ints.interfaces.unused %}
        member {{ interface }};
        {% endfor %}
    }
  {% else %}
    {# j_ints.interfaces.unused was empty or undefined, #}
    {# so delete any existing interface-range unused #}
    interface-range unused {
        member {{ j_ints.interfaces.configured[0] }};
    }
    delete: interface-range unused;
  {% endif %}
}
```

Save this in your *templates* directory as *unused-interfaces.j2*.

Ansible can load custom modules from a few places, but the easiest is the *library* subdirectory within your playbook directory. Create a subdirectory *library* if it does not already exist, then create the file *junos_unused_interfaces* within that directory. This module is a Python program, but do not add the extension *.py* to the filename – the module's filename must match the module call on line 26 of the playbook.

Module *junos_unused_interfaces* contains the following Python program (line numbers added for easy reference):

```
 1|#!/usr/bin/env python
 2|
 3|# required Ansible helper module
 4|from ansible.module_utils.basic import AnsibleModule
 5|
 6|# required Python modules
 7|module_import_error = False
 8|try:
 9|    import os
10|    from jnpr.junos import Device
11|    from jnpr.junos.factory import FactoryLoader
12|except ImportError as err:
13|    module_import_error = True
14|    module_msg = 'Error importing required modules: %s' % str(err)
```

```

#######################################################################


def main():

   # define arguments from Ansible
   module = AnsibleModule(
      argument_spec=dict(
         host=dict(required=True),
         user=dict(required=False, default=os.getenv(‘USER’)),
         passwd=dict(required=False, default=None, no_log=True),
         port=dict(required=False, type=’int’, default=830)
         ),
      supports_check_mode=False
      )

   # early exit if required modules failed to import
   if module_import_error:
      module.fail_json(msg=module_msg)

   host = module.params[‘host’]
   username = module.params[‘user’]
   password = module.params[‘passwd’]
   ncport = module.params[‘port’]

   # define an operational table and view
   table_view = {
      ‘EthPortTerseView’: {
         ‘fields’: {
            ‘oper’: ‘oper-status’,
            ‘admin’: ‘admin-status’,
            ‘units’: ‘.//logical-interface/name’,
            ‘name’: ‘name’
         }
      },
      ‘EthPortTerseTable’: {
         ‘item’: ‘physical-interface’,
         ‘rpc’: ‘get-interface-information’,
         ‘args’: {
            ‘terse’: True,
            ‘interface_name’: ‘[fgxe][et]-*’
         },
         ‘args_key’: ‘interface_name’,
         ‘view’: ‘EthPortTerseView’
      }
   }

   try:
      dev = Device(host=host, gather_facts=False, user=username,
                 passwd=password, port=ncport)
      dev.open()
   except Exception as err:
      msg = ‘Error opening device connection: %s’ % str(err)
      module.fail_json(msg=msg)

   try:
      fl = FactoryLoader()
      eth_terse = fl.load(table_view)
      eth_table = eth_terse[‘EthPortTerseTable’](dev)
   except Exception as err:
      msg = ‘Error loading operational table: %s’ % str(err)
      module.fail_json(msg=msg)

   try:
      eth_table.get()
   except Exception as err:
      msg = ‘Error getting interface data: %s’ % str(err)
      module.fail_json(msg=msg)

   try:
      dev.close()
   except Exception as err:
      msg = ‘Error closing device connection: %s’ % str(err)
      module.fail_json(msg=msg)

   used_interfaces = []
   unused_interfaces = []
   link_interfaces = []
   no_link_interfaces = []

   for eth_int in eth_table:
      if eth_int[‘oper’] == ‘up’:
         link_interfaces.append(eth_int[‘name’])
      else:
         no_link_interfaces.append(eth_int[‘name’])

      if (eth_int[‘units’] is not None) or (eth_int[‘oper’] == ‘up’):
         used_interfaces.append(eth_int[‘name’])
      else:
         unused_interfaces.append(eth_int[‘name’])

   result = {‘unused’: unused_interfaces,
             ‘used’: used_interfaces,
             ‘link’: link_interfaces,
             ‘no_link’: no_link_interfaces}
   module.exit_json(changed=False, msg=’Interfaces’, interfaces=result)

######################################################################


if __name__ == ‘__main__’:
   main()
```

Those three files – the playbook itself, the Jinja2 template, and the custom module for finding unused interfaces – complete the solution.

At this point you should be able to run the playbook against a Junos device. The following output is from a run using verbose mode:

```
mbp15:pyez-cookbook sean$ ansible-playbook disable-unused-interfaces.yaml -v --limit=bilbo
Using /Users/sean/pyez-cookbook/ansible.cfg as config file

PLAY [Disable unused Ethernet interfaces] ************************************************************
****
```

```
TASK [check netconf connectivity] ********************************************************************
****
ok: [bilbo] => {“changed”: false, “elapsed”: 0, “path”: null, “port”: 830, “search_regex”: null,
“state”: “started”}

TASK [get unused interfaces] *************************************************************************
****
ok: [bilbo] => {“changed”: false, “interfaces”: {“link”: [“ge-0/0/11”], “no_link”: [“ge-0/0/0”,
“ge-0/0/1”, “ge-0/0/2”, “ge-0/0/3”, “ge-0/0/4”, “ge-0/0/5”, “ge-0/0/6”, “ge-0/0/7”, “ge-0/0/8”,
“ge-0/0/9”, “ge-0/0/10”, “ge-0/1/0”, “ge-0/1/1”], “unused”: [“ge-0/0/0”, “ge-0/0/1”, “ge-0/0/2”,
“ge-0/0/3”, “ge-0/0/4”, “ge-0/0/5”, “ge-0/0/6”, “ge-0/0/7”, “ge-0/1/0”, “ge-0/1/1”], “used”: [“ge-
0/0/8”, “ge-0/0/9”, “ge-0/0/10”, “ge-0/0/11”]}, “msg”: “Interface lists”}

TASK [display unused interfaces in verbose mode] *****************************************************
****
ok: [bilbo] => {
   “j_ints”: {
      “changed”: false,
      “interfaces”: {
         “link”: [
            “ge-0/0/11”
         ],
         “no_link”: [
            “ge-0/0/0”,
            “ge-0/0/1”,
            “ge-0/0/2”,
            “ge-0/0/3”,
            “ge-0/0/4”,
            “ge-0/0/5”,
            “ge-0/0/6”,
            “ge-0/0/7”,
            “ge-0/0/8”,
            “ge-0/0/9”,
            “ge-0/0/10”,
            “ge-0/1/0”,
            “ge-0/1/1”
         ],
         “unused”: [
            “ge-0/0/0”,
            “ge-0/0/1”,
            “ge-0/0/2”,
            “ge-0/0/3”,
            “ge-0/0/4”,
            “ge-0/0/5”,
            “ge-0/0/6”,
            “ge-0/0/7”,
            “ge-0/1/0”,
            “ge-0/1/1”
         ],
         “used”: [
            “ge-0/0/8”,
            “ge-0/0/9”,
            “ge-0/0/10”,
            “ge-0/0/11”
         ]
      },
      “msg”: “Interface lists”
   }
}
```

```
TASK [confirm/create configs directory] ***************************************************************
**
ok: [bilbo] => {"changed": false, "gid": 20, "group": "staff", "mode": "0755", "owner": "sean", "path": 
"configs", "size": 68, "state": "directory", "uid": 502}

TASK [generate filename for config file] **************************************************************
*
ok: [bilbo] => {"ansible_facts": {"config_file": "configs/unused-ints-bilbo.conf"}, "changed": false}

TASK [generate configuration file] ********************************************************************
**
changed: [bilbo] => {"changed": true, "checksum": "c7ec0f8e91de7853117d6557d50dea34336ef24a", "dest": 
"configs/unused-ints-bilbo.conf", "gid": 20, "group": "staff", "md5sum": 
"d1272d7cde3f6f0d5ff3787a19971407", "mode": "0644", "owner": "sean", "size": 413, "src": "/Users/
sean/.ansible/tmp/ansible-tmp-1504216899.17-16860701128770/source", "state": "file", "uid": 502}

TASK [push configuration change to device] ************************************************************
***
ok: [bilbo] => {"changed": false, "file": "/Users/sean/pyez-cookbook/configs/unused-ints-bilbo.conf"}

PLAY RECAP ********************************************************************************************
*****
bilbo                  : ok=7    changed=1    unreachable=0    failed=0
```

Discussion

There are two aspects of the *junos_unused_interfaces* module that warrant discussion. The first is the operational table and view that is at the heart of how the module gathers data from the devices, and the second is how Ansible communicates with the module using the AnsibleModule class.

Operational Tables and Views

The *junos_unused_interfaces* module uses an operational table and view to query a Junos device for interface information and then present that information in a Python-friendly way.

Operational Tables and Views are a PyEZ feature. Operational Tables describe an RPC (remote procedure call) that will be issued to the device and the XML elements we wish to capture from the response. Views provide a way to map the XML data from the table into a Python data structure for easy use in a Python program.

PyEZ includes a number of operational Tables and Views, each described in a YAML data file. None were quite what was needed for this program, though the EthPortTable and EthPortView described in ethport.yml was close. The module defines a new table and view based on ethport.yml but passing a different argument to the RPC and requesting a different set of fields in the view.

Rather than putting this module’s table and view definitions in a separate YAML file, the definitions are in a Python dictionary in the module’s code (lines 41-61). Embedding the definitions means you do not need to worry about maintaining and importing separate a YAML file (though it also makes it difficult to share the table and view between several programs, should that become a consideration).

Let’s take a look at the operational table, called *EthPortTerseTable*:

```
43|      ‘EthPortTerseTable’: {
44|         ‘rpc’: ‘get-interface-information’,
45|         ‘args’: {
46|            ‘terse’: True,
47|            ‘interface_name’: ‘[fgxe][et]-*’
48|         },
49|         ‘item’: ‘physical-interface’,
50|         ‘args_key’: ‘interface_name’,
51|         ‘view’: ‘EthPortTerseView’
52|      },
```

Here, `‘rpc’` identifies the RPC that will be issued to the Junos device. The *get-interface-information* RPC is equivalent to the Junos CLI command `show interfaces`.

The ‘args’ dictionary declares two arguments to the RPC that modify its behavior: ‘terse’ asking for a terse interface listing, and ‘interface_name’ providing a regular expression to match the names of the interfaces in which we are interested. The CLI equivalent command with these arguments is `show interfaces terse “[fgxe][et]-*”`. The regular expression matches any interface name where the first letter is in the group [fgxe], the second letter in [et], followed by a hyphen (-) and then any other characters. Essentially, this will match Junos’ Ethernet interface names: fe-, ge-, xe-, and et-.

The ‘item’ identifies the XML tag for the elements that should be included in the table, in this case the elements contained by `<physical-interface>` tags. Remember that an RPC request and response are in XML, and you can view the XML response at the Junos CLI using the `| display xml` modifier. For example:

```
sean@bilbo> show interfaces terse ge-0/0/0 | display xml
<rpc-reply xmlns:junos=”http://xml.juniper.net/junos/15.1R6/junos”>
   <interface-information xmlns=”http://xml.juniper.net/junos/15.1R6/junos-interface”
junos:style=”terse”>
      <physical-interface>
         <name>ge-0/0/0</name>
         <admin-status>down</admin-status>
         <oper-status>down</oper-status>
      </physical-interface>
   </interface-information>
   <cli>
      <banner>{master:0}</banner>
   </cli>
</rpc-reply>
```

Notice the `<physical-interface>` and `</physical-interface>` opening and closing tags, and the data contained within those tags. The XML results will include one of those for each matching interface. (This example requested only ge-0/0/0 to keep the output short.)

The ‘args_key’ allows the user to specify an argument; in this case, the user of the table could specify an `interface_name` other than the regular expression provided in the ‘args’ dictionary. This is not needed in the final module, but it was helpful during testing and development to be able to request a single interface instead of all Ethernet interfaces on the device.

And the ‘view’ specifies the operational view that should be used to map the XML data in the table to a Python data structure.

Now, let’s take a quick look at the operational view, called *EthPortTerseView*:

```
   ‘EthPortTerseView’: {
      ‘fields’: {
         ‘admin’: ‘admin-status’,
         ‘oper’: ‘oper-status’,
         ‘units’: ‘.//logical-interface/name’,
         ‘name’: ‘name’
      }
   }
```

The ‘fields’ dictionary identifies the data that should be in the Python data structure. For each key:value pair in the ‘fields’ dictionary, the key (for example, ‘admin’) will become a key in a Python <dict> data structure, while the value (for example, ‘admin-status’) references the XML returned by the RPC.

The ‘admin’, ‘oper’, and ‘name’ fields reference the XML tags ‘admin-status’,’oper-status’, and ‘name’, respectively, all elements within the XML contents of the <physical-interface> tag. In each case, the contents of the referenced element will be copied into the Python dictionary.

The ‘units’ field is a bit more interesting. The value here, `‘.//logical-interface/name’`, is an XPath expression that finds the <name> element in each <logical-interface> element within the current <physical-interface> element. This will return None (null) if there are no logical-interfaces on this physical-interface, a single name (as a string) if there is only one logical-interface, or a list (array) of names if there are multiple logical-interfaces. Our program requires only that we determine if the physical interface has logical interfaces, so any value other than None is an affirmative for our purposes.

Making the view and table accessible to our Python code requires a few steps. First, the program imports PyEZ’s FactoryLoader class:

```
from jnpr.junos.factory import FactoryLoader
```

Then the program creates an instance of FactoryLoader, uses it to load the table and view definitions, and assigns this object to the variable `eth_terse`:

```
    fl = FactoryLoader()
    eth_terse = fl.load(table_view)
```

Next, the program associates the table *EthPortTerseTable* from the loaded definition with the PyEZ Device instance, `dev`, and assigns this to the variable that will hold the table's data, `eth_table`:

```
    eth_table = eth_terse[‘EthPortTerseTable’](dev)
```

Finally, the program calls the table object's `get()` method to cause the table to retrieve the requested data from the device and store it:

```
    eth_table.get()’
```

This table variable `eth_table` stores the data from the device in Python data structures that can be read by the program:

```
for eth_int in eth_table:
    if eth_int[‘oper’] == ‘up’:
        link_interfaces.append(eth_int[‘name’])
    else:
        no_link_interfaces.append(eth_int[‘name’])

    if (eth_int[‘units’] is not None) or (eth_int[‘oper’] == ‘up’):
        used_interfaces.append(eth_int[‘name’])
    else:
        unused_interfaces.append(eth_int[‘name’])
```

The *unused_interfaces* variable contains the list of interfaces that have subinterfaces and are `link-down`. The reader may wonder why we need the other three variables, *link_interfaces*, *no_link_interfaces*, and *used_interfaces*. For this recipe, we do not need them. However, the author calls this module from another Ansible playbook that disables interfaces based strictly on link state – the elements of the *nolink_interfaces* variable – and he found it helpful to have *used_interfaces* and *link_interfaces* during testing to ensure all Ethernet interfaces had been accounted for.

Using AnsibleModule

When an Ansible playbook calls a module, Ansible needs a way to pass arguments to the module, and the module needs a way to pass results back to Ansible. The AnsibleModule class provides an interface for this communication.

Start by importing the class from the Ansible library:

```
from ansible.module_utils.basic import AnsibleModule
```

Declare an instance of the class, initializing it with a dictionary of arguments to be passed from Ansible to the module and any other needed arguments:

```
module = AnsibleModule(
   argument_spec=dict(
      host=dict(required=True),
      user=dict(required=False, default=os.getenv(‘USER’)),
      passwd=dict(required=False, default=None, no_log=True),
      port=dict(required=False, type=’int’, default=830)
      ),
   supports_check_mode=False
   )
```

Ansible’s “check mode” allows a playbook to call a module in a “test mode” where the module returns results without doing the normal work expected of the module. The author chose not to support check mode for this module. The argument, `supports_check_mode=False`, lets AnsibleModule know to return an error should the Ansible playbook try to call the module in check mode.

The argument, `argument_spec=dict(…)`, contains the dictionary describing the module’s arguments. Ansible will pass a dictionary to the module containing the arguments provided by the playbook, and the keys in that dictionary should be a subset of the keys defined here. AnsibleModule does error checking – unknown arguments, or required but missing arguments, will cause AnsibleModule to return an error.

We can define whether an argument is required, such as *host*, or optional, such as *passwd*. For optional arguments, you can define a default value that will be used by the module when no value is provided by the playbook. AnsibleModule assumes arguments are strings; use the *type*= value to require a different data type, as with the *port* argument, which requires an integer. Finally, if you do not want an argument’s data, such as a password, included in Ansible’s logs, such as a password, include `no_log=True`.

Each argument, or default value, is placed in the *params* dictionary of the AnsibleModule instance variable *module*. To read the value of an argument, read the appropriate dictionary entry by key; for example, `module.params[‘host’]`.

While not strictly necessary, it’s convenient to copy the arguments into local variables:

```
host = module.params[‘host’]
username = module.params[‘user’]
password = module.params[‘passwd’]
ncport = module.params[‘port’]
```

That takes care of the Ansible playbook passing arguments to the module. Now how does the module return results to the calling playbook?

The *module* instance variable has two instance methods for exiting the module and returning data to the calling playbook. The *fail_json()* method allows the module to indicate it encountered an error (the playbook output will show the module results with a ‘fatal’ status) and return an error message back to Ansible. There are several examples of this in the module, such as the *except* portion of the following code, which will be called should the module be unable to open a connection to the Junos device:

```
try:
   dev = Device(host=host, gather_facts=False, user=username,
              passwd=password, port=ncport)
   dev.open()
except Exception as err:
   msg = ‘Error opening device connection: %s’ % str(err)
   module.fail_json(msg=msg)
```

The other instance method is *exit_json(),* which returns an ‘ok’ or ‘changed’ status back to Ansible based on the value of the *changed*= argument, along with a message and any other data the module wishes. There is one example in our module: after the module has successfully processed the interface data and created the interface lists that will be returned to the playbook, the program puts those interface lists into a dictionary called *result* and returns that dictionary to the playbook:

```
result = {‘unused’: unused_interfaces,
        ‘used’: used_interfaces,
        ‘link’: link_interfaces,
        ‘no_link’: no_link_interfaces}
module.exit_json(changed=False, msg=’Interface lists’, interfaces=result)
```

Because our module cannot change the Junos device it sets *changed=False* (the playbook output will show the module results as ‘ok’). The *msg*= argument returns a status message. Other arguments, such as our *interfaces=result* argument, are returned to the Ansible playbook as part of the dictionary of results returned by the module to the playbook; this allows the module to return arbitrary data.

As the names of these methods suggest, they return JSON-formatted data. You can see this if you review the output of the playbook when run in verbose mode. While not nicely formatted for human consumption, this is recognizably JSON:

```
TASK [get unused interfaces] ***************************************************
ok: [bilbo] => {“changed”: false, “interfaces”: {“link”: [“ge-0/0/11”], “no_link”: [“ge-0/0/0”,
“ge-0/0/1”, “ge-0/0/2”, “ge-0/0/3”, “ge-0/0/4”, “ge-0/0/5”, “ge-0/0/6”, “ge-0/0/7”, “ge-0/0/8”,
“ge-0/0/9”, “ge-0/0/10”, “ge-0/1/0”, “ge-0/1/1”], “unused”: [“ge-0/0/0”, “ge-0/0/1”, “ge-0/0/2”,
“ge-0/0/3”, “ge-0/0/4”, “ge-0/0/5”, “ge-0/0/6”, “ge-0/0/7”, “ge-0/1/0”, “ge-0/1/1”], “used”: [“ge-
0/0/8”, “ge-0/0/9”, “ge-0/0/10”, “ge-0/0/11”]}, “msg”: “Interface lists”}
```

Recipe 22 - Track Down IP Conflicts with PyEZ

by Matt Mellin

■ Python Version Used:	3.6.2
■ PyEZ Version Used:	2.1.5
■ Junos OS Used: (MX)	13.3R8.7, (EX) 12.3R11.2, (QFX) 14.1X53-D30.3

- Juniper Platforms General Applicability: MX, EX, QFX

This recipe utilizes some advanced Python features such as multi-core processing for concurrent compute and recursive functions, along with extensive use of PyEZ Tables and Views. It also uses LLDP for device auto-discovery. And, by the way, it assumes the network is only using IPv4.

Problem

In a large, dynamic lab network, especially one without DNS/IPAM/DHCP, IP conflicts can be a real pain. Which devices are misconfigured? Where are they located? Duplicate IP addressing can mean loss of productivity, unauthorized accidental configuration changes, and loss of connectivity. These types of issues can often occur when you are moving around routing engines (REs) and re-configuring systems.

It's a problem in Juniper's Proof-of-Concept (POC). With over 125 racks and 2000+ interfaces, the POC at Juniper's headquarters is fairly sizeable. The lab houses many varieties of Junos devices, unfortunately, not all of which have DHCP capability. DNS and IPAM were not a perfect fit either, so for many reasons, static IP addressing was historically applied.

In order to create custom network topologies for customers, gear is moved around and IP addressing can sometimes be duplicated from device to device, causing major troubleshooting issues, project delays, and general confusion. An RE in an MX960 today could be used in an MX240 tomorrow, and when configurations get copied, or pushed via management tools, IP duplications can occur. When they do, however, where do you look? With over 2000+ top of rack (TOR) ports, manually checking MAC Tables was slow and cumbersome.

These issues only become worse because of the time sensitive nature of these projects. While problems didn't happen too frequently, they occurred often enough to warrant an automated solution.

So the exact problem this recipe, as illustrated in Figure 22.1, aims to solve is helping to identify *which physical interfaces, within the lab's switching infrastructure, are connected to potential misconfigured devices that are using the same IP address*.

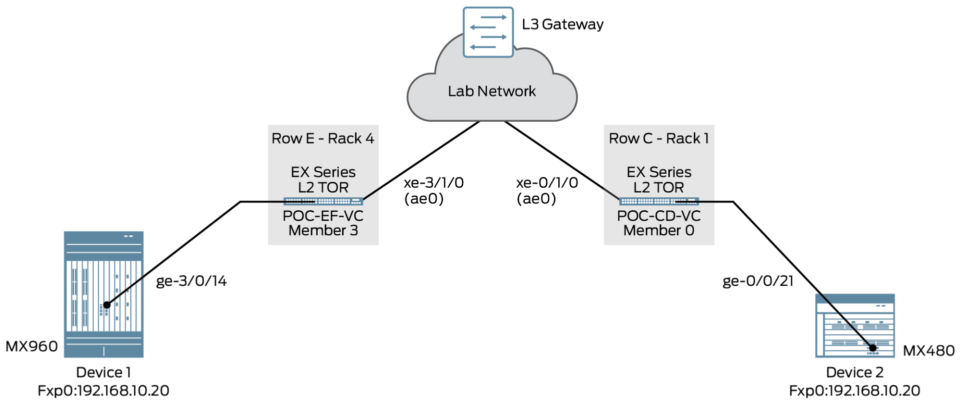


Figure 22.1 Duplicate IP Addressing Illustrated

The networking topology shown in the next illustration, Figure 22.2, roughly relates to that of the POC lab in question. There are multiple layers of virtual chassis switches connecting various racks full of servers, and physical Junos gear for use in customer-facing pre-sales proof of concepts. Note it uses Layer 2 bridging on the MX gateway layer to provide fault tolerance and redundancy on the network.

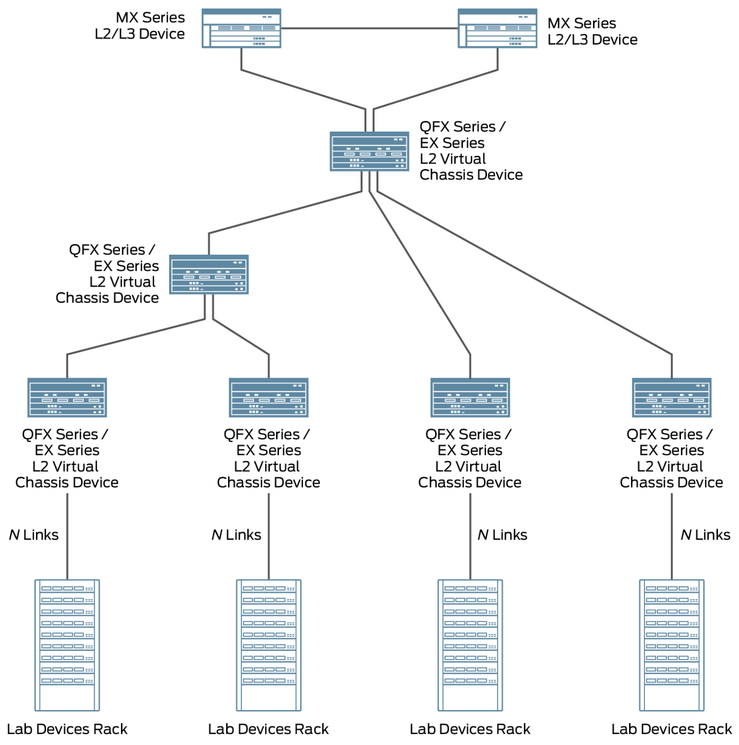


Figure 22.2 Sample Hierarchical Lab Network Topology

NOTE Layer 3 interfaces within Layer-2 bridges (VLANs) are called IRB interfaces, meaning the recipe solution needs to look not only through the Ethernet MAC Tables of the EX Series and QFX Series switches, but also through the MX Series devices.

Solution

Create 'duplicate_ips' log file

In order to search for duplicate IP addresses, you must first know about them! To find a list of potential offending IP addresses to search for, we looked through the MX gateway's log file. When the MAC addressing associated with an IP address changes, the following message is logged via syslog:

```
Aug  4 07:34:44  POC-CORE-MX1 /kernel: KERN_ARP_ADDR_CHANGE: arp info overwritten for 192.168.1.10 from
00:18:8d:2c:0b:4e to 0c:c4:8a:60:d1:ba
```

The first step is to create a log file on the gateway that matches just these logs. This is done in order for the program to have raw data from which to find potential duplicate IP addresses and their associated MACs. Later, Python will parse through the file and create a data structure in order to start the search.

```
root@POC-CORE-MX1> show configuration groups duplicate_ip
system {
   syslog {
      file duplicate_ips {
         kernel any;
         match KERN_ARP_ADDR_CHANGE;
         archive size 655536 files 10;
      }
   }
}
```

NOTE By default, log files live under the `/var/log` directory on the Junos OS router.

Python Code

All of the following steps are taken care of via Python.

1. Using PyEZ log into the gateway MX router and download the “duplicate_ips” log file.

2. Parse this log file to get a list of IP addresses and associated MAC addresses.

3. Then use PyEZ to look through the gateway router’s route table and discover the interfaces and VLANs on which the MAC addresses were learned.

4. With this information, you can begin to scan the LLDP table for neighboring Layer 2 switching devices.

5. Once found, using PyEZ, you can recursively log into these devices and initiate a search for MAC addressing and associated learned interfaces within the scope of these VLANs based on the data from prior searches.

6. Using this technique, you will both discover and search through the tree of lab infrastructure devices, attempting to find physical interfaces directly connected to misconfigured devices.

7. In order to increase the speed of the search, you can employ multiprocessing techniques to run scans of devices concurrently. While Python is not a truly parallel programming language by any means, it does have some capability to take advantage of a system with multiple cores and run processes and threads

concurrently.

The following sections discuss the various components of the application. To understand how to install, set up, and use it, see this recipe's *Discussion* section, or read through the README at https://github.com/mmellin/findDuplicateIp.

```
'''
Identify physical switch interfaces associated within an IP conflict.

This program finds IP conflicts within a JUNOS based L3/L3 tiered network, and tracks down which access
interfaces the MAC addresses are coming from.
Copyright (c) 2017 Matthew Mellin

This library is free software.  It is licensed under the Apache License version 2.0.

The manuf.py library is copyright (c) 2017 Michael Huang and licensed under the terms of the GNU Lesser
General Public
License version 3.0 (or any later version) and the Apache License version 2.0. <https://github.com/
coolbho3k/manuf>

For more information, see:

<http://www.gnu.org/licenses/>
<http://www.apache.org/licenses/>

'''
import argparse
import getpass
import json
import os
import socket
import sys
from pathlib import Path
from collections import defaultdict
from pkgs.utils.log import get_logger
from pkgs.utils import ipconflictslib as ipc
```

The first thing to do coding-wise is to import Python modules for use within our program. These are either importing the functionality from the standard library or from our own or third-party modules located in the `pkgs/` package. Functions called from our `pkgs.utils.ipconflictslib` will be namespaced to `ipc.<function_name>`.

```
if __name__ == '__main__':
   main()
```

The `main()` function will be executed when the file `main.py` is run from the terminal:

```
def main():
   # Script Arguments
   args = create_help()
   seed_router = args.seed_router
   username = args.user
   password = None
   auth_mode = args.mode
   dup_filename = args.file
   router_log_path = args.router_log_path
```

```
    local_log_dir = "logs/"
    connect_timeout = args.connect_timeout
    resolve_vendor = args.vendor
    update_vendor = args.update_vendor
    processes = args.processes

    if auth_mode == 'password':
        password = getpass.getpass()
```

Let's begin by calling the create_help() function to create our help menu, take care of our program arguments, and initialize our argument variables. Use the `getpass` package to hide our password as we enter it, so that it is not stored in our system's history. By default, the program supports connecting into the Junos OS device via username and password, but additional code could be created to support SSH keys. This is why we have the `auth_mode` variable, but let's leave this as a future enhancement:

```
def create_help():
    """ Build help documentation and creates program arguments."""
    parser = argparse.ArgumentParser()

    parser.add_argument("seed_router", type=str,
                        help="Gateway router for hosts to begin search.  Hostname or IP.")
    parser.add_argument("-u", "--user", type=str, help="Seed_router username for login.")
    # parser.add_argument("-p", "--password", type=str, help="Seed_router login password.")
    parser.add_argument("-m", "--mode", type=str, default="password",
                        help="Method of login: 'ssh-key' or 'password'. Defaults to password.")
    parser.add_argument("-f", "--file", type=str, default="duplicate_ips",
                        help="Name of duplicate IP log file on router.")
    parser.add_argument("--connect_timeout", type=int, help="PyEz device connect timeout")
    parser.add_argument("--processes", type=int, help="Max number of threads to run concurrent.")
    parser.add_argument("--vendor", action='store_true', help="Try to resolve the HW vendor for each 
MAC.")
    parser.add_argument("--update_vendor", action='store_true',
                        help="Update local copy of Wireshark's OUI DB from the web.")
    parser.add_argument("--router_log_path", type=str, default='/var/log/',
                        help="Folder path on router where the 'duplicate_ips' log is stored. Defaults to 
'var/log/")
    args = parser.parse_args()

    return args
```

You can see the `create_help()` function that `main()` calls to build the help menu and ingest arguments. It returns a simple namespace object that acts like a dictionary so `main()` can assign variables in the `username = args.user` fashion. This functionality is built using the standard `argparse.py` library:

```
# Initialize variables
credentials = {'username': username, 'password': password, 'auth_mode': auth_mode}
base_dir = os.path.dirname(os.path.realpath(__file__))
local_log_abspath = os.path.join(base_dir, local_log_dir)
data_dir_abspath = os.path.join(base_dir, 'data')
vlan_id_name_dict = {}
vlans = defaultdict(list)
remote_systems = defaultdict(dict)
```

```
seed_ip = socket.gethostbyname(seed_router)
manuf_file_path = Path(os.path.join(data_dir_abspath, "manuf"))
updated = False
```

Now let's initialize variables used within main.py. First, let the program know about your base directory (the absolute path from where you are running main.py), and some directories relevant to your program such as `logs/` and `data/`. The `VLANs` variable is a dictionary-like data structure to house the names and tags of VLANs you find when initially searching the gateway router. The `remote_systems` variable is a dictionary-like data structure to house all the information you find about the systems scanned.

In addition to these, there is an `updated` variable that lets you know whether or not the Wireshark OUI text file has been updated or not, used for MAC-to-hardware vendor resolution.

Finally, there are the `credentials`, which are used for device connectivity, and `seed_ip`, which is the IP address of the gateway host. Here, we use a socket library to resolve any potential DNS hostname entry from the command-line-arguments into an IP:

```
# Attempt to update the Wireshark OUI directory if vendor resolution is required.
if resolve_vendor:
   # Verify that a Wireshark OUI file exists
   if not manuf_file_path.is_file():
      logger.info("OUI resolution file does not exist.")
      logger.info("Downloading and updating '{}' Wireshark OUI
                  file.".format(manuf_file_path.as_posix()))
      try:
         ipc.update_vendor_file(path=manuf_file_path)
         logger.info("Download successful.")
         updated = True
      except Exception as e:
         logger.error(e)
         logger.error("Download failed. MAC resolution disabled.")
         resolve_vendor = False
```

Since the program supports MAC OUI vendor resolution via a nice library called `Manuf` (https://github.com/coolbho3k/manuf), add in the following code to check the vendor resolution-related arguments. The data this library works with is a text file, called `Manuf`, downloaded from the Wireshark online site. Put this data file under your `data/` folder.

Since this file comes from the Web, it may be updated periodically. You can easily run the program without updating each time, which is faster, but sometimes you'll want to update this file. In fact, this file is not included in the initial clone of the repository, so the first time you run this program with the `–vendor` flag, it will automatically call for an update to download this file to your localhost.

In our code above, we checked for the existence of the `Manuf` file. If it is not found, it uses `update_vendor_file()` to download it from the Web, then set the `updated` variable to `True`:

```
# Attempt to update the OUI file if it hasn't been already.
if update_vendor:
   # If we've already updated, skip updating again.
   if updated:
      pass
   else:
      logger.info("Downloading and updating '{}' Wireshark OUI 
                    file.".format(manuf_file_path.as_posix()))
      try:
         ipc.update_vendor_file(path=manuf_file_path)
         logger.info("Download successful.")
         updated = True
      except URLError as e:
         logger.error("Download failed.")
```

If the `–update` flag is used as an argument, it first checks to see if the `Manuf` file has already been updated. If it has, it skip the update, otherwise it uses `update_vendor_file()` to download the file to the localhost:

```
# Begin print log output header
logger.info("#" * 60)
logger.info("Locating Duplicate IPs in your Network")
logger.info("#" * 60)
logger.info('\n')
logger.info("Opening a connection to the Gateway Router.\n")

# Begin connecting to network and finding baseline information
with ipc.connect_dut(seed_router, credentials, connect_timeout=connect_timeout) as dev:
   personality = dev.facts.get('personality')
   model = dev.facts.get('model')
   name = dev.facts.get('hostname')
   description = model + " " + dev.facts.get('serialnumber')
   remote_systems[seed_ip] = {'facts': {'personality': personality, 'model': model}, 'name': 
                                    name, 'description': description, 'searched': False}

   """ Step 1 - Parse the log file and identify Potential Duplicate IPs """
   logfile = ipc.LogFile(dup_filename, local_log_abspath, router_log_path)
   logger.info('Copying log file {} to local disk...'.format(logfile.filename))

   seed_dict = logfile.create_seed_dict(dev)

   logger.debug("{} -> {}".format(logfile.remote_file_path, logfile.local_file_path))
   logger.debug("Creating the 'seed_dict' data structure for parsing.")
   logger.info("Log file copied, starting to parse...")
   logger.debug(json.dumps(seed_dict, indent=2))
   logger.info("Log file parsed.  Starting to find mappings for IP, Vlan and MAC.")

   # Validate that you have any potential duplicate IP logs, else exit.
   for item in seed_dict.values():
      if len(item) == 0:
         logger.info("No duplicate IP entries found in the provided Duplicate IP log. 
                        Exiting.")
         sys.exit(1)
```

Let's start to output information to the terminal window and into the local log file, then connect to the gateway router.

The next step to understanding where duplicate IP addressing is located in the network is to determine which IP addresses and MAC addresses to scan for, and on which VLANs. We begin this process by using a custom function called `connect_dut` from our `ipconflictslib` module to connect and log in to the gateway MX Series device.

Once we set up some variables with the device facts, instantiate a `LogFile` object, which is a class from the `ipconflictslib` module. Then call the object's `create_seed_dict` method to open the file and return a list of lines from our router's `duplicate_ips` log file to parse.

NOTE Throughout the program, a logger is used to output both to the `stdout` and to a log file. You'll want to use this method for your scripts as well. See *Logging* under the *Discussion* section of this recipe for further details.

```
""" Step 2 – Determine which VLANs and interfaces the IPs were learned over """
   iter1 = ipc.determine_VLAN(dev, seed_dict)
   for data_tuple in iter1:
      VLAN_name = data_tuple[0]
      VLAN_id = data_tuple[1]
      ip = data_tuple[2]
      macs = data_tuple[3]
      logger.info("Potential duplicate IP {} found on VLAN {}({})".format(ip, VLAN_name,
                                                                           VLAN_id))
      VLAN_id_name_dict[VLAN_id] = VLAN_name
      VLANs[VLAN_id].extend([mac for mac in macs if mac not in VLANs[VLAN_id]])

   remote_systems[seed_ip]['VLANs'] = VLANs
   logger.debug(VLAN_id_name_dict)
```

Begin the second processing step while still connected to the MX Series gateway, to determine over which VLANs and physical interfaces the MAC addresses were learned. We will use this information to scan for neighboring devices, and when we find one and can connect to it, we'll use the VLAN as a filter when scanning the Ethernet table.

The seemingly simple function `determine_VLAN` in our `ipconflictslib` package accepts the current devices connection, as well as the data gleamed from the log file, and returns a list of tuples. These tuples include the VLAN's name and tag, along with an IP address and the MAC addressing associated with this MAC address. This return data is considered "flat" because there are many repetitive lines, and each line has the same set of information. Flat structures take up more memory but can be faster to parse, group, or run joins on than a dictionary.

With that step completed, we have matched our potential duplicate IP addresses up to MACs, and organized this data by VLANs. Print out what has been found, and add the VLAN name and tag information into its own dictionary, for use later, as well as to the `remote_systems` dictionary, where it is matched to the IP address of the currently scanned device:

```
logger.info("Begin to scan the systems found attached to the Seed Router.\n")
scanned_systems, all_macs_found = ipc.system_scan(remote_systems, credentials, processes,
                                              connect_timeout=connect_timeout)
logger.debug(json.dumps(scanned_systems, indent=2))
logger.info("Search complete.")

output = ipc.create_output_structures(scanned_systems, seed_dict, resolve_vendor=resolve_vendor)
logger.debug(json.dumps(output, indent=2))
```

The first system is now added into the `remote_systems` dictionary and it's time to begin the search. Another seemingly small bit of Python, a call to our recursive function `system_scan`, hides a lot of code. It's covered in more detail shortly, in a section covering `ipconflictslib.py`. It's in charge of expanding the size of the remote_ systems dictionary by scanning and searching all systems attached to this first device, then repeating the search for all systems connected to those, and so on.

What it outputs is a new version of `remote_systems` with all the information you need to determine where potential duplicate IP addresses live, along with information about all the systems it has come in contact with and how they are connected to each other.

This new data structure has a bunch of information, but isn't in the best format to easily output on the screen, so let's parse it and filter out the information you need about the leaf nodes, the devices at the end of the search connected to misconfigured systems, and the IP and interface information. This is done via a call to `create_output_structures` from `ipconflictslib`. This is also the function that calls the `Manuf.py` library for the MAC OUI vendor resolution:

```
print("\n\n")
logger.info("FINAL OUTPUT: \n")
for vlan_id, ip_values in output.items():
    vlan_name = vlan_id_name_dict[vlan_id]
    logger.info("Vlan {}: {}".format(vlan_name, vlan_id))
    for ip, tuples in ip_values.items():
        logger.info("\tIP {} found on:".format(ip))
        for data in tuples:
            mac = data[0]
            vendor = data[1]
            name = data[2]
            mgt_ip = data[3]
            if not mgt_ip:
                if not vendor:
                    logger.info("\t\t{} -> {} (MAC not found in search) ".format(mac, 'System
                                      name not found'))
                else:
                    logger.info("\t\t{}({}) -> {} (MAC not found in search) ".format(mac, vendor,
                                    'System name not found'))
            else:
                interface = data[4]
            if not vendor:
               logger.info("\t\t{} -> {}: {} (mgt IP: {})".format(mac, name, interface,
                                                                   mgt_ip))
            else:
               logger.info("\t\t{}({}) -> {}: {} (mgt IP: {})".format(mac, vendor, name,
                                                                       interface,
mgt_ip))
```

Finally, print out the findings. The data structure `output` returned from our last function has everything in place to make it easy for you to do just that. You organize the output by VLAN, then by IP address, printing out the MACs associated with the IP and any remote systems and interface naming attached. Also included is the MAC's hardware vendor if that option was selected.

If a MAC found in the log file is no longer active on the network, then we pad that output and let the user know that we couldn't find it – it could be because the misconfigured device with that IP address is offline, like a routing engine or VM, or that the problem was fixed.

Sample Output

```
(env3)$ python main.py 192.168.1.254 -u user1 --vendor
Password:
2017/08/24 21:36:09 __main__ 95 - INFO: ##############################################################
2017/08/24 21:36:09 __main__ 96 - INFO: Locating Duplicate IPs in your Network
2017/08/24 21:36:09 __main__ 97 - INFO: ##############################################################
2017/08/24 21:36:09 __main__ 98 - INFO:

2017/08/24 21:36:09 __main__ 99 - INFO: Opening a connection to the Gateway Router.

2017/08/24 21:36:12 pkgs.utils.ipconflictslib 48 - INFO: Connection Successful (POC-GW-MX1 - 
192.168.1.254).
2017/08/24 21:36:13 __main__ 112 - INFO: Copying log file duplicate_ips.0.gz to local disk...
2017/08/24 21:36:14 __main__ 118 - INFO: Log file copied, starting to parse...
2017/08/24 21:36:14 __main__ 120 - INFO: Log file parsed.  Starting to find mappings for IP, VLAN and MAC.
2017/08/24 21:36:15 __main__ 135 - INFO: Potential duplicate IP 192.168.1.7 found on VLAN V100(100)
2017/08/24 21:36:16 __main__ 135 - INFO: Potential duplicate IP 172.12.10.55 found on VLAN COMMON-
SERVICES(16)
2017/08/24 21:36:17 __main__ 135 - INFO: Potential duplicate IP 172.12.10.56 found on VLAN COMMON-
SERVICES(16)
2017/08/24 21:36:18 __main__ 135 - INFO: Potential duplicate IP 172.12.33.44 found on VLAN DEVICE-
FXP0(18)
2017/08/24 21:36:19 __main__ 135 - INFO: Potential duplicate IP 172.12.33.173 found on VLAN DEVICE-
FXP0(18)
2017/08/24 21:36:20 __main__ 135 - INFO: Potential duplicate IP 172.12.32.196 found on VLAN DEVICE-
FXP0(18)
2017/08/24 21:36:21 __main__ 135 - INFO: Potential duplicate IP 172.12.34.133 found on VLAN DEVICE-
FXP0(18)
2017/08/24 21:36:22 __main__ 135 - INFO: Potential duplicate IP 172.12.34.134 found on VLAN DEVICE-
FXP0(18)
2017/08/24 21:36:23 __main__ 135 - INFO: Potential duplicate IP 172.12.32.154 found on VLAN DEVICE-
FXP0(18)
2017/08/24 21:36:24 __main__ 135 - INFO: Potential duplicate IP 172.12.32.86 found on VLAN DEVICE-
FXP0(18)
2017/08/24 21:36:25 __main__ 135 - INFO: Potential duplicate IP 172.12.33.13 found on VLAN DEVICE-
FXP0(18)
```

```
2017/08/24 21:36:26 __main__ 135 - INFO: Potential duplicate IP 172.12.33.7 found on VLAN DEVICE-
FXP0(18)
2017/08/24 21:36:26 pkgs.utils.ipconflictslib 60 - INFO: Closed connection to 192.168.1.254
2017/08/24 21:36:26 __main__ 142 - INFO: Begin to scan the systems found attached to the Seed Router.

2017/08/24 21:36:26 pkgs.utils.ipconflictslib 360 - INFO: #### Searching System 192.168.1.254
2017/08/24 21:36:29 pkgs.utils.ipconflictslib 48 - INFO: Connection Successful (POC-GW-MX1 -
192.168.1.254).
2017/08/24 21:36:29 pkgs.utils.ipconflictslib 428 - WARNING: No MACs found while searching VLAN 100 on
device 192.168.1.254
2017/08/24 21:36:34 pkgs.utils.ipconflictslib 60 - INFO: Closed connection to 192.168.1.254
2017/08/24 21:36:34 pkgs.utils.ipconflictslib 360 - INFO: #### Searching System 192.168.1.5
2017/08/24 21:36:38 pkgs.utils.ipconflictslib 48 - INFO: Connection Successful (POC-CORE-EXVC -
192.168.1.5).
2017/08/24 21:36:41 pkgs.utils.ipconflictslib 60 - INFO: Closed connection to 192.168.1.5
2017/08/24 21:36:41 pkgs.utils.ipconflictslib 360 - INFO: #### Searching System 192.168.1.15
2017/08/24 21:36:41 pkgs.utils.ipconflictslib 360 - INFO: #### Searching System 192.168.1.22
2017/08/24 21:36:41 pkgs.utils.ipconflictslib 360 - INFO: #### Searching System 192.168.1.14
2017/08/24 21:36:41 pkgs.utils.ipconflictslib 360 - INFO: #### Searching System 192.168.1.16
2017/08/24 21:36:49 pkgs.utils.ipconflictslib 48 - INFO: Connection Successful (POC-EX-VC-A -
192.168.1.16).
2017/08/24 21:36:50 pkgs.utils.ipconflictslib 48 - INFO: Connection Successful (POC-EX-VC-B -
192.168.1.14).
2017/08/24 21:36:50 pkgs.utils.ipconflictslib 60 - INFO: Closed connection to 192.168.1.16
2017/08/24 21:36:50 pkgs.utils.ipconflictslib 360 - INFO: #### Searching System 192.168.1.13
2017/08/24 21:36:51 pkgs.utils.ipconflictslib 60 - INFO: Closed connection to 192.168.1.14
2017/08/24 21:36:51 pkgs.utils.ipconflictslib 48 - INFO: Connection Successful (CS-DATA-VC1-0 -
192.168.1.22).
2017/08/24 21:36:51 pkgs.utils.ipconflictslib 48 - INFO: Connection Successful (POC-EX-VC-D -
192.168.1.15).
2017/08/24 21:36:51 pkgs.utils.ipconflictslib 60 - INFO: Closed connection to 192.168.1.22
2017/08/24 21:36:52 pkgs.utils.ipconflictslib 60 - INFO: Closed connection to 192.168.1.15
2017/08/24 21:36:57 pkgs.utils.ipconflictslib 48 - INFO: Connection Successful (POC-EX-VC-C -
192.168.1.13).
2017/08/24 21:36:58 pkgs.utils.ipconflictslib 60 - INFO: Closed connection to 192.168.1.13
2017/08/24 21:36:58 pkgs.utils.ipconflictslib 360 - INFO: #### Searching System 172.12.34.107
2017/08/24 21:36:58 pkgs.utils.ipconflictslib 360 - INFO: #### Searching System 172.12.39.18
2017/08/24 21:36:58 pkgs.utils.ipconflictslib 360 - INFO: #### Searching System 172.12.32.86
2017/08/24 21:36:58 pkgs.utils.ipconflictslib 55 - ERROR: Failed to connect to 172.12.39.18.
 ConnectRefusedError(172.12.39.18)
2017/08/24 21:36:59 pkgs.utils.ipconflictslib 52 - ERROR: Authentication failed to 172.12.32.86.
 ConnectAuthError(172.12.32.86)
2017/08/24 21:36:59 pkgs.utils.ipconflictslib 52 - ERROR: Authentication failed to 172.12.34.107.
 ConnectAuthError(172.12.34.107)
2017/08/24 21:36:59 __main__ 145 - INFO: Search complete.



2017/08/24 21:37:01 __main__ 151 - INFO: FINAL OUTPUT:

2017/08/24 21:37:01 __main__ 154 - INFO: VLAN COMMON-SERVICES: 16
2017/08/24 21:37:01 __main__ 156 - INFO:      IP 172.12.10.55 found on:
2017/08/24 21:37:01 __main__ 166 - INFO:            0c:c4:7a:68:42:56(SuperMic) -> CS-DATA-VC1-0:
xe-2/0/40.0 (mgt IP: 192.168.1.22)
2017/08/24 21:37:01 __main__ 163 - INFO:            0c:c4:7a:68:42:57(SuperMic) -> System name not
found (MAC not found in search)
```

```
2017/08/24 21:37:01 __main__ 154 - INFO: VLAN DEVICE-FXP0: 18
2017/08/24 21:37:01 __main__ 156 - INFO:      IP 172.12.33.13 found on:
2017/08/24 21:37:01 __main__ 166 - INFO:            00:a0:a5:90:1d:31(TeknorMi) -> POC-EX-VC-E:
ge-0/0/44.0 (mgt IP: 192.168.1.14)
2017/08/24 21:37:01 __main__ 166 - INFO:            50:c5:8d:ab:80:92(JuniperN) -> POC-EX-VC-C:
ge-0/0/0.0 (mgt IP: 192.168.1.13)
2017/08/24 21:37:01 __main__ 156 - INFO:      IP 172.12.32.154 found on:
2017/08/24 21:37:01 __main__ 166 - INFO:            40:b4:f0:79:57:ff(JuniperN) -> POC-EX-VC-A:
ge-4/0/29.0 (mgt IP: 192.168.1.16)
2017/08/24 21:37:01 __main__ 163 - INFO:            40:b4:f0:79:57:c1(JuniperN) -> System name not
found (MAC not found in search)
2017/08/24 21:37:01 __main__ 156 - INFO:      IP 172.12.34.133 found on:
2017/08/24 21:37:01 __main__ 166 - INFO:            e8:b6:c2:84:34:68(JuniperN) -> POC-EX-VC-A:
ge-4/0/24.0 (mgt IP: 192.168.1.16)
2017/08/24 21:37:01 __main__ 166 - INFO:            e8:b6:c2:84:35:68(JuniperN) -> POC-EX-VC-A:
ge-4/0/25.0 (mgt IP: 192.168.1.16)
2017/08/24 21:37:01 __main__ 156 - INFO:      IP 172.12.34.134 found on:
2017/08/24 21:37:01 __main__ 166 - INFO:            e8:b6:c2:84:34:68(JuniperN) -> POC-EX-VC-A:
ge-4/0/24.0 (mgt IP: 192.168.1.16)
2017/08/24 21:37:01 __main__ 166 - INFO:            e8:b6:c2:84:35:68(JuniperN) -> POC-EX-VC-A:
ge-4/0/25.0 (mgt IP: 192.168.1.16)
2017/08/24 21:37:01 __main__ 156 - INFO:      IP 172.12.32.86 found on:
2017/08/24 21:37:01 __main__ 166 - INFO:            00:a0:a5:84:ae:5f(TeknorMi) -> POC-EX-VC-D:
ge-5/0/4.0 (mgt IP: 192.168.1.15)
2017/08/24 21:37:01 __main__ 166 - INFO:            00:a0:a5:84:ae:5f(TeknorMi) -> POC-EX-VC-D:
ge-5/0/4.0 (mgt IP: 192.168.1.15)
```

Supporting Package Code

This section walks you through the Python code that parses the log file, handles device discovery, and creates data about the network for `main.py` to output.

ipconflictslib.py

The module begins with import statements as usual:

```
“”” Module that contains code for finding IP conflicts “””
import gzip
import magic
import os
from collections import defaultdict
from contextlib import contextmanager
from multiprocessing import Pool
from functools import partial
from pathlib import Path
from pkgs.utils.log import get_logger
from pkgs.utils import manuf
from jnpr.junos import Device
from jnpr.junos.utils.scp import SCP
from jnpr.junos.exception import (ConnectAuthError, ConnectError, ConnectTimeoutError)
from jnpr.junos.factory import loadyaml
```

From PyEZ, make sure to import the components you know you’ll be using, such as Device and the SCP utility, along with some exceptions that these raise in the event of failure. In addition, you need the `loadyaml()` function to turn the Table/

View YAML file into functions within the modules namespace. Make sure to import the get_logger() function to get logging functionality.

```
# Setup logger
logger = get_logger(__name__)

base_dir = os.path.dirname(os.path.dirname(os.path.dirname(os.path.realpath(__file__))))
data_dir = os.path.join(base_dir, 'data')
log_dir = os.path.join(base_dir, 'logs')

# Create classes per table/view and add them to the global namespace
globals().update(loadyaml(os.path.join(data_dir, 'op_table_views.yml')))
```

Next set up the logger by calling the get_logger()function. The argument `__name__` tells logger to use whatever the current namespace is, which here happens to be the name of the module, `ipconflictslib`. Then set up some global variables within the namespace for the location of the directories used within the program. Lastly, load the Tables and Views into the global namespace in order to use them by name within the code.

The first function that main.py calls from `ipconflictslib.py` (line 77 and 88) is `update_vendor_file()`:

```
def update_vendor_file(path=None):
   """Updates the Wireshark OUI text file for use with Manuf.py module.

   Args:
      path (str): Absolute path of the Wireshark MAC OUI Manuf text file.

   Returns:
      None

   """
   if not path:
      path = Path(os.path.join(data_dir, "manuf"))
   try:
      manuf.MacParser(manuf_name=path.as_posix(), update=True)
      logger.info("Update done successfully.")
   except Exception:
      raise
```

This function creates an object from the `MacParser` class within `manuf.py` and updates the Wireshark OUI text file:

```
@contextmanager
def connect_dut(dut, credentials, connect_timeout=None, facts=True):
   """Custom function to connect to device. Wraps PyEZ Device.open() function.

   SSH key support currently not tested.

   Args:
      dut (str): IP or hostname of a JUNOS device with which to establish an SSH session.
      credentials (dict): Contains credential information along with the mode of requested
authentication.
```

```
        connect_timeout (int): PyEZ connection timeout in seconds..
        facts (bool): Allows establishing a PyEZ connection with or without ‘facts’ gathering.

    Returns:
        dev (obj): PyEZ object with open SSH connection to JUNOS device (dut).

    Raises:
        ConnectAuthError: If the authentication fails.
        ConnectError: If there are connectivity problems during connection establishment.

    “””
    user = credentials[‘username’]
    password = credentials[‘password’]
    auth_mode = credentials[‘auth_mode’]
    dev = None
    try:
        if auth_mode == ‘password’ and password:
            if connect_timeout:
                Device.timeout = connect_timeout
            dev = Device(host=dut, user=user, password=password, gather_facts=facts).open()
        elif auth_mode == ‘ssh-key’:
            dev = Device(host=dut, gather_facts=True).open()
        else:
            logger.error(“Proper credentials for device login not provided.”)
        logger.info(‘Connection Successful ({} - {}).’.format(dev.facts.get(‘hostname’), dut))
        yield dev

    except ConnectAuthError as e:
        logger.error(“Authentication failed to {}.\n {}”.format(dut, e))
        raise
    except ConnectTimeoutError as e:
        logger.error(“Connection timeout to {}.\n {}”.format(dut, e))
        raise
    except ConnectError as e:
        logger.error(“Failed to connect to {}.\n {}”.format(dut, e))
        raise
    finally:
        if dev:
            dev.close()
            logger.info(“Closed connection to {}”.format(dut))
```

The next function to be called by `main.py` is `connect_dut()`, which allows you to connect to a Junos device using PyEZ. Wrap the normal method of opening an SSH channel to a Junos device in PyEZ so that you can feed the credentials dictionary as an argument. This way you can choose either user or password authentication, or SSH key-based authentication, in one function.

NOTE The `@contextmanager` is a decorator for this function. Using this decorator from Python’s `contextlib` allows you to use this function in a “with” statement, such as in `main.py` on line 101. In addition to this decorator, you’ll need the function to `yield` a result and have a `finally` statement that is executed when the “when” statement completes:

```
class LogFile:
   “””Object the represents a ‘duplicate_ips’ log file.

   Contains functions to open and parse the log file that was created on the MX gateway router
   to document IP conflicts.

   Attributes:
      filename (str): Name of the IP conflict log file.
      remote_log_dir (str): Absolute path of the remote directory housing the IP conflict log
                            file.
      local_log_dir (str): Absolute path of the local log directory.
      remote_file_path (str): Absolute path of the remote IP conflict log file.
      local_file_path (str): Absolute path of the local IP conflict log file.
      filetype (str): The type of file that is the IP conflict log file.

   “””
   def __init__(self, filename, local_log_dir, remote_log_dir):
      “””

      Args:
         filename (str): Name of the IP conflict log file. Used for both remote and local
                         files.
         local_log_dir (str): Absolute path of the local log directory.
         remote_log_dir (str): Absolute path of the remote directory housing the IP conflict
                               log file.

      Returns:

      “””
      self.filename = filename
      self.remote_log_dir = remote_log_dir
      self.local_log_dir = local_log_dir
      self.remote_file_path = self.create_file_path(self.remote_log_dir, self.filename)
      self.local_file_path = self.create_file_path(self.local_log_dir, self.filename)
      self.filetype = self.determine_type(self.local_file_path)

   @staticmethod
   def create_file_path(directory, filename):
      “””Joins the directory and filename”””
      return os.path.join(directory, filename)

   @staticmethod
   def determine_type(file):
      “””Helps determine the type of input file”””
      mime = magic.Magic(mime=True)
      return mime.from_file(file)

   def copy_log_to_local(self, device):
      “””Copies a remote file to the local device via SCP”””
      try:
         with SCP(device) as scp:
            scp.get(self.remote_file_path, self.local_file_path)
      except Exception as e:
         logger.exception(e)
         raise

   def create_lines(self, file):
      “””Creates a Python list data structure for lines in an input file.
```

```
        Args:
            file (str): Absolute path of a file to open.

        Returns:
            (list): Output from f.readlines()

        “””
        if self.filetype == ‘text/plain’:
            with open(file, ‘r’) as f:
                return f.readlines()
        elif self.filetype == ‘application/x-gzip’:
            with gzip.open(file, ‘rt’) as f:
                return f.readlines()
        else:
            raise UnsupportedError(“Unsupported file type: {}”.format(self.filetype))

    # Main function
    def create_seed_dict(self, dev):
        “””Function that initiates the creation of IP/MAC mappings from the IP conflict log file.

        Args:
            dev (obj): PyEZ JUNOS device with open SSH connection.

        Returns:
            (dict): Output of a call to the generate_seed_dict method with in LogParsers class.
            A LogParsers object is created and the list of lines from the file is fed to a method
                which returns a dictionary with IP/MAC mappings.

        “””
        self.copy_log_to_local(dev)
        lines = self.create_lines(self.local_file_path)
        return LogParsers.generate_seed_dict(lines)
```

After opening a connection to the network’s gateway device, the `LogFile` class is used to create an object that handles the copying over of the log file to the local computer and creates a seed dictionary to begin network search. The class has a few attributes, like `filename`, which come in handy. The main function within this class is `create_seed_dict`, which itself uses a separate class and returns a dictionary structure that will ultimately be our seed dictionary. These two roles are broken into separate functions so the code is easier to test, and grouped by function:

```
class LogParsers:
    “”” Parses the log file and finds all IP addresses and the associated MACs.
    This code works for unstructure syslog only.

    Sample log output:
    Jun 12 23:26:53  My-MX1 /kernel: KERN_ARP_ADDR_CHANGE: arp info overwritten for 192.168.1.10
    from 00:11:7d:1f:0b:3e to 0c:c5:7a:52:d1:aa
    “””

    @staticmethod
    def generate_seed_dict(lines):
        “””Create a dictionary that maps IP addresses to MAC and vice versa.

        Args:
            lines (list): List of lines form IP conflict file.
        Returns:
           seed_dict (dict): A mapping between IP and MAC addressing.

        “””
        ip_to_macs = defaultdict(list)
        mac_to_ips = defaultdict(list)
        seed_dict = defaultdict(dict)
        for line in lines:
           if ‘kernel’ in line:
              junk, data = line.split(“for”)
              ip, mac_strings = data.split(“from”)
              ip = ip.strip()
              mac_raw_list = mac_strings.split(“to”)
              mac_raw_list = [mac.strip() for mac in mac_raw_list]
              for mac in mac_raw_list:
                 if mac not in ip_to_macs[ip]:
                    ip_to_macs[ip].append(mac)
                 if ip not in mac_to_ips[mac]:
                    mac_to_ips[mac].append(ip)
        seed_dict[‘ip_to_mac_mapping’] = ip_to_macs
        seed_dict[‘mac_to_ip_mapping’] = mac_to_ips
        return seed_dict
```

While you didn’t have to create a class for this function, it’s best as an object when called from other objects. You can also add further log parsing functions to this class later if you choose. Here, in `generate_seed_dict()`, you go through the lines in the file, parsing out the IP address and the MACs associated with them, and putting them into a dictionary-like data structure.

TIP This recipe makes heavy use of an object called `defaultdict`, which acts like a dictionary. It’s used mainly for code brevity. When a key is encountered for the first time, if it is not already in the mapping, an entry is automatically created. You can also control what structure the keys value will take. Check it out!

```
def determine_vlan(dev, seed_dict):
   “””Generator function that finds and returns VLAN information for a set of MACs given an IP.

   Args:
      dev (obj): PyEZ JUNOS device with open SSH connection.
      seed_dict (dict): IP/MAC mappings.

   Yields:
      (tuple):
          vlan_name (str): The name of the VLAN as determined by the MX configuration associated
                           with the IP/MAC info.
          vlan_id (str): VLAN tag of the VLAN discovered.
          ip (str): IP Address used for the search and associated with the yielded MACs.
          macs (list): List of strings.  The MAC addresses associated with the IP.

   “””
   for ip, macs in seed_dict[‘ip_to_mac_mapping’].items():
      rt_tbl = RouteTable(dev)
      ifl_tbl = LogicalInterfaceTable(dev)
      rt_tbl.get(ip)
```

```
    # Assumes only a single active element
    find = rt_tbl[0]
    nh_int = find.via
    ifl_tbl.get(nh_int)
    ifl_info = ifl_tbl[0].bridge
    # Capture vlan name and TAG (JUNOS Format = VLAN_NAME+TAGID)
    vlan_name, vlan_id = ifl_info.split('+')
    yield (vlan_name, vlan_id, ip, macs)
```

After obtaining the IP to MAC address mappings in `main.py`, you call `determine_vlan()` from `ipconflictslib.py`. This function is the first to utilize PyEZ's Tables and Views to get information from the device. Here, we are looking through the gateway's routing table, trying to find routes to the IP addresses found in the IP conflicts log. This route lookup will ultimately net the VLAN information and interface we are looking for.

This is a generator function, yielding back the VLANs name, tag, associated IP for the route lookup, and the MACs associated with the IP – all as a tuple. The generator was created because you need to take a set of actions in `main.py` for each tuple found. You can see this loop for yourself starting on line 140 of `main.py`, and the loop provides a chance to print an output for each IP/VLAN searched, as well as keep track of all the MAC address to VLAN mappings:

```
def system_scan(systems_dict, credentials, processes, connect_timeout=None):
  """Recursive function to scan the network, building a data structure of the things it finds.

  Args:
    systems_dict (obj): Defaultdict dictionary like data structure with current view of network.
    credentials (dict): Contains credential information along with the mode of requested
authentication.
    processes (int): Number of concurrent threads for multiprocessing.Pool()
    connect_timeout (int): PyEZ connection timeout in seconds.

  Returns:
    systems_dict (obj): Defaultdict dictionary like data structure housing all data relevant
                            to the network search for IP conflicts.
    all_macs_found (list): All the MAC addresses that where found during the search.
                        Sometimes MAC addressing in the log file do not show up in the
                                 search because the address had timed out already.  We keep track
                                 of what we do find for later.

  """
  depth = len(systems_dict)
  # Keeps track of all MACs that were returned from the search where an interface was found.
  all_macs_found = []
  logger.debug("Depth = {}".format(depth))
  # Find all systems which haven't yet been searched
  tosearch = [system for system in systems_dict if not systems_dict[system]['searched']]
  systems_pool = [[(system, systems_dict[system])] for system in tosearch]
  worker = partial(search_remote_system, credentials, connect_timeout)

  with Pool(processes) as p:
    results = p.map(worker, systems_pool)
  for item in results:
    systems_dict.update(item[0])
```

```
        all_macs_found.extend(item[1])

    new_depth = len(systems_dict.keys())
    logger.debug("New depth = {}".format(new_depth))
    if depth < new_depth:
        return system_scan(systems_dict, credentials, processes)
    else:
        return systems_dict, all_macs_found
```

Here is where you employ the power of recursion and multiprocessing to scan the network and search each system that you find along the way. The function `system_scan()` is called from `main.py` to run this search. A couple of key items here are:

- `depth`: This variable keeps track of how many systems there are in the dictionary.
- `new_depth`: This variable keeps track of how many systems are in the dictionary after each round of scans. If this variable is larger than the original `depth`, it lets the function know to recurse and keep scanning. Otherwise, there have been no changes (no new neighbors found to scan) and the entire dictionary of data is returned to the calling function in `main.py`.
- `Pool(processes)`: This function accepts an integer as an argument that tunes the maximum number of Python processes that will spin up if needed concurrently. Because None is used, the default is to spin up as many as there are CPUs on the local device.
- `Pool.map()`: This class method is one way to achieve concurrency in Python by using the `multiprocessing` library. It accepts a single worker function and a single argument to that function as arguments of itself. The worker function is what will run concurrently.
- `partial()`: Due to the fact that `Pool.map()` accepts worker functions with only a single argument, you need `partial()` in order to create a "half-baked" function that includes all the other arguments needed. Our worker function, `search_remote_system()`, takes more than one argument. Here, we use `partial()` to create a variable `worker` to house our "half-baked" function with two of the three required arguments. The other argument is filled in when calling `Pool.map()` as shown above. `Partials` is from Python's `functools` library.

Initially the logic goes that you look for all systems in the `systems_dict` that do *not* have their "searched" key value set to `True`. This will give us a list of everything that we need to search. We want to give each worker function a key/value pair so that it can easily re-form a singular dictionary structure for the system it is scanning, carved out from the main `systems_dict`. Due to some multi-processing, and behind the scenes logic, each worker can only accept a single argument value, so put what you need into a single tuple and feed it to the worker.

After the worker does its processing, its dictionary will have grown if it has found local neighbors. Each of these neighbors' *searched* key will be set to `False`, while its own will be set to `True`. The original dictionary is extended with this new updated data, the depth of the original dictionary increases, and the process repeats itself until no more neighbors are found.

Making use of the `multiprocessing` library allows you to scan several systems concurrently for a decrease in the time to data. Newly discovered neighbors are fed into a list, which can be distributed to various workers for scanning:

```
def search_remote_system(credentials, connect_timeout, system_data):
   “””Searches a JUNOS system for MAC/IP/VLAN information, discovers neighbors, builds the tree.

   This function is the main worker function, used with multiprocessing, to scan through a JUNOS
   system, call other library functions and build a remote systems dictionary.  This data
   structure maps IP and addressing to VLANs, interfaces and neighbors, along with which devices
   are end leaf devices and which have leaves connected to them.

   Args:
      credentials (dict): Contains credential information along with the mode of requested
                            authentication.
      connect_timeout (int): PyEZ connection timeout in seconds.
      system_data (list): Contains a single tuple with system_ip and a system’s value
                            dictionary housing complete information about what has been found so
                            far for a single device.

   Returns:
      mytuple (tuple): Tuple of _systems_dict (dict) and macs_found (list).
                 _systems_dict gets larger each time this function discovers new devices.
                        Each device marked with a bool ‘searched: True’ if it has been processed,
                        or ‘searched: False’ if it has just been discovered.

                 Macs_found is a total list of all MAC addressing found
                 while searching.

   “””
   assert len(system_data) == 1, “Invalid length of input to ‘search_remote_system’.”
   # Convert tuple argument into dictionary for processing
   system_ip = system_data[0][0]
   _system_dict = dict(system_data)
   # Keep track of macs that we find interfaces for.
   macs_found = []
   logger.info(“#### Searching System {}”.format(system_ip))
```

The worker function `search_remote_system()` is responsible for the scanning and discovery of an individual system. It begins with an assert statement to make sure that the input list `system_data` contains only a single tuple, and not any more or less. This is a basic validity check. Next, recreate the dictionary from this list of tuple elements by converting it into a dictionary with `dict(system_data)`. We also capture the system IP address into a variable for later use. Finally, create a list that will hold all the MAC addresses found during the scan:

```
try:
   with connect_dut(system_ip, credentials, connect_timeout=connect_timeout) as dev:
    interfaces_to_macs = defaultdict(list)
    leaves = defaultdict(dict)
    parent_info = defaultdict(dict)

    # Add basic facts for this system if they don't exist
    keys = _system_dict.keys()
    if 'facts' not in keys:
      personality = dev.facts.get('personality')
      model = dev.facts.get('model')
      _system_dict[system_ip]['facts'] = {'personality': personality, 'model': model}
```

We reuse the `connect_dut()` function to make a connection to the device that `system_ip` represents, and create a few data structures to hold information found along our scan. If device-specific information like "personality" and "model" are not already in your systems dictionary, you can add them now:

```
old_interface = None
for vlan, my_vlans_macs in _system_dict[system_ip]['vlans'].items():
  try:
    learned_interfaces = find_learned_interface(dev, personality, model, vlan, my_vlans_macs)
  except UnsupportedError as e:
    raise e
```

We begin to loop through the system dictionary per VLAN. The goal here is to find all the interfaces that the MAC addresses for that VLAN were learned over, and then map them together. The function `find_learned_interface()` takes care of this. The return value of a dictionary-like object is assigned to the variable `learned_interfaces`:

```
if len(learned_interfaces) > 0:
  for interface, my_ifaces_macs in learned_interfaces.items():
    this_interfaces_macs = []      # Placeholder for MACs in the parent info
    if interface:
      macs_found.extend([mac for mac in my_ifaces_macs if mac not in macs_found])
      local_int = find_local_interface(dev, interface=interface)
      if local_int:
        if old_interface:
          new_interface = local_int
        else:
          old_interface = local_int
```

The returned value from attempting to find interfaces for a specific VLAN could be empty, so if it's not, we will continue. We then loop through the interfaces and associated MAC addresses and attempt to resolve any potential aggregated/bundled type interfaces into a specific physical interface for use with a later LLDP lookup. Sometimes the returned value for the interface within the dictionary is actually None, because an interface was not found, so we skip these by using the if interface statement. We keep track of whether or not this interface was resolved so that we have this information if we need to print out an error message later on:

```
# If you found a local interface via LLDP, attempt to see if there is a remote IP associated.
r_ip, name, description = find_remote_system(dev, model=dev.facts.get('model'),
                                    local_int=local_int)
# If you didn’t find a remote IP in the LLDP table, this interface is a leaf
if not r_ip:
   interfaces_to_macs[interface].extend([mac for mac in my_ifaces_macs if mac not in
                                 interfaces_to_macs[interface]])
```

For every interface found, you attempt to determine if there is a neighboring infrastructure device directly connected. So we use `find_remote_system()` to do this, which returns either the remote IP address and device information, or `None`. If neighboring IP addresses were found, then that interface is a leaf interface, connected to something we cannot scan, so we record the MAC addresses along with the interface to put into our “leaves” data structure:

```
else:
      # Creates an association between the parent and remote systems for these MACs
      parent_info[system_ip][vlan] = {local_int: this_interfaces_macs}
      this_interfaces_macs.extend(my_ifaces_macs)
      # Add a new remote system to the data structure
      if r_ip not in _system_dict:
         _system_dict[r_ip] = {‘personality’: personality, ‘model’: model, ‘name’: name,
                          ‘description’: description, ‘searched’: False,
                          ‘vlans’: {vlan: []}, ‘parent_info’: parent_info}
      # If we have not yet seen this MAC yet, add it to our existing data structure
      try:
         vlan_mac_list = _system_dict[r_ip][‘vlans’][vlan]
         vlan_mac_list.extend([mac for mac in my_ifaces_macs if mac not in vlan_mac_list])
      except KeyError:
         if old_interface == new_interface:
            _system_dict[r_ip][‘vlans’][vlan] = []
            vlan_mac_list = _system_dict[r_ip][‘vlans’][vlan]
            vlan_mac_list.extend([mac for mac in my_ifaces_macs if mac not in vlan_mac_list])
         else:
            logger.error(“Two distinct interfaces are connected from {} to {}. Potential mis-
                                 match access vlan and/or mis-cabling issue.”.format(system_ip,
                                                                                   r_ip))
            logger.error(“Check the following connections: {}:{}, {} <--> {}”
                       .format(system_ip, old_interface, new_interface, r_ip))
         continue
else:
   # If you didn’t find a local interface in the LLDP table, this interface is a leaf
   interfaces_to_macs[interface].extend([mac for mac in my_ifaces_macs if mac not in
                                 interfaces_to_macs[interface]])
```

If an interface with a remote IP address is found, then you must do a few things. First, you need to create mapping between this interface, its neighboring system, and the MAC addresses associated with it. This is called a parent relationship, and it helps to determine where the MACs were learned. We need this because you still aren’t aware if remote IP is accessible yet. The worker eventually picks up this IP and its data in the future, but the first thing the function does is attempt to log in. If you cannot log in, then the system itself is a leaf system and you want to report the directly upstream parent interface in the output. The `parent_info` dictionary is what allows you to do this.

If the newly discovered remote system IP is not yet in the dictionary, add it and some well-known data, like "personality", "model", etc. Here is where you initialize the "searched" key to `False`.

There is a rudimentary check to see that the parent interface and its directly-connected neighbor are within the same VLAN. Sometimes there might be two access interfaces, representing two distinct VLANs, connected together accidently. An error message will be logged in that case:

```
else:
            logger.warning("No MACs found while searching Vlan {} on device {}".format(vlan, system_
ip))
         leaves[vlan].update(interfaces_to_macs)
      _system_dict[system_ip]['leaves'] = leaves
      _system_dict[system_ip]['searched'] = True
except:
   # If you cannot login to the system, it is a leaf node.
   _system_dict[system_ip]['searched'] = True
mytuple = (_system_dict, macs_found)
return mytuple
```

For various reasons, no MAC addresses may be found during a search for any given VLAN, and it outputs a warning message if that happens to be the case.

We update what are considered "leaves" that this searched system contains – the interface along with its associated MAC addressing. It is used to output the endpoint interfaces in later searches. We also update this system's "searched" key to `True` because of the finished search.

The last `except` statement handles the situation if you cannot log in to the device, thus being unable able to search it. As mentioned previously, this device would immediately be considered a leaf endpoint device and its parent would have interfaces of interest for the output.

A tuple, containing the expanded dictionary structure and a list of MACs that were found during the search, is returned:

```
def find_learned_interface(dev, personality, model, vlan_id, macs):
   """Determine from which interfaces a set of MAC addresses was learned.

   Args:
      dev (obj): PyEZ JUNOS device with open SSH connection.
      personality (str): General type of JUNOS device. 'MX' or 'SWITCH' supported.
      model (str): JUNOS model name.  'MX', 'EX' or 'QFX' supported.
      vlan_id (str): VLAN tag of the VLAN where the MACs are located.
      macs (list): MAC addresses for which to search for interfaces.

   Returns:
      parents (obj): DefaultDict dictionary like object keyed by the interface. Maps interfaces
                   to MACs.

   Raises:
      UnsupportedError: If personality or model is not supported, this error is raised.

   """
```

```
    parents = defaultdict(list)
    if personality.lower() == ‘mx’:
        logger.debug(“MACs associated with this IP:”)
        # Compare MACs from duplicate list to MACs in this device’s MAC table
        for mac in macs:
            # Because of invalid XML in 13.3R8.7, cannot search just bridge-domain.
            mac_table = BridgeTable(dev).get(address=mac)
            if len(mac_table.values()) > 0:
                for item in mac_table:
                    if item.vlan_id == vlan_id:
                        parents[item.interface].append(mac)
                        logger.debug(“{} found on {}”.format(mac, item.interface))
            else:
                logger.debug(“No interface found for {}.”.format(mac))

    elif personality == ‘SWITCH’:
        if ‘QFX’ in model:
            mac_table = QFXEtherSwTable(dev).get(vlan_id=vlan_id)
            logger.debug(“MACs associated with this IP:”)
            for item in mac_table:
                # Compare MACs from duplicate list to MACs in this device’s MAC table
                for mac in macs:
                    if mac == item.mac:
                        logger.debug(“{} found on {}”.format(mac, item.interface))
                        parents[item.interface].append(mac)

        elif ‘EX’ in model:
            mac_table = EtherSwTable(dev).get(vlan_name=vlan_id)
            logger.debug(“MACs associated with this IP:”)
            for item in mac_table:
                # Compare MACs from duplicate list to MACs in this device’s MAC table
                for mac in macs:
                    if mac == item.mac:
                        logger.debug(“{} found on {}”.format(mac, item.interface))
                        parents[item.interface].append(mac)
            logger.debug(“Parents: {}”.format(parents))

    else:
        raise UnsupportedError(“No matching Table/View for hostname personality: {}, hostname model: 
{}”
                               .format(personality, model))
    return parents
```

The `find_learned_interface()` function uses PyEZ’s Tables and Views to get interface information about specific MAC addresses from the device being searched. Due to the variety of XML across Junos platforms, we’ve created three separate table/view combinations to support each of the supported device types (MX, QFX, EX). Some of these devices support directly searching for a MAC address as a CLI argument (MX). The others support only filtering, based on the VLAN ID, and then looping through the resultant items. If you have very large L2 Tables, then this may be inefficient. In this lab, it was not a problem.

We are basically looking into the MAC table to see on which interface a specific MAC was learned, then mapping that result, and the MAC, into a dictionary which is returned:

```
def find_local_interface(dev, interface):
    “””Using LLDP, resolve the input interface to the first local physical interface.

    An interface could be an aggregate/bundle (AE) and this function finds, via LLDP, the first
    matching physical interface associated.  This local physical interface will be used to
    determine whether there are any attached or not.

    Args:
        dev (obj): PyEZ JUNOS device with open SSH connection.
        interface (str): The interface we are looking to resolve into a physical interface.

    Returns:
        local_int (str): If the input interface argument is an aggregate/bundle, it will return
                              item.local_parent.
              If the item was a singular interface to begin with, it will return that IFD.
        None: local_int will be ‘None’ if interface not found in LLDP table.

    “””
    def _find_local_int(table):
        “””Internal function to iterate through LLDP table.

        Uses PyEZ Table/Views to determine LLDP information.

        Args:
            table (obj): PyEZ LLDPTable object that acts like a list. Each item in the table is
                              an LLDPTableView object with attributes created via YAML.

        Returns:
            (str): Interface name if a match was found.
            None: If no match is found.

        “””
        logger.debug(“Interface: {}”.format(interface))
        for item in table:
            if item.local_int in interface:
                logger.debug(“No parent match. Interface: {}, item.local_int:
                                    {}”.format(interface, item.local_int))
                return item.local_int
            elif item.local_parent == “-”:
                pass
            elif item.local_parent in interface:
                logger.debug(“Local_parent {} match: local_int: {}”.format(item.local_parent,
                                                                                          item.local_
int))
                # Returns the first local-interface that matches, not all interfaces.
                # This should be OK because an AE locals would be connected to same remote
                        system.
                return item.local_int

    lldp_main_tbl = LLDPTable(dev).get()
    local_int = _find_local_int(lldp_main_tbl)
    return local_int
```

The find_local_interface() function is meant to resolve a discovered interface down to its physical name. The input is either an aggregated or bundled interface, such as an AE, or a singular interface. Either way the output is a single physical interface that the program uses to do an LLDP lookup in order to determine if there are any neighboring systems connected to it. It uses an LLDP table and view you can look through. If a matching interface is not found, then `None` is returned:

```
def find_remote_system(dev, model, local_int):
    “””Finds, via LLDP, whether or not a device has a directly connected neighbor.

    Uses PyEZ Table/Views to determine LLDP information.

    Args:
        dev (obj): PyEZ JUNOS device with open SSH connection.
        model (str): JUNOS model name.  ‘MX’, ‘EX’ or ‘QFX’ supported. Function assumes EX if not
                     MX or QFX.
        local_int (str): Interface within the LLDP table on which to search for a neighbor.

    Returns:
        ip (str): IP address of the neighbor.
        name (str): Hostname of the neighbor.
        description (str): Description string of the neighbor from the LLDP table.

    “””
    logger.debug(“Model: {}”.format(model.lower()))
    if ‘mx’ in model.lower():
        lldp_int_tbl = LLDPInterfaceNeighborMX(dev).get(interface_device=local_int)
    elif ‘qfx’ in model.lower():
        lldp_int_tbl = LLDPInterfaceNeighborMX(dev).get(interface_device=local_int)
    else:
        lldp_int_tbl = LLDPInterfaceNeighborEX(dev).get(interface_name=local_int)
    ip = lldp_int_tbl[0].remote_system_ip
    name = lldp_int_tbl[0].remote_system_name
    description = lldp_int_tbl[0].remote_system_description

    return ip, name, description
```

While the RPC returned for the LLDP main table has similar values across these three Junos systems, the RPC call for a specific interface did not, so we created a couple of Tables and Views. Here, in `find_remote_system()`, you look through the LLDP table for a specific interface and attempt to find if there is a remote system connected. You can grab information such as the IP address, the name, and the description values of the remote system:

```
def create_output_structures(_scanned_systems, _seed_dict, resolve_vendor=False):
    “””Creates an easily consumed data structure for use when printing program findings.

    Args:
        _scanned_systems (dict): Final output of network scan. Contains all relevant system data.
        _seed_dict (dict): Initial Mac/IP mappings.
        resolve_vendor (bool): Allows toggling of MAC OUI vendor resolution.

    Returns:
        output (obj): DefaultDict dictionary like object with all relevant findings.

    “””
    output = defaultdict(lambda: defaultdict(list))
    leaves = None
    mgt_ip = None
    vendor = None
    # Build initial output dictionary
    for system_ip, system_values in _scanned_systems.items():
        try:
            leaves = system_values['leaves']
            mgt_ip = system_ip

        except KeyError:
            for parent, parent_info in system_values['parent_info'].items():
                leaves = parent_info
                mgt_ip = parent
```

Lastly, once all the searching is done and you have everything you need for printing the output, you need to slightly change the formatting to make it more friendly to printing out to the log file. This is what `create_output_structures()`accomplishes. This function is called in `main.py` after all the scanning is done, but before the output printing.

The program starts off initializing some data structures and variables and then begins to assemble the output dictionary. It loops through the larger `_scanned_systems` dictionary (which was a result of the network search) looking for everything it considered leaves, or endpoints, to report in the output. If the system in question has a "leaves" key, then information is taken from there. If it does not have a leaves key, then it was one of the systems to which it couldn't be connected, thus it takes the parent information:

```
assert leaves is not None, "The leaves dictionary does not exist.  Cannot parse."
for vlan_id, values in leaves.items():
    for interface, macs in values.items():
        for mac in macs:
            dup_ip = _seed_dict['mac_to_ip_mapping'][mac]
            if resolve_vendor:
                vendor = determine_vendor(mac)
            for ip in dup_ip:
                output[vlan_id][ip].append((mac, vendor, _scanned_systems[mgt_ip]['name'],
                                            mgt_ip, interface))
```

An `assert` statement validates that the `leaves` variable is not empty, which shouldn't happen. After this, it loops through the information contained within `leaves` based on the VLAN. For each VLAN it looks for interfaces, and for each interface it looks for MACs. Based on the MAC address, you can check your original seed dictionary to find the IP address associated with it and then run a vendor resolution if specified. Once this is done you add this all to the `output` dictionary.

NOTE The key/value structure of this output dictionary matches the order in which you want to print your output. Organized by VLAN, you'd like each IP address to be printed, followed by the MAC addressing and associated information.

```
# Pad each IP in the output dictionary where a MAC was not found.
for vlan, vlan_values in output.items():
    for ip, values in vlan_values.items():
        '''Some MACs for a given IP may not have been found, thus the number of MACs for this IP
            in the current output dictionary might not match the original number of MACs found for
            this IP.
            We need to match up the lengths here and fill in blank values for the missing
            information.
        '''
        original_macs = _seed_dict['ip_to_mac_mapping'][ip]
        original_count = len(original_macs)
        new_count = len(values)
        # original_count should not be less than new_count
        if original_count > new_count:
            current_macs = []
            for value_tuple in values:
                mac_address = value_tuple[0]
                current_macs.append(mac_address)
            org_s = set(original_macs)
            new_s = set(current_macs)
            missing_macs = org_s - new_s
            for mac in missing_macs:
                if resolve_vendor:
                    vendor = determine_vendor(mac)
                blank_tuple = (mac, vendor, None, None)
                output[vlan][ip].append(blank_tuple)
return output
```

The code continues by adding padding for the MAC addressing information not found during the search. An IP address may have been expected to have three MACs associated within the network, but only two of which were found. Here you are adding some padding in the form of `None` for those variables. During the printing of the output, if there is not a MAC found, then you can still print out at least the MAC address as expected, and whose vendor it was, leaving the system from which it was originally learned blank:

```
def determine_vendor(mac):
    """Resolve MAc OUI to hardware vendor

    Args:
        mac (str): MAC address to resolve.

    Returns:
        vendor (str): MAC addressing hardware vendor name.

    """
    v = manuf.MacParser(manuf_name=os.path.join(data_dir, 'manuf'))
    vendor = v.get_manuf(mac)
    return vendor
```

Lastly, `ipconflictslib.py` contains the `determine_vendor()` function, which makes use of the `Manuf.py` library. You instantiate an instance of the `MacParser` class and do a MAC address search through the `manuf` Wireshark text file for a vendor name to return.

PyEZ Tables and Views

This program makes extensive use of PyEZ's Tables and Views concept – it helps get data returned from an RPC call into a Python data object that you can parse. The Tables and Views are loaded into the `ipconflictslib.py` namespace so that the functions within this package can use their names natively. For example:

```
from jnpr.junos.factory import loadyaml

# Create classes per table/view and add them to the global namespace
globals().update(loadyaml(os.path.join(data_dir, 'op_table_views.yml')))
```

Our Tables and Views are operational in nature, meaning that they support returned XML from operational mode commands that are sent to the device. The file itself is located in the `data/` folder. These Tables and Views were created especially for this program and were not (or at least, presumed) part of the default PyEZ install. Let's print them out here for your reference:

```
---
RouteTable:
  rpc: get-route-information
  args:
    table: <change-me.inet.0>
  args_key: destination
  item: route-table/rt
  key: rt-destination
  view: RouteTableView

RouteTableView:
  groups:
    nh: 'rt-entry[current-active]/nh[selected-next-hop]'
  fields_nh:
    via: via


LogicalInterfaceTable:
  rpc: get-interface-information
  args:
    detail: True
    interface_name: '*'
  args_key: interface_name
  item: logical-interface
  key: name
  view: LogicalInterfaceView

LogicalInterfaceView:
  groups:
    irb_domain: 'irb-domain'
    lag: 'lag-traffic-statistics/lag-link'
  fields_irb_domain:
    bridge: irb-bridge
    routing_instance: irb-routing-instance
  fields_lag:
    name: name
```

```
BridgeTable:
  rpc: get-bridge-mac-table
  args:
    address: '*'
  args_key: address
  item: l2ald-mac-entry
  view: BridgeTableView


BridgeTableView:
  fields:
    mac: l2-mac-address
    interface: l2-mac-logical-interface
    vlan_name: l2-mac-bridging-domain
    vlan_id: l2-bridge-vlan

# QFX
QFXEtherSwTable:
  rpc: get-ethernet-switching-table-information
  args:
    vlan_id: '[\d+]'
  item: l2ng-l2ald-mac-entry-vlan/l2ng-mac-entry
  key: l2ng-l2-mac-address
  view: QFXEtherSwView

QFXEtherSwView:
  fields:
    mac: l2ng-l2-mac-address
    vlan_name: l2ng-l2-mac-vlan-name
    interface: l2ng-l2-mac-logical-interface



EtherSwTable:
  rpc: get-vlan-ethernet-switching-table
  args:
    vlan_name: "*"
  item: ethernet-switching-table/mac-table-entry
  key: mac-address
  view: EtherSwView

EtherSwView:
  fields:
    mac: mac-address
    vlan_name: mac-vlan
    interface: mac-interfaces-list/mac-interfaces


LLDPTable:
  rpc: get-lldp-neighbors-information
  item: lldp-neighbor-information
  key: lldp-local-interface | lldp-local-port-id
  view: LLDPView

LLDPView:
  fields:
    local_int: lldp-local-interface | lldp-local-port-id
    local_parent: lldp-local-parent-interface-name
    remote_type: lldp-remote-chassis-id-subtype
    remote_chassis_id: lldp-remote-chassis-id
    remote_port_desc: lldp-remote-port-description

# For EX
LLDPInterfaceNeighborEX:
  rpc: get-lldp-interface-neighbors-information
  args:
    interface_name: '[afgxe]e*'
  item: lldp-neighbor-information
  key: lldp-local-interface
  view: LLDPInterfaceNeighborView

# For MX, QFX
LLDPInterfaceNeighborMX:
  rpc: get-lldp-interface-neighbors
  args:
    interface_device: '[afgx]e*'
  item: lldp-neighbor-information
  key: lldp-local-interface
  view: LLDPInterfaceNeighborView

LLDPInterfaceNeighborView:
  fields:
    local_int: lldp-local-interface
    local_parent: lldp-local-parent-interface-name
    remote_system_name: lldp-remote-system-name
    remote_system_ip: lldp-remote-management-address
    remote_system_description: lldp-system-description/lldp-remote-system-description
```

Discussion

You can choose to run this program via Docker or natively on your localhost within a virtual environment. The assumptions about this recipe are:

- Your network is made up of MX Series, EX Series, or QFX Series devices.
- You are using IRBs and bridge domains on your gateway MX.
- You are using IPv4 within your network. IPv6 is not supported.
- You have Python 3.6.2 installed.

To Install via Docker

1. Install Docker (https://www.docker.com/products/overview)
2. Install Git (https://git-scm.com/downloads)
3. Using Git, clone this project's repository:

```
$ git clone –branch <tag> https://github.com/mmellin/findDuplicateIp.git
```

4. Build the container:

```
$ cd findDuplicateIp
$ docker build –f Dockerfile –t fdip .
```

To Install Via the Virtual Environment

1. Install virtualenv (https://virtualenv.pypa.io/en/stable/installation/)
2. Install Git (https://git-scm.com/downloads)
3. Using Git, clone this project's repository:

```
$ git clone –branch <tag> https://github.com/mmellin/findDuplicateIp.git
```

4. Create a virtual environment:

Python 3:

```
$ cd findDuplicateIp
$ virtualenv env3 –p python3
$ source env3/bin/activate
$ pip install –r requirements.txt
```

Setup

Customize the `RouteTable: args: table` value within `data/op_table_views.yml` for your environment:

```
RouteTable:
  rpc: get-route-information
  args:
    table: <my-main-table.inet.0>
```

The argument here supports a topology with a custom VR as its main table. Enter your VR's custom table name, or just the default table name `inet.0` for IPv4.

Run

Run using either Docker or natively on your localhost.

To run using Docker

Simply run the container with:

```
$ cd findDuplicateIp
$ docker run –it –v $PWD/logs:/fdip/logs fdip <gateway_device> –u <username> --vendor
```

To run natively on your localhost

1. Make sure your virtual environment is running:

```
$ cd findDuplicateIp
$ source env3/bin/activate
```

2. Run `main.py` with arguments:

```
$ python main.py <gateway_device> -u <username> -p <password> --vendor
```

Logging

This program logs `INFO` level output to the screen, and `DEBUG` level input to a log file by default. The log file includes the most verbose logging. Both methods output `WARNING` and `ERROR`. If you want to change this behavior, you will need to modify `conf/logging.yml`.

To turn on DEBUG to screen:

Modify this portion of `logging.yml` as follows:

```
handlers:
  default:
    class: logging.StreamHandler
    level: DEBUG  # Changed from INFO
```

To enable only INFO level to log file:

Modify this portion of `logging.yml` as follows:

```
handlers:
  file:
    class : logging.FileHandler
    level: INFO # Changed from DEBUG
```

The log file during the program run gives several great pieces of information to debug issues. Here is an example of a line from the log:

```
2017/08/28 18:08:33 __main__ 111 - INFO: Copying log file duplicate_ips.0.gz to local disk...
2017/08/28 18:09:20 pkgs.utils.ipconflictslib 72 - ERROR: Failed to connect to 10.161.39.18.
```

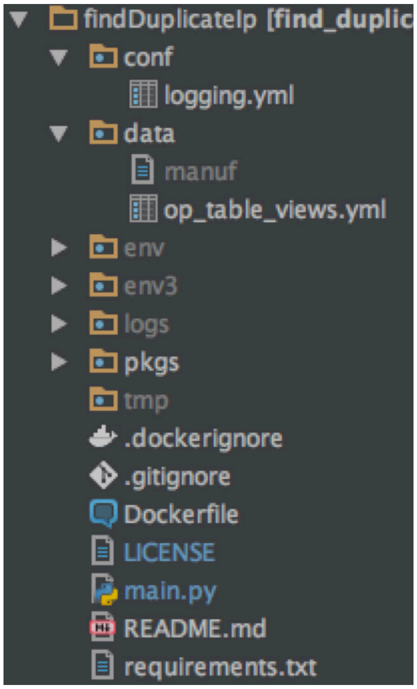
You can see it gives us several key items:

- Date and timestamp: `2017/08/28 18:08:33`
- Namespace: `__main__ or pkgs.util.ipconflictslib`

    - This shows us the name of the module from which the code is running so we can easily identify where any problems or actions are located.
- Code line number: `111`
    - Easily identify which line numbers an issue or an action occurs.
- Log message level: `INFO:`

Repository Structure

To better understand this code base and give you an example of how you can structure your own code, let's look through how the code repository is structured and the components therein.



```
▼ findDuplicateIp [find_duplic
  ▼ conf
      logging.yml
  ▼ data
      manuf
      op_table_views.yml
  ▶ env
  ▶ env3
  ▶ logs
  ▶ pkgs
    tmp
    .dockerignore
    .gitignore
    Dockerfile
    LICENSE
    main.py
    README.md
    requirements.txt
```

Figure 22.3 Local Copy of Development Repository for the findDuplicateIP Program

findDuplicateIp/

This folder is a Git repo and includes hidden folder related to the configuration of Git that has been automatically created after first initialization.

.gitignore

This file names local contents within my repo that I don't want to track and include in my remote repository. These files and folders end up getting greyed out as you can see above. An example of folders shown here are *env*, *env3,* and *logs*. It is best practice not to track your virtual environments folders in version control.

conf/

This folder houses the YAML file for logging configuration. It could also house other configuration and setup-related files.

data/

This is where I organize files related to my PyEZ table/views and the data related to Wireshark's OUI text file "database." Normally, this folder is where I put files related to data that my Python program needs.

env3/

This is the folder that contains my virtual environment, which house Python packages outside of the standard libraries that my script needs to call, along with my Python interpreter of choice. This is ignored via the *.gitignore* file.

logs/

This folder is automatically created by our Python program and is where we dump the log output from our program. It is ignored via the *.gitignore*.

pkgs/

Here is where I typically organize the Python packages and modules that my program uses. It is a package itself, as you can see from the empty __init__.py file located directly in it. This is needed in order for the main program to call code from these packages/modules.

pkgs/utils

This package (note the __init.__.py) contains all my module code for doing all the cool things my program does. I organized it this way so it wouldn't clutter than main.py program, and I can use it in a more object-oriented fashion.

tmp/

This folder is automatically created and used as a dumping point for the log file that we pull off of the router. While we are parsing it and putting it into memory, we use this folder to store the local copy of the file. We then delete the file once parsing is complete. This folder is ignored via the *.gitignore* file.

.dockerignore

This is the equivalent of .gitignore for Docker. We have this so that when we build our docker container, we can send as little data into the Docker context as possible so the build time is fast and efficient.

Dockerfile

This is the file Docker uses to know what to build within its container. It's like a script telling Docker what steps to take and what to build. It is only used if we are building and running our program from a Docker container. Why would you do this? Docker enables portability, so if a system supports Docker, it can run our application without having Python and our libraries installed.

LICENSE

Open source License file for using this software.

main.py

This is the main program file. It is the python file we run when we want to run findDuplicateIp against our network.

README.md

Every repo should have a well-structured README in its repo, and this is ours. It will inform the user of how to install, set up, and run the findDuplicateIp program. README files can be created using a variety of methods and here I'm using Markdown as the syntax of choice.

requirements.txt

The requirements file is where we document the specific, non-standard libraries needed for our program to run. We install these libraries, prior to running our program, either within a virtual environment or Docker. Either method uses pip install –r requirements to install the libraries.

NOTE It is best practice to specify exact versions of the libraries you are using within this file, so that you have no unexpected behavior if a newer version of the library is used, for example: **`junos-eznc==2.1.5`** .

Recipe 23 - Configuration Audit Using XML Schema (XSD)

by Diogo Montagner

- Python Version Used: 2.7
- PyEZ Version Used: 2.15
- Junos OS Used: 17.2
- Juniper Platforms General Applicability: vMX

This recipe will demonstrate how to leverage XML schemas (XSD) to perform network audits against Junos OS MX Series routers.

Problem

As networks grow in size and complexity, the amount of people and systems performing changes in the network follow the same path. To deal with this scenario, many companies enforce a strict change management control in order to limit who is performing changes, when they are doing it, and how. Although this sounds reasonable from a process and risk management perspective, over time, this type of approach eventually slows down the organization, because it is proven that extremely strict processes are not enough to prevent mistakes and deviations. Why? Because the majority of mistakes and deviations seen on networks are caused by human factors. So, what can you do about it?

Let's assume for now that you have higher levels of automation on your network and that all changes are executed only through systems. In a scenario like this, the chances of deviations are minimized because there is very minimal human interaction. But even in scenarios like those mentioned above, deviations can occur. For instance, if you allow network engineers to fix problems via CLI, that opens a door for deviations.

The truth is: you can't avoid deviations. In fact, the best thing you can do is to assume that deviations will occur. In doing so, you acknowledge the fact that this problem, from time to time, will occur in your environment. Even better, you will have an action plan to mitigate it.

So, let's assume the following scenario:

- You have a L3VPN customer who had a problem with their CPE router.
- The customer had to replace the CPE with another device that no longer supports BGP.
- Your network operation team changed the PE-CE routing protocol from BGP to Static in order to recover the service.
- The PE-CE routing protocol will be reverted once the customer replaces the backup CPE with a new one that supports BGP.

In the scenario above, two things can happen:

The customer replaces the CPE and forgets to organize a change with you to revert back to BGP.

Your network supports Static as routing protocol in the PE-CE connection, but the operation team forgot to update the customer documentation.

Arguably, forgetting to revert back to BGP is a small problem compared to forgetting to update the customer documentation. In the first case, the customer's service is working fine, though with a few drawbacks, when compared to the BGP routing option. The biggest problem here is the second case. Let's assume another engineer from the customer rings your operation team complaining that the service is no longer working. The change to Static routing is not registered anywhere on your systems and the engineer from your operations team is not familiar with the change that was made in the past from BGP to Static routing. Your operation engineer then looks up this customer on your customer database and finds out they should be using BGP, instead of Static. The engineer then reverts the configuration to BGP, but the service is still not recovered.

The time taken to figure out the routing protocol misconfiguration was just a distraction from the real problem. This can be costly for the customer's business, because the time it took to recover the service was much longer than expected.

If the network operations team had at least had a tool to alert it about deviations, surely someone would have recorded that the deviation was caused by a CPE replacement and that the reversion to BGP was waiting for the customer to request. If the network operations engineer had such a tool, he would have not wasted time trying to fix the configuration. And he certainly would have asked the customer to confirm the routing protocol currently configured on their side before attempting anything.

The problem described here is one of the many issues that can arise due to misconfiguration, deviations, or out-of-compliance devices. I haven't even described the security implications if the deviations are related to security.

Solution

Implementing a solution that performs configuration audits regularly on your network devices can prevent and mitigate issues like the ones described in this recipe. It would be even better if you could perform the audit as soon as a configuration change is made, which is called *event-driven automation,* but that is beyond the scope of this recipe.

There are multiple ways to perform configuration audits in a network device. Most of the configuration audits we have seen in the field often rely on complex regular expressions or innumerable lines of code containing nested `if-then-else` instructions. While this works, and may be better than not having an audit process, it does pose many challenges as the networks and services delivered by these networks grow in size and complexity. The issue here lies in the fact that many of us are still developing automation for humans instead of developing automation for machines. And here is where this PyEZ recipe comes in to help you with network configuration audit.

This recipe provides you with an example of small framework for network configuration audit. It will be up to you to develop it further to cover all aspects of the configuration audits that you need.

Let's start looking to the architecture of this framework. Figure 23.1 presents an overview of the components.

The provisioning system shown in Figure 23.1 just demonstrates that the audit tool is independent from the provisioning system. Also, you may be wondering why there is an inventory in this picture if the recipe is about configuration audit. The truth is that you need a source of truth to compare against, otherwise your network audit has no value. In the case of this recipe, the source of truth is the inventory. Both provisioning and audit systems interact with the inventory while executing their tasks. For the sake of simplicity, our inventory system will be a simple YAML file describing each router.

NOTE There are many design decisions that come into play while designing your inventory and source of truth systems. Here, we decided to track on a per device basis and not a per service or function basis. While this model offers simplicity in storing the data and easily enforcing templates, it is not the best model for service modeling because the information is kept at the device level. So, if you need to find out where an ABC service is configured, you need to walk all devices in the network, verifying if they have the ABC service configured. Converting this approach to another approach where you can easily track the services is not a difficult task, and it can be built on top of the model used here. Get into the lab and try it.

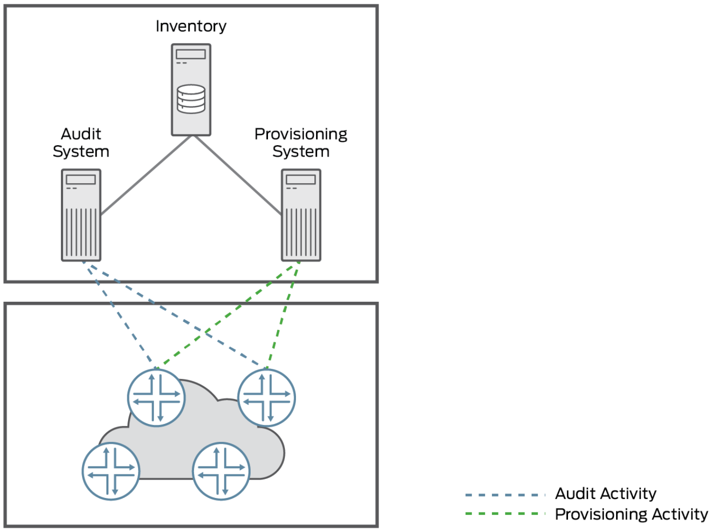


Figure 23.1 *Overview of Configuration Audit Recipe Components*

The following code represents the inventory file for our `vmx1` router:

```
---
infrastructure:
  router_hostname: vmx1
  router_lo0: '1.1.1.1/32'
  router_mgmt_fxp0: '192.168.122.35'
  uplinks:
    - {interface: 'ge-0/0/0.0', address: '10.1.1.1/30', mpls: 'yes', ospf: 'yes', ldp: 'yes', rsvp: 
'no'}
    - {interface: 'ge-0/0/1.0', address: '10.1.1.5/30', mpls: 'yes', ospf: 'yes', ldp: 'yes', rsvp: 
'no'}

services:
  l3vpn:
    - service_name: 'VPNA'
      service_type: 'vrf'
      rt: '65000:100'
      rd: '65000:100'
      sites:
        - {site_id: 'site1', description: 'VPNA Site 1', interface: 'ge-0/0/2.100', protocol: 'bgp', 
peer_as: '65001', address: '10.100.100.1/30', bgp_neighbor: '10.100.100.2'}
        - {site_id: 'site2', description: 'VPNA Site 2', interface: 'ge-0/0/2.101', protocol: 'bgp', 
peer_as: '65001', address: '10.100.100.5/30', bgp_neighbor: '10.100.100.6'}
        - {site_id: 'site3', description: 'VPNA Site 3', interface: 'ge-0/0/2.102', protocol: 'bgp',
peer_as: '65001', address: '10.100.100.9/30', bgp_neighbor: '10.100.100.10'}
    - service_name: 'VPNB'
      service_type: 'vrf'
      rt: '65000:200'
      rd: '65000:200'
      sites:
        - {site_id: 'site1', description: 'VPNB Site 1', interface: 'ge-0/0/2.200', protocol: 'bgp',
peer_as: '65001', address: '10.100.100.1/30', bgp_neighbor: '10.100.100.2'}
        - {site_id: 'site2', description: 'VPNB Site 2', interface: 'ge-0/0/2.201', protocol: 'bgp',
peer_as: '65001', address: '10.100.100.5/30', bgp_neighbor: '10.100.100.6'}
        - {site_id: 'site3', description: 'VPNB Site 3', interface: 'ge-0/0/2.202', protocol: 'bgp',
peer_as: '65001', address: '10.100.100.9/30', bgp_neighbor: '10.100.100.10'}
```

As you can see, there is a section of the YAML file that describes the infrastructure part of the router (lo0, hostname, uplinks, and protocols) as well as a section for the services.

MORE? We are assuming you are familiar with the configuration of MPLS L3VPNs in Junos – in case you are not, refer to the Junos VPN Guide: https://www.juniper.net/documentation/en_US/junos/information-products/pathway-pages/config-guide-vpns/index.html.

The focus of this recipe is in the audit of the L3VPN services, and Figure 23.2 illustrates the architecture of the solution.

Figure 23.2 presents the workflow of this recipe. As mentioned before, the building blocks presented here can be used to build a much larger and complete configuration audit system.

You may not be familiar with some of the components presented in the architecture of this recipe. Moreover, a recipe collection is not the best place to describe each of the frameworks and data formats in detail. But if you are comfortable with the components presented in the architecture, you can skip this part, otherwise, we will provide a brief introduction on each component with a few pointers as to where to find additional information about them.

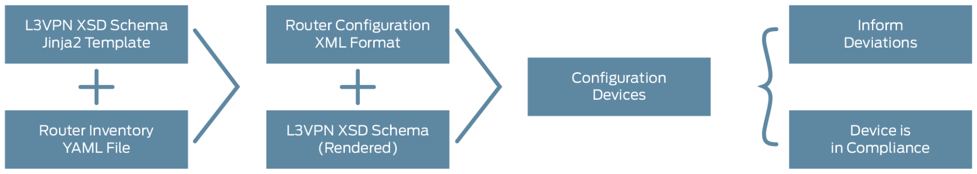

L3VPN XSD Schema Jinja2 Template
+
Router Inventory YAML File
Router Configuration XML Format
+
L3VPN XSD Schema (Rendered)
Configuration Devices
Inform Deviations
Device is in Compliance

Figure 23.2 Auditing L3VPN Services

YAML

YAML stands for Yet Another Markup Language. From its Wikipedia definition, YAML is a human-readable data serialization language. In a more informal definition, YAML is a simple way to present structured data. It can be easily loaded into a JSON object.

An example of YAML file is the inventory file used earlier in this recipe.

CAUTION YAML is very sensitive to indentation. Make sure you have your favorite text editor configured to replace tabs with spaces.

XML

XML stands for Extensible Markup Language. Similar to YAML, it is a way to present structured data. Contrary to YAML, XML is more complex and powerful but sometimes can repel people because it is not as readable by humans as YAML. The example below shows a Juniper router interface configuration presented in XML:

```
<configuration junos:commit-seconds="1505525716" junos:commit-localtime="2017-09-16 01:35:16 UTC"
junos:commit-user="dmontagner">
   <interfaces>
      <interface>
         <name>ge-0/0/0</name>
         <description>connection with vmx2</description>
         <flexible-vlan-tagging/>
         <encapsulation>flexible-ethernet-services</encapsulation>
         <unit>
            <name>100</name>
            <vlan-id>100</vlan-id>
            <family>
               <inet>
                  <address>
                     <name>10.254.254.2/30</name>
                  </address>
               </inet>
            </family>
         </unit>
      </interface>
   </interfaces>
</configuration>
```

Another key difference between XML and YAML is that you can use XSL Transformation to transform one XML document into another. This is where things start to get complicated, because XSLT is not user friendly (it is machine friendly).

XML also provides a way to validate documents via one of the two schema languages: Document Type Definition (DTD) and XML Schema Definition (XSD). DTD is the elder schema language and while it does the job, it falls short in many aspects. XSD is a newer schema language and is far more powerful than DTD. Its rich data type, associated with its new features in the XSD 1.1, enables reliable

XML documents validation.

The XML XSD is the key component of this recipe.

MORE? The best XML tutorial we have seen for beginners is the one from https://www.w3schools.com .

XML Schema (XSD)

As mentioned in the previous section, the XML Schema Definition (XSD) is a way to validate the structure and the content of a XML document. This recipe explores the XSD 1.0 features only to keep it simple.

MORE? To explore the XSD 1.1 features, try this blog article outlining the new cool features of XSD 1.1 at: https://blogs.infosupport.com/exploring-cool-new-features-of-xsd-1-1/.

The creation of the schema document can be a tedious process. It is a bit more complicated than dealing with a simple and plain XML document. Let's present a few tips in order to ease the pain and help you to shortcut on this problem. Consider this simple XML document:

```
<a>
   <b>vmx</b>
</a>
```

The schema to validate this document would be:

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" xmlns:xs="http://www.w3.
org/2001/XMLSchema">
  <xs:element name="a">
    <xs:complexType>
      <xs:sequence>
        <xs:element type="xs:string" name="b"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

This schema does not validate the content of the element `b` in the XML document. If you use this schema to validate the XML document presented earlier, whatever value you assign to element `b`, it will be accepted. This is not acceptable while performing network configuration audits. The good thing is that this problem can be solved using XSD 1.0 (don't need 1.1 yet). Below is the XSD to validate the content of element `b`:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="a" type="Aelem"/>
      <xsd:complexType name="Aelem">
      <xsd:sequence>
         <xsd:element minOccurs="1" maxOccurs="2" name="b" type="TElemBVal" />
      </xsd:sequence>
   </xsd:complexType>
   <xsd:simpleType name="TElemBVal">
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="mx"></xsd:enumeration>
         <xsd:enumeration value="ptx"></xsd:enumeration>
         <xsd:enumeration value="qfx"></xsd:enumeration>
      </xsd:restriction>
   </xsd:simpleType>
</xsd:schema>
```

Now this schema only accepts `mx`, `ptx`, or `qfx` as the value of element `b`. If the value of element `b` is `abc`, the validation of the XML document will fail. What XSD 1.0 does not solve in this case is the test if one of the values of `b` is not present or if a combination of them is present or missing. However, XSD 1.1 *will not be covered* in this recipe, but we highly encourage you to use XSD 1.1 once you are familiar with XSD 1.0.

NOTE The schema document can be generated in a few different ways. In this particular case, I am using the *Russian Doll XSD* design.

MORE? If you think that generating these schemas is too hard, even for a simple XML document, you are not alone. There are a few online XSD generators that will help you simplify this process, such as those at freeformater.com. It does not provide everything (for example, it does not generate the enumeration options) but it saves more than 80% of your time while creating the schema documents. See: https://www.freeformatter.com/xsd-generator.html.

Jinja2 Templates

Jinja2 is a template language for Python. It is widely used for configuration and document generation. It is modelled after the DJango's templates.

Basically, you insert variables in the middle of your text and then you render the template passing values to these variables. During the rendering, the variables will be replaced by the values you have passed on. Below is a simple example of Jinja2 template:

```
interfaces {
   ge-0/0/0 {
      description "connection with vmx2";
      flexible-vlan-tagging;
      encapsulation flexible-ethernet-services;
      unit {{ vlan_id }} {
         vlan-id {{ vlan_id }};
         family inet {
                address {{ interface_ptp_ip }};
            }
        }
    }
}
```

This configuration shows a Junos router interface configuration. Note that the `unit` number, the `vlan-id`, and the interface `IP address` have been replaced by Jinja2 variables. If you were going to render this template, you need to pass two values: one for the `vlan_id` variable and another for the `interface_ptp_ip` variable. The rendering engine will replace them with the values you have passed on.

Jinja2 can be much more powerful than this example, of course. It allows you to include loops and if-then conditions in order to have multiple iterations and conditional rendering.

Putting Everything Together

Now it's time to understand how this recipe works. The workflow description follows the architecture illustrated in Figure 23.2. Here are the steps, and a detailed description of each one.

Step 1

The first step is to generate the schema document (XSD) that will validate the `vmx1` router. The schema document at this stage is a XSD 1.0 Jinja2 template file because you want to reuse the template across multiple routers. For each router, you will have a different inventory file (YAML file).

What is happening at this stage is the Jinja2-template rendering using the inventory data as input for the Jinja2 variables. To execute you need the Jinja2 XSD template and the inventory file for `vmx1`, so let's start by looking into the inventory file for `vmx1` (filename = `router_vmx1.yaml`):

```
---
infrastructure:
  router_hostname: vmx1
  router_lo0: '1.1.1.1/32'
  router_mgmt_fxp0: '192.168.122.35'
  uplinks:
    - {interface: 'ge-0/0/0.0', address: '10.1.1.1/30', mpls: 'yes', ospf: 'yes', ldp: 'yes', rsvp: 
'no'}
    - {interface: 'ge-0/0/1.0', address: '10.1.1.5/30', mpls: 'yes', ospf: 'yes', ldp: 'yes', rsvp: 
'no'}

services:
  l3vpn:
    - service_name: 'VPNA'
      service_type: 'vrf'
      rt: '65000:100'
      rd: '65000:100'
```

```
     sites:
       - {site_id: 'site1', description: 'VPNA Site 1', interface: 'ge-0/0/2.100', protocol: 'bgp',
peer_as: '65001', address: '10.100.100.1/30', bgp_neighbor: '10.100.100.2'}
       - {site_id: 'site2', description: 'VPNA Site 2', interface: 'ge-0/0/2.101', protocol: 'bgp',
peer_as: '65001', address: '10.100.100.5/30', bgp_neighbor: '10.100.100.6'}
       - {site_id: 'site3', description: 'VPNA Site 3', interface: 'ge-0/0/2.102', protocol: 'bgp',
peer_as: '65001', address: '10.100.100.9/30', bgp_neighbor: '10.100.100.10'}
   - service_name: 'VPNB'
     service_type: 'vrf'
     rt: '65000:200'
     rd: '65000:200'
     sites:
       - {site_id: 'site1', description: 'VPNB Site 1', interface: 'ge-0/0/2.200', protocol: 'bgp',
peer_as: '65001', address: '10.100.100.1/30', bgp_neighbor: '10.100.100.2'}
       - {site_id: 'site2', description: 'VPNB Site 2', interface: 'ge-0/0/2.201', protocol: 'bgp',
peer_as: '65001', address: '10.100.100.5/30', bgp_neighbor: '10.100.100.6'}
       - {site_id: 'site3', description: 'VPNB Site 3', interface: 'ge-0/0/2.202', protocol: 'bgp',
peer_as: '65001', address: '10.100.100.9/30', bgp_neighbor: '10.100.100.10'}
```

The next thing you need is the Jinja2 template for the XSD file. Note that in this recipe, only the L3VPN services part will be validated. Consequently, the Jinja2 XSD template will only cover the L3VPN services. The code below presents the Jinja2 XSD template (filename = `l3vpns.xsd.j2`):

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" xmlns:xs="http://www.w3.
org/2001/XMLSchema">
  <xs:element name="instance">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name">
          <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="{{ template['service_name'] }}"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="instance-type">
          <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="{{ template['service_type'] }}"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="interface" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="name">
                <xs:simpleType>
                  <xs:restriction base="xs:string">
                    {% for site in template['sites'] -%}
                      <xs:enumeration value="{{ site['interface'] }}"/>
                    {% endfor %}
                  </xs:restriction>
                </xs:simpleType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
```

```
</xs:element>
<xs:element name=”route-distinguisher”>
  <xs:complexType>
    <xs:sequence>
      <xs:element name=”rd-type”>
        <xs:simpleType>
          <xs:restriction base=”xs:string”>
            <xs:enumeration value=”{{ template[‘rd’] }}”/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name=”vrf-target”>
  <xs:complexType>
    <xs:sequence>
      <xs:element name=”community”>
        <xs:simpleType>
          <xs:restriction base=”xs:string”>
            <xs:enumeration value=”target:{{ template[‘rt’] }}”/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name=”protocols”>
  <xs:complexType>
    <xs:sequence>
      <xs:element name=”bgp”>
        <xs:complexType>
          <xs:sequence>
            <xs:element name=”group”>
              <xs:complexType>
                <xs:sequence>
                  <xs:element type=”xs:string” name=”name”/>
                  <xs:element type=”xs:string” name=”type”/>
                  <xs:element type=”xs:int” name=”peer-as”/>
                  <xs:element type=”xs:string” name=”as-override”/>
                  <xs:element name=”neighbor” maxOccurs=”unbounded” minOccurs=”0”>
                    <xs:complexType>
                      <xs:sequence>
                        <xs:element name=”name”>
                          <xs:simpleType>
                            <xs:restriction base=”xs:string”>
                            {% for site in template[‘sites’] -%}
                              <xs:enumeration value=”{{ site[‘bgp_neighbor’] }}”/>
                            {% endfor %}
                            </xs:restriction>
                          </xs:simpleType>
                        </xs:element>
                      </xs:sequence>
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
```

```
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>
</xs:schema>
```

The schema document looks complex, but it isn't really. In fact, the task was just to add the Jinja2 variables and the enumeration rules, but as you can see everything else was generated by the online XSD generator. So, don't abandon it yet!

Step 2

Alright, now that we've got the schema document (XSD) and the inventory file (YAML), let's render it to get the final schema document. This is the schema that will be used to validate the L3VPN services configuration of `vmx1`.

As mentioned earlier in this recipe, Jinja2 is a template language for Python, so our render will be coded in Python. But instead of creating a separate render, you include the render in the main script that performs the audit in the L3VPN services. This may not always be the best option, but it does serve the purpose of this recipe. In a scaled production environment, you would need to have a micro-service just for the render. The code presented here is an excerpt from the audit script, highlighting only the relevant part to the render:

```
<... omitted for brevity ...>

    router_inventory_file = "router_vmx1.yaml"

<... omitted for brevity ...>

    try:
        inventory_file = open(router_inventory_file, "r")

    except Exception as e:
        print "error: error opening the file " + router_inventory_file
        print e
        sys.exit(-1)

    yaml_file = yaml.load(inventory_file)

<... omitted for brevity ...>

    for vpn in yaml_file['services']['l3vpn']:

<... omitted for brevity ...>

        rendered_schema_file = str(render_l3vpn_schema(yaml_file, l3vpn_schema_template, vpn['service_
name']))

<... omitted for brevity ...>
```

You can read that the code is responsible for generating the final schema document that contains specific information about the VPN obtained from the inventory file (the YAML file). Now let's use this schema to validate `vmx1`'s L3VPN services configuration in Step 3.

Step 3

Step 3 involves collecting the vmx1 configuration. Since only L3VPN services are going to be validated, you only need to collect the configuration under the `routing-instances` tree (the CLI equivalent of `show configuration routing-instances`).

That code responsible to connect to the router and extract the routing-instance tree for each VPN is shown here:

```
<... omitted for brevity>

   try:
      vmx1_mgmt = '127.0.0.1'
      rtUser = "lab"
      rtPassword = "lab123"
      dev = Device(host=vmx1_mgmt, user=rtUser, password=rtPassword, gather_facts=False)
      dev.open()
      print "\nConnection to %s established with success!" % vmx1_hostname

   except Exception as e:
      print "error: could not connect to %s" % vmx1_hostname
      print e
      sys.exit(-1)

   try:
      xmlConfig = dev.rpc.get_config()
      print "\nConfiguration collected from %s" % vmx1_hostname

   except Exception as e:
      print "error: could not collect the configuration from router %s" % vmx1_hostname
      print e
      sys.exit(-1)

<... omitted for brevity>
```

In case you didn't notice, the code actually collected the entire router configuration. That wasn't a mistake. You need the entire configuration to audit every single L3VPN configured in the router – but how do you extract just the routing-instance configuration for a particular VPN? The code responsible for extracting that rout ing-instance configuration from the complete configuration is here:

```
<... omitted for brevity>

      if (len(xmlConfig.xpath("//configuration/routing-instances/instance[name=\"" + vpn['service_
name'] + "\"]")) > 0):
        # The VPN configuration exist in the router. Proceed with the audit.
        for rtInstance in xmlConfig.xpath("//configuration/routing-instances/instance[name=\"" +
vpn['service_name'] + "\"]"):

           if (rtInstance.xpath("./name")[0].text == vpn['service_name']):
              my_routing_instance_str = etree.tostring(rtInstance)

<... omitted for brevity>
```

The routing-instance configuration will be stored in `rtInstance` (as an XML object) and in `my_routing_instance_str` as a string.

Step 4 (final step)

Now let's execute the validation of the routing-instance configuration extracted from the complete configuration of the `vmx1` router against the XML schema generated for this particular VPN. At this stage in the code, all the elements needed are generated and stored in memory. You only need to parse the XML document for the VPN using the schema previously generated. The excerpt of code responsible for Step 4 is:

```
<... omitted for brevity>

              try:
                 etree.fromstring(my_routing_instance_str, myXMLparser)
                 print "    - audit results: PASS"
                 log.warn("Routing instance configuration for vpn %s validated against %s" %
(vpn['service_name'], l3vpn_schema_template))
              except etree.XMLSchemaError as e1:
                 log.warn("error: error validating routing instance %s" % vpn['service_name'])
                 print "    - audit results: FAIL"
                 print e1.message

              except Exception as e:
                 log.warn("error validating routing instance %s" % vpn['service_name'])
                 print e

<... omitted for brevity>
```

If the audit fails, the exception message will indicate where and why it failed.

Performing Configuration Audit for L3VPNs

In order to demonstrate how to use the audit script we first need to recap the automation workflow. Figure 23.3 illustrates a more detailed workflow than Figure 23.2 presented earlier.

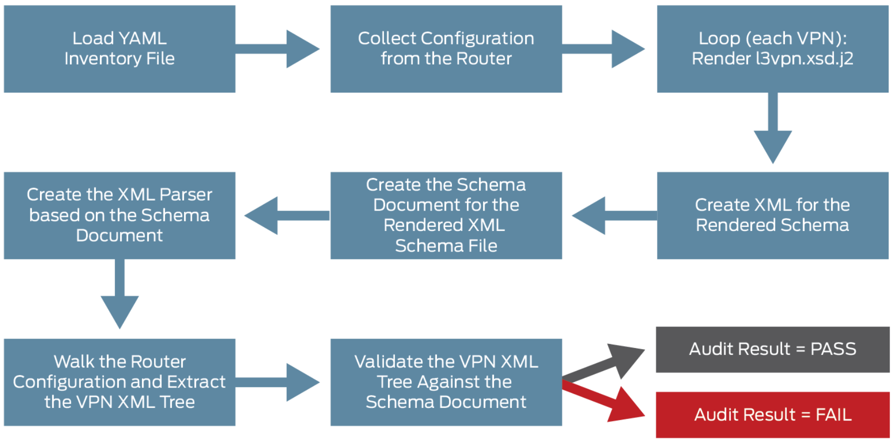

Load YAML Inventory File
Collect Configuration from the Router
Loop (each VPN): Render l3vpn.xsd.j2
Create XML for the Rendered Schema
Create the Schema Document for the Rendered XML Schema File
Create the XML Parser based on the Schema Document
Walk the Router Configuration and Extract the VPN XML Tree
Validate the VPN XML Tree Against the Schema Document
Audit Result = PASS
Audit Result = FAIL

Figure 23.3 Using the Audit Script Workflow

Instead of presenting the complete configuration for VMX1, only the configuration we will be auditing is listed here. The full configuration of the VMX1 is available with the all the code of this recipe at this book's GitHub repository. The VMX1's routing-instances configuration follows:

```
dmontagner@vmx1> show configuration routing-instances
VPNA {
    instance-type vrf;
    interface ge-0/0/2.100;
    interface ge-0/0/2.101;
    interface ge-0/0/2.102;
    route-distinguisher 65000:100;
    vrf-target target:65000:100;
    protocols {
        bgp {
            group CEs {
                type external;
                peer-as 65001;
                as-override;
                neighbor 10.100.100.2;
                neighbor 10.100.100.6;
                neighbor 10.100.100.10;
            }
        }
    }
}
VPNB {
    instance-type vrf;
    interface ge-0/0/2.200;
    interface ge-0/0/2.201;
    interface ge-0/0/2.202;
    route-distinguisher 65000:200;
    vrf-target target:65000:200;
    protocols {
      bgp {
         group CEs {
            type external;
            peer-as 65001;
            as-override;
            neighbor 10.100.100.2;
            neighbor 10.100.100.6;
            neighbor 10.100.100.10;
         }
      }
   }
}
```

Now that all elements have been presented, let's perform the L3VPN configuration audit against vmx1.

```
dmontagner@querencia> ./validate_vmx1_config.py

Connection to vmx1 established with success!

Configuration collected from vmx1

Auditing VPN VPNA ...
   - audit results: PASS

Auditing VPN VPNB ...
   - audit results: PASS
```

As you can observe, the audit was successful to both VPNs. Now, let's introduce two errors in the VPNA configuration:

```
dmontagner@vmx1> configure private
warning: uncommitted changes will be discarded on exit
Entering configuration mode

[edit]
dmontagner@vmx1# delete routing-instances VPNA protocols bgp group CEs as-override

[edit]
dmontagner@vmx1# set routing-instances VPNA route-distinguisher 1234:1234

[edit]
dmontagner@vmx1# commit and-quit
commit complete
Exiting configuration mode

dmontagner@vmx1>

dmontagner@vmx1> show configuration routing-instances VPNA
instance-type vrf;
interface ge-0/0/2.100;
interface ge-0/0/2.101;
interface ge-0/0/2.102;
route-distinguisher 1234:1234;
vrf-target target:65000:100;
protocols {
   bgp {
        group CEs {
            type external;
            peer-as 65001;
            neighbor 10.100.100.2;
            neighbor 10.100.100.6;
            neighbor 10.100.100.10;
        }
    }
}
```

With a configuration for VPNA that is different from its template, let's now see if our audit tool can detect these issues:

```
dmontagner@querencia> ./validate_vmx1_config.py

Connection to vmx1 established with success!

Configuration collected from vmx1

Auditing VPN VPNA ...
Element 'rd-type': [facet 'enumeration'] The value '1234:1234' is not an element of the set
{'65000:100'}. (line 0)

Auditing VPN VPNB ...
    - audit results: PASS
```

The audit tool only found one issue, not two. The good thing is that it found that the configuration is out of compliance. The problem here is that the XML parser terminated on the first validation error it encountered. So, how can you proceed the validation through the broken (not compliant with the schema) XML ?

By default, the *lxml* parser does not try to recover from errors, so setting the `recover` parameter to True will modify this behavior. However, the parser will not always be able to recover, so the suggestion is to always stop on the *first* out-of-compliance issue. This will ensure nothing goes missing:

```
dmontagner@querencia> ./validate_vmx1_config.py

Connection to vmx1 established with success!

Configuration collected from vmx1

Auditing VPN VPNA ...
Element 'neighbor': This element is not expected. Expected is ( as-override ). (line 0)

Auditing VPN VPNB ...
    - audit results: PASS
```

If you fix the first issue pointed out by our tool, then you will find that the tool shows the next issue when you run it again, as demonstrated. In the next sequence, we are fixing the second problem and putting the configuration back in compliance:

```
dmontagner@vmx1> configure private
warning: uncommitted changes will be discarded on exit
Entering configuration mode
```

```
[edit]
dmontagner@vmx1# set routing-instances VPNA protocols bgp group CEs as-override

[edit]
dmontagner@vmx1# commit and-quit
commit complete
Exiting configuration mode

dmontagner@vmx1>
```

And when you run the configuration audit again, you will see the audit report as PASS for both VPNs:

```
dmontagner@querencia> ./validate_vmx1_config.py

Connection to vmx1 established with success!

Configuration collected from vmx1

Auditing VPN VPNA ...
   - audit results: PASS

Auditing VPN VPNB ...
   - audit results: PASS
```

As explained earlier, we're using XSD 1.0 for this recipe. The fact that we were using XSD 1.0 instead of XSD 1.1 might create a gap in some audit scenarios. Be aware the gap can be addressed using XSD 1.1, but it was not shown here for the sake of brevity.

Complete Code of Recipe 23

Here is the complete code of Recipe 23, also available at https://github.com/Juniper/junosautomation/tree/master/pyez. Note that in order to run this recipe, you will need the YAML and Jinja2 files presented in steps 1 to 4. These two files are also available in the cookbook's repository.

```
#!/opt/local/bin/python

import lxml
from lxml import etree
import sys
import StringIO
import yaml
from jnpr.junos import Device
from jinja2 import Environment, FileSystemLoader, Template
import pprint
import logging


# setting logging capabilities
log = logging.getLogger() # 'root' Logger
console = logging.StreamHandler()
format_str = '%(asctime)s\t%(levelname)s -- %(funcName)s %(filename)s:%(lineno)s -- %(message)s'
console.setFormatter(logging.Formatter(format_str))
```

```
log.addHandler(console) # prints to console.

# set the log level here
#log.setLevel(logging.INFO)
log.setLevel(logging.ERROR)


def render_l3vpn_schema(router_inventory_file, schema_template_file, vpn_name):

   ENV = Environment(loader=FileSystemLoader('./'))

   try:
      template = ENV.get_template(schema_template_file)
      log.info("template %s rendered !!!" % schema_template_file)

   except Exception as e:
      print "error: error rendering the template " + schema_template_file
      print e
      return None

   log.info("")
   log.info("printing the rendered template ...")
   log.info("")

   for vpn in router_inventory_file['services']['l3vpn']:
      if vpn['service_name'] == vpn_name:
         rendered_template = template.render(template=vpn)
         log.debug(rendered_template)
         return rendered_template

   # if got here, it means no VPN with service_name equals to vpn_name has been found
   print "error: could not find VPN %s" % vpn_name
   return None



def main():


   router_inventory_file = "router_vmx1.yaml"
   l3vpn_schema_template = "l3vpns.xsd.j2"
   l3vpn_schema_file = "l3vpns.xsd"

   #
   # loading the router inventory data into a dictionary
   #
   try:
      inventory_file = open(router_inventory_file, "r")

   except Exception as e:
      print "error: error opening the file " + router_inventory_file
      print e
      sys.exit(-1)

   yaml_file = yaml.load(inventory_file)

   vmx1_hostname = yaml_file['infrastructure']['router_hostname']
   vmx1_mgmt = yaml_file['infrastructure']['router_mgmt_fxp0']


   #
   # collecting the router XML configuration
   #
```

```
    # temporary override
    rtUser = “lab”
    rtPassword = “lab123”

    try:
        dev = Device(host=vmx1_mgmt, user=rtUser, password=rtPassword, port=2222, gather_facts=False)
        dev.open()
        print “\nConnection to %s established with success!” % vmx1_hostname

    except Exception as e:
        print “error: could not connect to %s” % vmx1_hostname
        print e
        sys.exit(-1)

    try:
        xmlConfig = dev.rpc.get_config()
        print “\nConfiguration collected from %s” % vmx1_hostname

    except Exception as e:
        print “error: could not collect the configuration from router %s” % vmx1_hostname
        print e
        sys.exit(-1)

    #
    # looping through each VPN in the inventory file
    #
    for vpn in yaml_file[‘services’][‘l3vpn’]:

        print “”
        print “Auditing VPN %s ...” % vpn[‘service_name’]

        #
        # rendering the schema file for this VPN
        #
        rendered_schema_file = str(render_l3vpn_schema(yaml_file, l3vpn_schema_template, vpn[‘service_
name’]))

        if rendered_schema_file is None:
            print “”
            print “error: could not render the schema file !!!”
            print “”
            sys.exit(-1)

        #
        # creating the XML document for the rendered schema file
        #
        try:
            schemaXML = etree.fromstring(rendered_schema_file)
            log.warn(“XML doc for %s created!” % l3vpn_schema_template)

        except Exception as e:
            print “could not create XML doc from XML schema file %s” % l3vpn_schema_template
            print e
            sys.exit(-1)

        #
        # creating the schema document for the rendered XML document of the schema file
        #
        try:
            schemaDoc = etree.XMLSchema(schemaXML)
            log.warn(“XML schema created for vpn %s with schema file %s” % (vpn[‘service_name’], l3vpn_
schema_template))
```

```
    except Exception as e:
      print “could not create XML schema for vpn %s based on schema file %s” % (vpn[‘service_name’], 
l3vpn_schema_template)
      print e
      sys.exit(-1)

    #
    # creating the XML parser based on the XML schema created previously
    #
    try:
      myXMLparser = etree.XMLParser(schema=schemaDoc)
      log.warn(“XML parser based on rendered %s schema created!” % l3vpn_schema_template)
    except Exception as e:
      print “could not create XML parser based on routing_instance.xsd schema”
      print e
      sys.exit(-1)

    #
    # validating the L3VPNs configuration based on l3vpn service schema
    #
    log.warn(“walking on xml tree for vpn %s” % vpn[‘service_name’])

    if (len(xmlConfig.xpath(“//configuration/routing-instances/instance[name=\”” + vpn[‘service_
name’] + “\”]”)) > 0):

      # The VPN configuration exist in the router. Proceed with the audit.
      for rtInstance in xmlConfig.xpath(“//configuration/routing-instances/instance[name=\”” + 
vpn[‘service_name’] + “\”]”):

        if (rtInstance.xpath(“./name”)[0].text == vpn[‘service_name’]):
          my_routing_instance_str = etree.tostring(rtInstance)
          log.debug(my_routing_instance_str)

          log.warn(“Validating routing instance configuration for vpn %s against schema %s ...” % 
(vpn[‘service_name’], l3vpn_schema_template))

          try:
            etree.fromstring(my_routing_instance_str, myXMLparser)
            print “   - audit results: PASS”
            log.warn(“Routing instance configuration for vpn %s validated against %s” % 
(vpn[‘service_name’], l3vpn_schema_template))
          except etree.XMLSchemaError as e1:
            log.warn(“error: error validating routing instance %s” % vpn[‘service_name’])
            print “   - audit results: FAIL”
            print e1.message

          except Exception as e:
            log.warn(“error validating routing instance %s” % vpn[‘service_name’])
            print e

    else:
      print “\nerror: could not find vpn %s in the router %s” % (vpn[‘service_name’], vmx1_hostname)

if __name__ == “__main__”:
  main()
```