

Programmability and Automation

with Cisco Open NX-OS

Brenden Buresh
Brian Daugherty
Cesar Obediente
Errol Roberts
Jason Pfeifer
Kenny Garreau
Lukas Krattiger
Ranga Rao
Shane Corban
Todd Escalona



Programmability and Automation

with Cisco Open NX-OS



cisco.com

Preface

Authors and Contributors

This book represents a collaborative effort between Technical Marketing and Sales Engineers during a week-long intensive session at Cisco Headquarters Building 17 in San Jose, CA.

Authors

Brenden Buresh - Systems Engineering
Brian Daugherty - Systems Engineering
Cesar Obediente - Systems Engineering
Errol Roberts - Systems Engineering
Jason Pfeifer - Systems Engineering
Kenny Garreau - Systems Engineering
Lukas Krattiger - Technical Marketing
Ranga Rao - Technical Marketing
Shane Corban - Product Management
Todd Escalona - Systems Engineering

Contributors

Sanjaya Choudhury
Nicolas Delecroix
Parag Deshpande
Sam Henderson
Vishal Jain
Devarshi Shah

Acknowledgments

A special thanks to Cisco's Insieme BU Executive, Technical Marketing and Engineering teams, who supported the realization of this book. Thanks to Yousuf Khan, Joe Onisick, Jim Pisano, James Christopher and Matt Smorto for supporting this effort. Thanks to Cisco Sales Leadership for supporting the group of individual contributors who have dedicated their time authoring this book. Additionally, Cisco's CSG Engineering and QA Teams are acknowledged for developing some of the code and features addressed in this publication.

We would also like to thank Cynthia Broderick for her exceptional resource organization and support throughout our journey.

We are also genuinely appreciative to our Book Sprint (www.booksprints.net) team:

Adam Hyde (Founder)

Laia Ros (Facilitator)

Henrik van Leeuwen (Illustrator)

Raewyn Whyte (Proof Reader)

Juan Carlos Gutiérrez Barquero and Julien Taquet (Technical Support)

Laia and the team created an enabling environment that allowed us to exercise our social and technical skills to produce a quality publication.

Table of Contents

Introduction

Introduction	11
Organization of this Book.....	13
Expected Audience	15
Book Writing Methodology	17

Open NX-OS and Linux

Introduction	21
Cisco Nexus Switch as a Linux Device	23
Linux Containers and the Guest Shell.....	47
Open NX-OS Architecture.....	59
Third-party Application Integration.....	69

Network Programmability Fundamentals

Introduction	79
Conventional Network Interfaces.....	81
Programmable Network Elements	85
NX-API CLI	91

Model Driven Programming

Introduction	99
Model-driven Programming	101
Cisco Open NX-OS MDP Architecture.....	105
REST API Primer	117
Cisco NX-API REST Interface	127
Cisco NX-API WebSocket Notifications.....	137

Configuration Management and Automation

Introduction	143
Device Power-On Automation	145
Configuration and Lifecycle Management	157
IT Automation Tools.....	165

Practical Applications of Network Programmability

Introduction	177
Infrastructure Provisioning Automation	179
Automating Access Configuration with Ansible	189
Workload On-Boarding.....	195
Infrastructure as Code.....	205
Troubleshooting with Linux Capabilities	211
Network Monitoring with Splunk	215
Network Monitoring with Open Source Tools.....	221
Automating Network Auditing and Compliance	227
Automated Network Topology Verification.....	237
Workload Mobility and Correlation	243
Network Resiliency.....	249

Programmability Tools for Network Engineers

Introduction	255
Languages and Environments.....	257
Development and Testing Tools.....	261
Source Code and Version Control	273
Cisco DevNet for Open NX-OS	275
Learning and Helpful Resources	283

Introduction

Introduction

DevOps. Programmability. Automation.

These concepts have become a central and pervasive theme in many areas of information technology. What does it all mean to the world of network infrastructure?

Automated workflows and virtualization technologies have led to dramatic improvements in data center scale, agility, and efficiency. Application developers, server administrators, and Cloud and DevOps teams have been utilizing the processes and tools around automation for many years, resulting in streamlined and less error prone workflows. These teams are able to keep up with the speed of business requirements and market transitions due to modern workflows. Leveraging open development frameworks has been essential for innovation.

Why not leverage these concepts for the network, whose management methods are still dominated by human-to-machine interfaces?

Enter Open NX-OS on the Cisco Nexus platform, a rich software suite built on a Linux foundation that exposes APIs, data models, and programmatic constructs. Using Application Programmatic Interfaces (APIs) and configuration agents, operators can affect configuration changes in a more programmatic way.

This book explores Open NX-OS and many of the tools and options it provides. The chapters below examine the drivers for network automation, the fundamental supporting technologies, and the many new capabilities now available to network infrastructures. Real-world use cases are provided that can be immediately utilized for a successful transition to more efficient, safer, repeatable operations.

The Open NX-OS features and functionality discussed within were first introduced on the Cisco Nexus 9000 and Nexus 3000 Series Switches beginning with NX-OS Software Release 7.0(3).

Organization of this Book

Introduction

Within the introduction, we provide an initial walkthrough of the sections and chapters of this book. We try to highlight some important industry trends like the emergence of highly distributed applications and the adoption of Cloud and DevOps methodologies that are driving new paradigms in the network.

Open NX-OS and Linux

This primary section emphasizes how Open NX-OS based networking devices expose the full capability of the Linux operating system for end-users to utilize. Readers can learn how standard Linux tools like *ifconfig*, *ethtool*, *route*, *tcpdump* can be used to manage a Cisco Nexus Switch. Further, we help readers understand how they can extend the functionality of their switch with their own RPMs and containers, and unlock innovative new use cases.

Network Programmability Fundamentals

This section provides an overview of the evolution of interfaces on networking devices from being human-centric to being programmability-friendly. It introduces the readers to some easy-to-use programmatic interfaces like NX-API CLI and helps them get started down the path of programmability.

Model-Driven Programming

This section explores the advantages of a model-driven approach to programmability, and highlights the powerful, new capabilities in Cisco NX-API REST, being a data model-backed RESTful API, brings to the table.

Configuration Management and Automation

Infrastructure and network automation, driven by programmability, is a key enabler for the DevOps transformation. This chapter highlights the broad set of tools, features and capabilities that Open NX-OS incorporates to enable automation. The discussion covers integration with modern config management tools like Puppet, Chef and Ansible.

Practical Applications of Network Programmability

This section shifts the focus from description of network programmability and automation technologies to practical applications of these technologies. Although not exhaustive, the use-cases showcase real solutions intended to spark ideas for new innovative deployments. Each use-case is presented in the following format:

- Problem Statement - Overview of issues to be addressed
- Solution - Summarizes components used within exemplified solution
- Solution Approach - Utilized tools and enabled NX-OS capability
- Conclusion - Outcomes of exemplified solution

Programmability Tools for Network Engineers

This section explores essential tools for network programming as well as the underlying languages and environments. An introduction to Cisco DevNet for Open NX-OS is provided for developers and users to explore the capabilities of Open NX-OS.

Expected Audience

The intended audience for this book are those persons with a general need to understand how to utilize network programmability and unleash the power behind the Open NX-OS architecture. While interested development and IT practitioners may reap the most benefits from this content, the information and examples included within this book may be of use for every intermediate to advanced network professional interested in programmable fabrics.

Elements in this book explore topics beyond the traditional responsibility of a network administrator. The modular and extensible framework of the Open NX-OS modular architecture is not only discussed, but also exemplified through the ability to programmatically configure, operate, and manage Cisco Nexus switches as Linux servers and beyond.

Book Writing Methodology

Ten In, One Out: Ten individually-selected highly-skilled professionals from diverse backgrounds accepted the challenge to duel thoughts over the course of five days. Figuring out how to harness the brain power and collaborate effectively seemed to be nearly impossible, however opposites attracted and the team persisted through the hurdles. The Book Sprint (www.booksprints.net) methodology captured each of our strengths, fostered a team-oriented environment and accelerated the overall time to completion. The assembled group leveraged their near two hundred years of experience and a thousand hours of diligent authorship which resulted in this publication.

Open NX-OS and Linux

Introduction

Cisco NX-OS is the network operating system (OS) that powers Cisco Nexus switches across thousands of customer environments. It was the first data center network operating system to be built with Linux. While Cisco NX-OS has always been powered by Linux, under the hood, it hasn't, till recently, exposed many of the Linux capabilities to end-users.

With the latest release of Cisco NX-OS, termed Open NX-OS, on the Cisco Nexus 9000 and Nexus 3000 Series switches, Cisco has exposed the full power of the underlying Linux operating system for end-users to utilize. In addition, Cisco has built numerous extensions that make it possible to access these capabilities with the appropriate level of security and protection desired by the specific user.

This chapter will explore the Linux underpinnings of Open NX-OS.

Cisco Nexus Switch as a Linux Device

Open NX-OS Linux

This chapter will introduce some of the Linux capabilities of Open NX-OS including:

- **Kernel 3.4:** At the core of Cisco's Open NX-OS is a 64-bit Linux kernel based on version 3.4. This kernel provides a balance of features, maturity, and stability; and serves as a solid foundation for programmability and Linux-based management of a Cisco Nexus switch.
- **Kernel Stack:** Cisco Open NX-OS leverages the native Linux networking stack, instead of a custom-built userspace stack (NetStack) that was used in prior versions of NX-OS. Nexus switch interfaces, including physical, port-channel, vPC, VLAN and other logical interfaces, are mapped to the kernel as standard Linux netdevs. VRFs on the switch map to standard Linux namespaces.
- **Open Package Management:** Open NX-OS leverages standard package management tools, such as RPM and yum for software management. The same tools can be used for Open NX-OS process-patching or installing external or custom-developed programs onto the switch.
- **Container support:** Open NX-OS supports running Linux Containers (LXC) directly on the platform, and provide access to Centos 7 based Guest Shell. This allows customers and third-party application developers to add custom functionality directly on the device in a secure, isolated environment.

In addition, Open NX-OS continues to uphold some of the Linux best practice capabilities that have always been part of NX-OS:

- **Modularity:** Modules are loaded into the kernel only when needed. Modules can be loaded and unloaded on demand.
- **Fault Isolation:** Provides complete process isolation for NX-OS features, services and user application processes.
- **Resiliency:** Graceful restart or initialization of processes following unexpected exit conditions (segfault, panic).

Open NX-OS is based on Wind River Linux 5. By leveraging a standard and unmodified Linux foundation, it is possible to run any standard Linux-based application without changes or wrapper libraries. Users can leverage their standard Linux server management tools and workflows to install their custom-developed Linux-based applications, or other standard open source programs, and have them function "out of the box" on the Nexus switch. It is straightforward to integrate common third-party configuration management agents like Puppet, Chef, and telemetry applications such as ganglia, splunk, collector, nagios on the switch.

Linux Kernel Stack

One of the core capabilities of Open NX-OS is the ability to expose all interfaces on the device, including front panel switching ports, as Linux netdevices, which enables:

- **Linux utilities for Interface Management:** Leverage standard Linux utilities like ifconfig, ethtool, route, etc to manage network interfaces, routing and associated parameters.
- **Linux tools for troubleshooting:** Leverage tools like tcpdump, ping, traceroute, etc to troubleshoot network issues.
- **VRF capabilities with Namespaces:** Each VRF created within Open NX-OS will have a corresponding namespace in Linux associated with it, maintaining the VRF isolation extends from Open NX-OS to the Linux Kernel
- **Linux socket communications:** Open NX-OS and user applications use the Linux kernel's networking stack to send and receive packets to/from the external world. This enables applications that leverage standard Linux sockets, such as, agents and monitoring tools, to work without custom compilation.

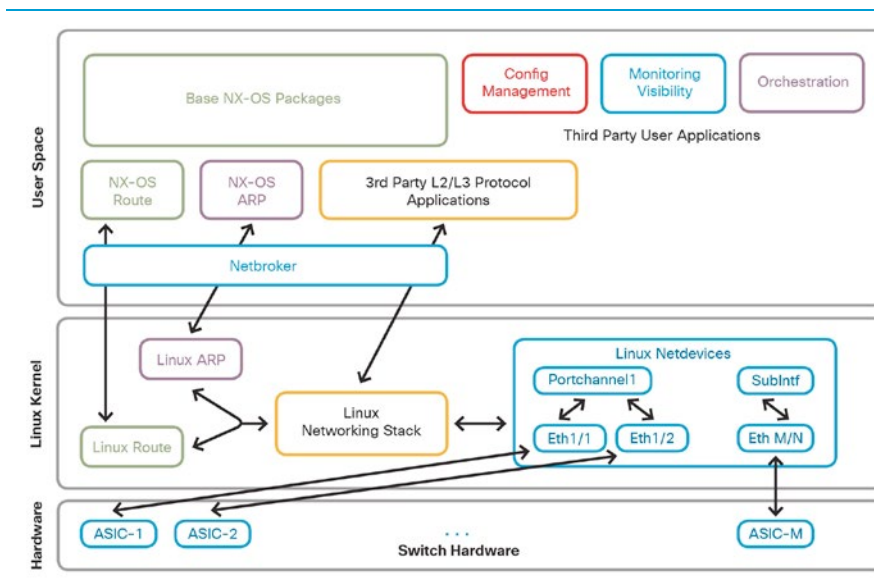
Open NX-OS Linux Network Architecture

Our Open NX-OS Linux Network Architecture is made up of two primary layers:

- User space processes and software -> traditional NX-OS software processes (ospf, vpc, bgp, nxos arp, vpc), third-party user applications (configuration management, visibility/analytics, custom built agents/tools)
- 64 Bit Linux 3.4.10 Kernel Layer -> linux kernel netdevices, linux networking stack(route, arp tables)

What has been exposed in Open NX-OS network architecture is access to the linux kernel networking stack, where the switch physical and logical interfaces have representation as a net device and an IP address in the kernel layer. This design opens the door to management of the routing and front panel ports using unmodified linux based tools and applications. However

there needs to be a synchronization function between NX-OS and the linux kernel layer, to ensure the two layers work effectively in tandem. This synchronization function between userspace NX-OS processes and kernel layer is provided by the netbroker module, which ensures changes implemented to physical and logical interfaces in NX-OS are reflected correctly to the linux netdevice interfaces. When NX-OS routing applications/processes like BGP program routes, they program these routes directly in the NX-OS route table, which pushes it to the linux kernel route table. Similarly if a route is installed at the linux kernel layer, the netbroker module checks the validity of the route addition by forwarding to the NX-OS Routing Information Base process, which then programs the route table in the hardware table if it's deemed valid. In the architecture, VRF's are implemented using linux network namespaces. Network namespaces are a natural fit as they provide the same isolation capabilities as VRFs. A kernel net device is associated with one and only one network namespace and the routing and ARP tables are local to a network namespace such that tasks running in the namespace see only the resources assigned to the namespace. Namespaces are covered in detail in a subsequent section.



Linux Shell Tools

Shell Access

Cisco Nexus switches support access to the Bourne-Again SHell (Bash). Bash interprets commands that you enter or commands that are read from a shell script. Using bash enables access to the underlying Linux system on the device and to manage the system.

Access to the bash shell and Linux is controlled via feature enablement on the Nexus platform. You must explicitly enable `feature bash-shell` to gain access to bash as user admin, which is by default part of DevOps role on the switch.

```
n9k-sw-1# show role name dev-ops

Role: dev-ops
Description: Predefined system role for devops access. This role
cannot be modified.
-----
Rule      Perm   Type      Scope      Entity
-----
6         permit command   conf t ; username *
5         permit command   attach module *
4         permit command   slot *
3         permit command   bcm module *
2         permit command   run bash *
1         permit command   python *
```

Enabling and Accessing Bash

```
n9k-sw-1(config)# do show feature | grep bash
bash-shell          1          disabled
n9k-sw-1(config)# feature bash-shell
n9k-sw-1(config)# do show feature | grep bash
bash-shell          1          enabled

n9k-sw-1# run bash
bash-4.2$ id
uid=2002(admin) gid=503(network-admin) groups=503(network-admin)
```

By default you are still user admin. In order to install third-party agents on the switch, you are required to be root user and either utilize `sudo root` to enable the agent to be installed in the filesystem, or authenticate as root within bash using `su - root`.

```
bash-4.2$ yum install puppet
Loaded plugins: downloadonly, importpubkey, localrpmDB, patchaction, patching,
               : protect-packages
You need to be root to perform this command.
```

Other capabilities available within bash are covered in the other sections of this chapter, such as manipulating, configuring and monitoring the switch in Linux, agent installation, etc. Please refer to these specific sections for further information.

From within bash, you also have the capability to execute NX-OS commands using our virtual shell utility (vsh). This enables the use of bash utilities (sed, grep, awk) to parse output to produce proper formatting.

Virtual shell utility example - Provisioning a new tenant on the network

```

Ensure following features are enabled on the switch for this particular example:
n93k-sw-1# show feature | include bash
bash-shell          1          enabled
n93k-sw-1# show feature | include interface-vlan
interface-vlan     1          enabled
Go to bash shell  n93k-sw-1# run bash
Switch to user root  bash-4.2$ su - root
Password:
Switch to management namespace in Linux&# root@n93k-sw-1#ip netns exec management bash
Verify current configuration of tenant interface eth2/4  root@n93k-sw-1#vsh -c "show
interface Eth2/4 brief"
-----
Ethernet          VLAN    Type Mode   Status Reason          Speed   Port
Interface                                     Ch #
-----
Eth2/4            1      eth  access up    none          40G(D)  --
Configure tenant vlan 200, SVI, and assign provision tenant port"
root@n93k-sw-1#vsh -c "config terminal ; vlan 200 ; name TenantA ; exit"
root@n93k-sw-1#vsh -c "config terminal ; interface vlan 200 ; no shutdown ; exit"
root@n93k-sw-1#vsh -c "configure terminal ; interface eth2/4 ; switchport access vlan 200 ;
no shutdown"
Verify tenant is configured correctly in the network
root@n93k-sw-1#vsh -c "show running-config interface Eth2/4"
!Command: show running-config interface Ethernet2/4
version 7.0(3)I2(1)
interface Ethernet2/4
    switchport access vlan 200
root@n93k-sw-1#vsh -c "show interface vlan 200 brief"

```

```

-----
Interface Secondary VLAN (Type)                Status Reason
-----
Vlan200    --                                up      --

```

The capabilities depicted above are simple examples to illustrate the flexibility of using bash for automation. These types of functions/examples could be combined and built into a bash-developed monitoring "agent" for your switch. For specifics on making agents/processes persistent in the native Linux shell, please refer to *Custom Developed Applications* section of the document.

Package Management Infrastructure

Open NX-OS provides support for standard package management infrastructure. It supports two possible hosting environments for installing packages:

- **Bash shell:** this is the native Open NX-OS Linux environment. It is disabled by default. To enable access, users must explicitly enable the bash shell feature on the switch.
- **Guest shell:** this is a secure Linux container environment running CentOS 7.

The focus of this section will be on managing packages in bash shell, or the native Linux environment where NX-OS runs. The guest shell environment will be covered in a subsequent section.

Yellowdog Updater, Modified (yum)

Yum is the default package and repository management tool for a number of operating systems, including Open NX-OS Linux. The yum package management infrastructure provides the following benefits:

- automatic resolution of software dependencies.
- easy to use Command Line Interface to install or upgrade software.
- yum can be configured to browse/search multiple software locations at one time for the existence of a specific package.
- ability to use either local (on box) or remote software repositories to install or upgrade software.

The yum client downloads software from repositories located on a local network or the Internet. RPM package files in these repositories are organized in a hierarchical manner so they can be found by the yum client.

From the command line, you can use the following subset of commands to interact with yum:

Command	Description
<code>yum install [package-name(s)]</code>	installs the specified package(s) along with any required dependencies.
<code>yum erase [package-name(s)]</code>	removes the specified package(s) from your system.
<code>yum search [search-pattern]</code>	searches the list of package names and descriptions for packages that match the search pattern and provides a list of package names along with architectures and a brief description of the package contents. Note that regular expression searches are not permitted.
<code>yum deplist [package-name]</code>	provides a listing of all of the libraries and modules that the named package depends on, along with the names of the packages (including versions) that provide those dependencies.
<code>yum check-update</code>	refreshes the local cache of the yum database so that dependency information and the latest packages are always up to date.
<code>yum info [package-name]</code>	provides the name, description of the package, as well as a link to the upstream home page for the software, release versions and the installed size of the software.
<code>yum reinstall [package-name(s)]</code>	erases and re-downloads a new copy of the package file and re-installs the software on your system.
<code>yum localinstall [local-rpm-file]</code>	checks the dependencies of a .rpm file and then installs it.
<code>yum update [optional-package-name]</code>	downloads and installs all updates including bug fixes, security releases, and upgrades, as provided by the distributors of your operating system. Note that you can specify package names with the update command.
<code>yum upgrade</code>	upgrades all packages installed on the system to the latest release.

Example: Installing Puppet Agent using yum

To install an agent or software package natively in Open NX-OS, users will need a routable connection to a software repository. This could be through any namespace that has connectivity externally (Default or Management).

The file located at `/etc/yum/yum.conf` provides system-wide configuration options for yum, as well as information about repositories. Specific repository information within Open NX-OS Linux is located in files ending in `.repo` under `/etc/yum/repos.d:` the repository to edit for pre-built third-party agents is `thirdparty.repo` on the switch. You will need to edit the `baseurl` field to point to your repository if you utilize one in your network.

Example for installing a software agent (Puppet) using yum:

```
bash-4.2$ cd /etc/yum/repos.d/

bash-4.2$ ls
groups.repo      nxos-extras.repo  nxos.repo.orig  thirdparty.repo
localrpmdb.repo  nxos.repo         patching.repo

bash-4.2$ more thirdparty.repo
[thirdparty]
name=Thirdparty RPM Database
baseurl=file:///bootflash/.rpmstore/thirdparty
enabled=1
gpgcheck=0
metadata_expire=0
cost=500

root@n9k-sw-1# rpm -qa | grep puppet
root@n9k-sw-1# Yum install puppet
Loaded plugins: downloadonly, importpubkey, localrpmDB, patchaction, patching,
               : protect-packages
groups-repo          | 1.1 kB   00:00 ...
localdb              | 951 B    00:00 ...
patching             | 951 B    00:00 ...
thirdparty           | 951 B    00:00 ...
Setting up Install Process
Resolving Dependencies
--> Running transaction check
```

```

---> Package puppet-agent.x86_64 0:1.1.0.153.g77189ea-1.nxos1 will be installed
--> Finished Dependency Resolution

Dependencies Resolved

=====
Package            Arch      Version                               Repository      Size
=====
Installing:
puppet-agent      x86_64    1.1.0.153.g77189ea-1.nxos1          thirdparty      37 M

Transaction Summary
=====
Install            1 Package

Total download size: 37 M
Installed size: 133 M
Is this ok [y/N]: y
Downloading Packages:
Running Transaction Check
Running Transaction Test
Transaction Test Succeeded
Running Transaction
   Installing : puppet-agent-1.1.0.153.g77189ea-1.nxos1.x86_64
1/1

Installed:
  puppet-agent.x86_64 0:1.1.0.153.g77189ea-1.nxos1

Complete!

```

Example for removing a software agent (Puppet) using yum:

```

root@n9k-sw-1# yum remove puppet
Loaded plugins: downloadonly, importpubkey, localrpmDB, patchaction, patching, protect-
packages
Setting up Remove Process
Resolving Dependencies

```



```

--> Running transaction check
---> Package puppet-agent.x86_64 0:1.1.0.153.g77189ea-1.nxos1 will be erased
--> Finished Dependency Resolution

Dependencies Resolved

=====
=====
Package                Arch          Version                               Repository
Size
=====
=====
Removing:
puppet-agent           x86_64        1.1.0.153.g77189ea-1.nxos1           @thirdparty
133 M

Transaction Summary

=====
=====
Remove                1 Package

Installed size: 133 M
Is this ok [y/N]: y
Downloading Packages:
Running Transaction Check
Running Transaction Test
Transaction Test Succeeded
Running Transaction

Erasing   : puppet-agent-1.1.0.153.g77189ea-1.nxos1.x86_64
1/1

Removed:
puppet-agent.x86_64 0:1.1.0.153.g77189ea-1.nxos1

Complete!

```

RedHat Package Manager (RPM)

In many ways, yum is simply a front end to a lower-level package management tool called rpm, similar to apt-get's relationship with dpkg. One key distinction to understand between the two utilities is that rpm does not perform dependency resolution.

The following commands should be run as root. The flags are expanded here in the pursuit of clarity, but the more conventional terse syntax is also included.

Command	Description
<pre>rpm --install --verbose --hash [local-rpm- file-name].rpm rpm -ivh [filename].rpm</pre>	<p>install an rpm from the file.</p> <p>rpm is also capable of installing RPM files from http and ftp sources as well as local files.</p>
<pre>rpm --erase [package-name] rpm -e</pre>	<p>remove the given package. Usually will not complete if [package-name] matches more than one package, but will remove more than one match if used with the --allmatches flag.</p>
<pre>rpm --query --all rpm -qa</pre>	<p>lists the name of all packages currently installed.</p>
<pre>rpm --query [package-name] rpm -q</pre>	<p>confirm or deny if a given package is installed in your system.</p>
<pre>rpm --query --info rpm -qi</pre>	<p>display the information about an installed package.</p>
<pre>rpm --query --list [package-name] rpm -ql</pre>	<p>generate a list of files installed by a given package.</p>
<pre>rpm --query --file rpm -q qf [file-name]</pre>	<p>check to see what installed package "owns" a given file.</p>

Example installing software agent using RPM (Puppet)

```
root@n9k-sw-1# rpm -qa | grep puppet
root@n9k-sw-1# rpm -ivh /bootflash/puppet-enterprise-3.7.1.rc2.6.g6cdc186-
1.pe.nxos.x86_64.rpm
```

```

Preparing...                               ##### [100%]
 1:puppet-enterprise                       ##### [100%]
root@n9k-sw-1# rpm -qa | grep puppet
puppet-enterprise-3.7.1.rc2.6.g6cdc186-1.pe.nxos.x86_64

```

Example removing software agent using RPM (Puppet)

```

root@n9k-sw-1# rpm -qa | grep puppet
puppet-enterprise-3.7.1.rc2.6.g6cdc186-1.pe.nxos.x86_64
root@n9k-sw-1# rpm -e puppet-enterprise-3.7.1.rc2.6.g6cdc186-1.pe.nxos.x86_64
root@n9k-sw-1# rpm -qa | grep puppet
root@n9k-sw-1#

```

Persistently Daemonizing a Third-party Process

Persistently starting your application from the Native Bash Shell

Your application should have a startup bash script that gets installed in `/etc/init.d/<application_name>`. This startup bash script should have the following general format :

- Install your application startup bash script that you created above into `/etc/init.d/<application_name>`
- Run your application with `/etc/init.d/<application_name> start`
- Run `chkconfig --add <application_name>`
- Run `chkconfig --level 3 <application_name> on`. `init` runlevel 3 is the standard multi-user runlevel in Open NX-OS.
- Verify that your application is scheduled to run on level 3 by running `chkconfig --list <application_name>` and confirm that level 3 is set to `on`
- Verify that your application is listed in `/etc/rc3.d`. You should see something similar to the following example, where there is an 'S' followed by a number, followed by your application name (tcollector in this example), and a link to your bash startup script in `../init.d/<application_name>`

```
bash-4.2# ls -l /etc/rc3.d/tcollector
lrwxrwxrwx 1 root root 20 Sep 25 22:56 /etc/rc3.d/S15tcollector -> ../init.d/tcollector
bash-4.2#
```

Full example: daemonizing an application

```
bash-4.2# cat /etc/init.d/hello.sh
#!/bin/bash
PIDFILE=/tmp/hello.pid
OUTPUTFILE=/tmp/hello
echo $$ > $PIDFILE
rm -f $OUTPUTFILE

while true
do
    echo $(date) >> $OUTPUTFILE
    echo 'Hello World' >> $OUTPUTFILE
    sleep 10
done
bash-4.2#
bash-4.2#
bash-4.2# cat /etc/init.d/hello

#!/bin/bash
#
# hello Trivial "hello world" example Third Party App
#
# chkconfig: 2345 15 85
# description: Trivial example Third Party App
#
### BEGIN INIT INFO
# Provides: hello
# Required-Start: $local_fs $remote_fs $network $named
# Required-Stop: $local_fs $remote_fs $network
# Description: Trivial example Third Party App
### END INIT INFO

PIDFILE=/tmp/hello.pid
```

```
# See how we were called.
case "$1" in
start)
    /etc/init.d/hello.sh &
    RETVAL=$?
;;
stop)
    kill -9 `cat $PIDFILE`
    RETVAL=$?
;;
status)
    ps -p `cat $PIDFILE`
    RETVAL=$?
;;
restart|force-reload|reload)
    kill -9 `cat $PIDFILE`
    /etc/init.d/hello.sh &
    RETVAL=$?
;;
*)
echo $"Usage: $prog {start|stop|status|restart|force-reload}"
RETVAL=2
esac
exit $RETVAL
bash-4.2#
bash-4.2# chkconfig --add hello
bash-4.2# chkconfig --level 3 hello on
bash-4.2# chkconfig --list hello
hello          0:off  1:off  2:on   3:on   4:on   5:on   6:off
bash-4.2# ls -al /etc/rc3.d/*hello*
lrwxrwxrwx 1 root root 15 Sep 27 18:00 /etc/rc3.d/S15hello -> ../init.d/hello
bash-4.2#
bash-4.2# reboot
```

Linux Networking

Netdevice

A netdevice (netdev) is a Linux kernel construct which represents a networking element. It can represent a physical interface like a front-end switch port, or a logical interface such as a tunnel. The netdev files on NX-OS exist under `/proc/net/dev` filesystem. The names for netdevices are similar to the NX-OS interface names. For example, Ethernet1/1 in NX-OS (port 1 in slot 1) refers to the corresponding Linux interface name of Eth1-1. It is important to note that interface names within Linux are limited to 15 characters, therefore 'Ethernet' is shortened to 'Eth'. This is consistent with the naming in `show interface brief` within NX-OS.

Using ifconfig on a Nexus Switch

Linux network utilities, commonly used by server admins, can now be used to configure, monitor, troubleshoot and manage the switch.

Using ifconfig to view interfaces:

```
root@n9k-sw-1# ifconfig -a | grep Eth
Eth1-1   Link encap:Ethernet  HWaddr 10:05:ca:f5:ee:98
Eth1-2   Link encap:Ethernet  HWaddr 10:05:ca:f5:ee:99
Eth1-3   Link encap:Ethernet  HWaddr 10:05:ca:f5:ee:9a
Eth1-4   Link encap:Ethernet  HWaddr 10:05:ca:f5:ee:9b
Eth1-5   Link encap:Ethernet  HWaddr 10:05:ca:f5:ee:9c
```

Using ifconfig to enable/disable an interface and verify in NX-OS

```
root@n9k-sw-1# ifconfig Eth2-4 down
root@n9k-sw-1# vsh -c "show interface Eth2/4"
Ethernet2/4 is down (Administratively down)
admin state is down, Dedicated Interface
  Hardware: 10000/40000 Ethernet, address: 6412.25ed.787f (bia 6412.25ed.787f)
  MTU 1500 bytes, BW 40000000 Kbit, DLY 10 usec
```

Notice above `vsh` was used to run a NX-OS CLI command. When working within the Bash Shell, the `vsh` command can be used to interact with the NX-OS parser.

Assigning a Layer 3 Address to an interface in Linux:

To maintain consistency between NX-OS and Linux, there are validation checks that are included for several operations. As an example, if you attempt to assign a Layer 3 address to a Layer 2 interface in Linux or via a third-party application, an error will be presented.

Error addressing Layer 2 interface

```
root@n9k-sw-1# vsh -c "show run int Eth2/4"

interface Ethernet2/4
switchport

root@n9k-sw-1# ifconfig Eth2-4 192.168.2.2 netmask 255.255.255.0
SIOCSIFADDR: Cannot assign requested address
SIOCSIFNETMASK: Cannot assign requested address
```

If you change the interface mode to Layer 3 within NX-OS you can now assign a Layer-3 address in Linux:

Assign a Layer 3 address

```
root@n9k-sw-1# ip netns exec default bash
root@n9k-sw-1# vsh -c "config terminal ; interface Eth2/4 ; no switchport ; no shutdown"&#101
root@n9k-sw-1# ifconfig Eth2-4 192.168.2.2 netmask 255.255.255.0
root@n9k-sw-1# ifconfig Eth2-4
Eth2-4    Link encap:Ethernet  HWaddr 10:05:ca:f5:ee:97
          inet addr:192.168.2.2  Bcast:192.168.2.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1837767 errors:0 dropped:1837763 overruns:0 frame:0
          TX packets:70576 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:95625447 (91.1 MiB)  TX bytes:17898072 (17.0 MiB)

root@n9k-sw-1# vsh -c "show run int Eth2/4"

interface Ethernet2/4
no switchport
```

```
ip address 192.168.2.2/24
no shutdown
```

Using ethtool on a Nexus Switch

Ethtool is a useful utility to view driver level interface statistics. This can be used to get information about front panel interfaces. An example is shown here, which gathers port statistics for the Ethernet2/4 interface.

Interface port statistics

```
root@n9k-sw-1# ethtool -S Eth2-4
NIC statistics:
    speed: 40000
    port_delay: 10
    port_bandwidth: 40000000
    admin_status: 1
    oper_status: 1
    port_mode: 0
    reset_counter: 6
    load-interval-1: 30
    rx_bit_rate1: 256
    rx_pkt_rate1: 0
    tx_bit_rate1: 144
    tx_pkt_rate1: 0
    load-interval-2: 300
    rx_bit_rate2: 248
```

Using tcpdump on a Nexus Switch

Tcpdump is a packet analyzer utility which can be run directly from the command line in the default namespace for a front panel interface. The example below uses tcpdump to monitor packets on an interface and store them in a pcap (packet capture) file named file.pcap. Use the `-i` flag to specify an interface.

tcpdump packet capture

```

root@n9k-sw-1# tcpdump -i Eth2-4 -w /bootflash/file.pcap
tcpdump: WARNING: Eth2-4: no IPv4 address assigned
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on Eth2-4, link-type EN10MB (Ethernet), capture size 65535 bytes
23:06:54.402365 STP 802.1w, Rapid STP, Flags [Learn, Forward], bridge-id
8001.84:b8:02:0e:f8:3b.8031, length 43
23:06:56.402207 STP 802.1w, Rapid STP, Flags [Learn, Forward], bridge-id
8001.84:b8:02:0e:f8:3b.8031, length 43

```

Using route commands on a Nexus Switch

Managing Routing Within Linux

Routing can be configured within Open NX-OS Linux. Static routes can be added and deleted directly using the `route` command. Any routing changes made will be immediately reflected within the NX-OS routing tables.

Set an IP address on e1/1

```

n9k-sw-1(config)# int eth2/4
n9k-sw-1(config-if)# ip address 192.168.1.2/24

```

Ensure you are root user and in default namespace to configure frontpanel interfaces

```

n93k-1-pm# run bash
bash-4.2$ id
uid=2002(admin) gid=503(network-admin) groups=503(network-admin)
bash-4.2$ su - root
Password:
root@n93k-1-pm#ip netns exec default bash

```

The following routing changes are made completely in bash.

Display routes

```

root@n9k-sw-1# route
Kernel IP routing table
Destination          Gateway              Genmask             Flags Metric Ref
Use Iface
192.168.1.0          *                   255.255.255.0      U           0       0       0 Eth2-4
127.1.1.0.0          *                   255.255.0.0        U           0       0       0 veobc
127.1.1.2.0          *                   255.255.255.0      U           0       0       0 veobc

```

Add a route

```

root@n9k-sw-1# ip route add 192.168.3.0/24 via 192.168.1.2 dev Eth2-4

```

Display updated route table

```

root@n9k-sw-1# route
Kernel IP routing table
Destination          Gateway              Genmask             Flags Metric Ref
Use Iface
192.168.1.0          *                   255.255.255.0      U           0       0       0 Eth2-4
192.168.3.0          192.168.1.2        255.255.255.0      UG          0       0       0 Eth2-4
127.1.1.0.0          *                   255.255.0.0        U           0       0       0 veobc
127.1.1.2.0          *                   255.255.255.0      U           0       0       0 veobc

```

Verify in NX-OS

```

n9k-sw-1# sh run | i "ip route"      ip route 192.168.2.0/24 Ethernet1/192.168.1.2 1
n9k-sw-1# sh ip route
192.168.1.0/24, ubest/mbest: 1/0, attached      *via 192.168.1.1, Eth1/1, [0/0], 00:02:08,
direct
192.168.1.2/32, ubest/mbest: 1/0, attached      *via 192.168.1.1, Eth1/1, [0/0], 00:02:08,
local
192.168.2.0/24, ubest/mbest: 1/0      *via 192.168.1.2, Eth1/1, [1/0], 00:01:37, static
n9k-sw-1#

```

Remove a route

```

n9k-sw-1# ip route del 192.168.3.0/24 via 192.168.1.2 dev Eth2-4

```

Display updated route table

```

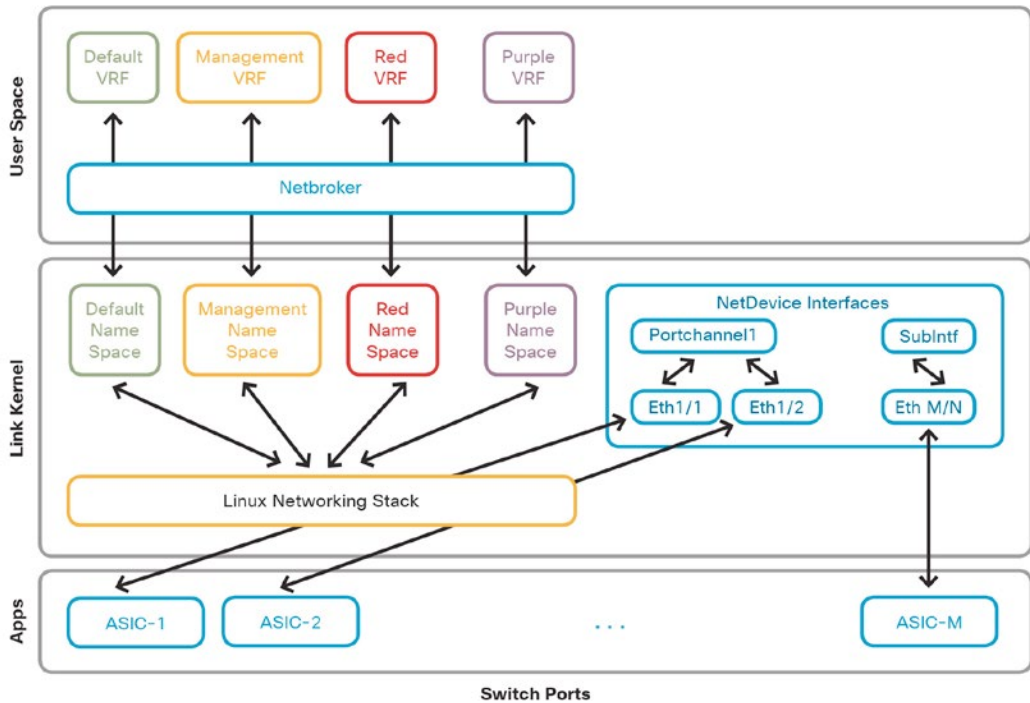
bash-4.2# route
Kernel IP routing table
Destination          Gateway             Genmask           Flags Metric Ref
Use Iface
192.168.1.0          *                  255.255.255.0    U           0       0       0 Eth1-1
127.1.1.0.0          *                  255.255.0.0      U           0       0       0 veobc
127.1.2.0            *                  255.255.255.0    U           0       0       0 veobc
bash-4.2#

```

Linux Network Namespaces and NX-OS Virtual Routing and Forwarding (VRF)

A single set of network interfaces and routing table entries are shared across the entire Linux operating system. Network namespaces virtualize these shared resources by providing different and separate instances of network interfaces and routing tables that operate independently of each other.

- Two namespaces are created by default in Linux - default and management. Each maps to VRFs of the same name within NX-OS.
- The default namespace (and VRF) enables access to the front panel ports and tunneling interfaces within Linux.
- The management namespace (and VRF) enables access to the management interface.
- Each new VRF created *within* NX-OS will map to a corresponding Linux namespace of the same name.

Figure: Open NX-OS Linux Network Architecture / Linux Namespace

Note: by default you are in the "default" namespace.

List available namespaces

```
root@n9k-sw-1# ip netns
management
default
```

List interfaces available in a namespace

```
root@n9k-sw-1# ip link list | grep Eth
88: Eth2-1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT qlen 100
89: Eth2-2: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT qlen 100
```

```
90: Eth2-3: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT qlen 100
91: Eth2-4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT
qlen 100
```

Run a process/command/shell within a given namespace

```
root@n9k-sw-1#ip netns exec <vrf name> <command>.

root@n9k-sw-1#ip netns exec management bash
```

Note: VRFs need to be created in NX-OS first to be represented and manipulated as namespaces in Linux.

Create a VRF in NX-OS and move an interface to this VRF

```
n9k-sw-1(config)# vrf context red
n9k-sw-1(config-vrf)#
n9k-sw-1(config)# int e2/4
n9k-sw-1(config-if)# vrf member red
Warning: Deleted all Layer-3 config on interface Ethernet2/4

n9k-sw-1(config-if)# ip address 192.168.1.2/24
n9k-sw-1(config-if)#
```

Check the list of namespaces

```
bash-4.2# ip netns list
red
management
default
```

Observe routing table differences between namespaces

```
bash-4.2# route
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref
Use Iface
127.1.0.0 * 255.255.0.0 U 0 0 0 veobc
```

```
127.1.1.2.0      *                255.255.255.0  U    0    0    0 veobc

bash-4.2# ip netns exec red route
Kernel IP routing table
Destination      Gateway         Genmask         Flags Metric Ref
Use Iface
192.168.1.0      *                255.255.255.0  U    0    0    0 Eth2-4

bash-4.2#
```

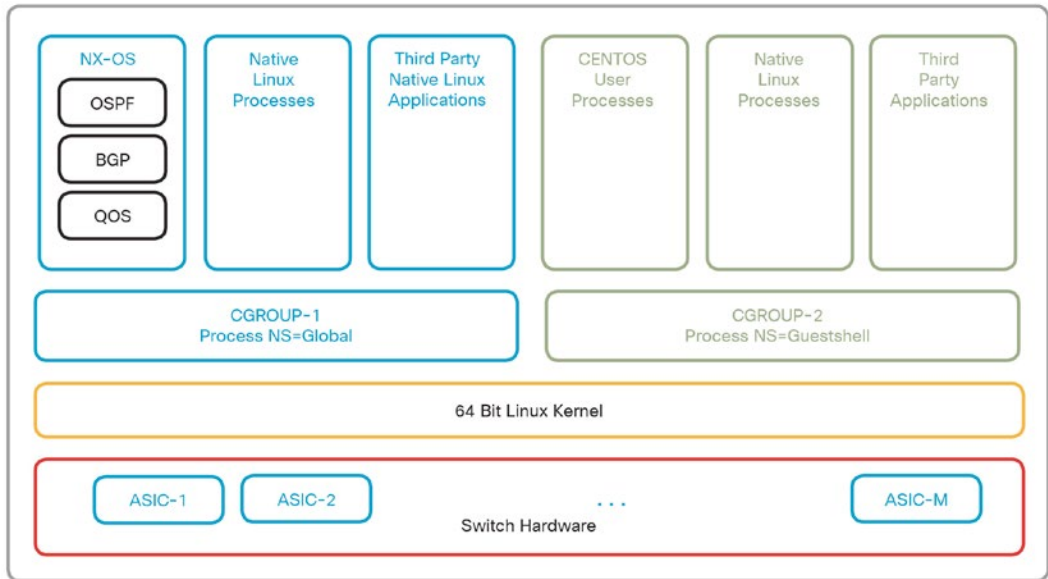
Cisco Open NX-OS is a Linux-based operating system that allows full Linux access to end-users. This includes enabling access and manageability of a Cisco Nexus 9000 and Nexus 3000 Series Switch via standard Linux tools. Further, Open NX-OS includes support for package managers that enable users to build and integrate open source or custom applications natively into NX-OS.

Linux Containers and the Guest Shell

Cisco Open NX-OS supports Linux containers (LXC) natively on the platform. This allows customers and third-party application developers the ability to add custom functionality directly to the switch, and host their applications on the device in a secure, isolated environment:

The benefits of utilizing containers include:

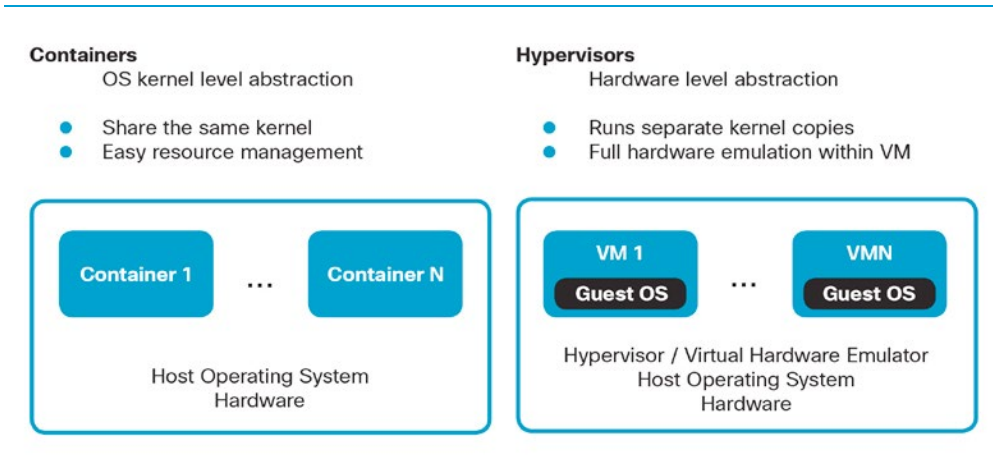
- Isolated and secure application execution environments
 - Resource and process isolation. Resources can be controlled, limiting the amount of CPU, memory, storage, and network bandwidth allocated to the container environment.
- Independent software release cycles
 - Customers and third-party developers can create their own open-source based custom applications, independent of the traditional NX-OS software release cycle.
 - Decreased time to market for custom-built features.
- Network Operation Optimization
 - Custom applications running within the container environment can reduce the need to use separate software solutions or interfaces. An example of this is a packet capture tool running locally on the device directly within the container, or applications that can monitor and control the switch.

Figure: Open NX-OS Container Architecture

The Linux kernel provides built-in resource controls (cgroups) which allow for the isolation and limiting of process, memory and filesystem resources. The isolation provided is transparent, so there is no process-scheduling overhead.

The processes within a container are given their own namespace which does not overlap with native NX-OS processes or any other containers that may be present on the system. This ensures that NX-OS processes are protected from the containerized guest processes. This separation is provided directly within the kernel.

It's worth mentioning that there are significant differences between containers and full hypervisor environments. Since they are implemented directly by the host kernel, only Linux containers can be supported. Even if the Linux distributions used within containers differ, they will still use the underlying host's kernel and libraries. Hypervisor-based virtual machines differ from containers in that each VM operates as an independent entity: there is no reliance on an up-stream kernel.

Figure: Differences in Container and Hypervisor-based Architectures

Guest Shell

The Open NX-OS platform presents a specialized container that is pre-built and installed within the system. This container environment is called the guest shell. The guest shell is based on CentOS 7.

To check if the guest shell is enabled on the system, the command "show virtual-service list" can be executed. The guest shell will be in "Activated" state if it is enabled:

```
n9k-sw-1# show virtual-service list
```

```
Virtual Service List:
```

Name	Status	Package Name
guestshell+	Activated	guestshell.ova

If the guest shell is not enabled, the command `guestshell enable` will activate the guest shell container on the system:

```
n9k-sw-1# guestshell enable
%% VDC-1 %% %VMAN-2-INSTALL_STATE: Installing virtual service 'guestshell+'
%% VDC-1 %% %VMAN-2-INSTALL_STATE: Install success virtual service 'guestshell+'; Activating
%% VDC-1 %% %VMAN-2-ACTIVATION_STATE: Activating virtual service 'guestshell+'
%% VDC-1 %% %VMAN-2-ACTIVATION_STATE: Successfully activated virtual service 'guestshell+'
```

To access the guest shell, enter `guestshell` on the NX-OS CLI.

```
n9k-sw-1# guestshell
[guestshell@guestshell ~]$
```

From the guest shell prompt, the user can run Linux commands:

```
[guestshell@guestshell ~]$ pwd
/home/guestshell
[guestshell@guestshell ~]$ ls
[guestshell@guestshell ~]$ whoami
guestshell
[guestshell@guestshell ~]$ ps -e | grep systemd
  1 ?          00:00:00 systemd
 10 ?          00:00:00 systemd-journal
 28 ?          00:00:00 systemd-logind
[guestshell@guestshell ~]$
```

The guest shell also provides direct access to the hosts bootflash on the switch. Files on `/bootflash` can be edited directly from within the guest shell environment. By default the guest shell comes with the vi editor pre-installed. More editors can be installed using yum or RPM to customize guest shell functionality to the shell of your choice.

Users within the guest shell can interact with the NX-OS host CLI to gather respective switch level information. The application `dohost` is provided to execute NX-OS CLI commands:

```
[guestshell@guestshell ~]$ dohost "show ip int brief vrf management"
IP Interface Status for VRF "management"(2)
Interface          IP Address          Interface Status
mgmt0              10.95.33.238        protocol-up/link-up/admin-up
```

The `dohost` command uses Unix domain sockets to facilitate the transfer of information between the guest shell and NX-OS host processes. Data retrieved from the `dohost` command can be used to take actions local to the network device within the guest shell. With this functionality, self-healing machines can be created. As an example, an application could be created in the Linux environment which captured the interface state periodically. When the interface state changes, the Linux application could be used to bring up a partner or backup interface.

The guest shell uses the default Virtual Routing and Forwarding (VRF) table for external connectivity. The application `chvrf` is provided for VRF management.

Usage of `chvrf`:

```
[guestshell@guestshell ~]$ chvrf
Usage: chvrf <vrf> [<cmd> ...]
```

Ping a host through the management VRF:

```
[guestshell@guestshell ~]$ chvrf management ping 10.70.42.150
PING 10.70.42.150 (10.70.42.150) 56(84) bytes of data.
 64 bytes from 10.70.42.150: icmp_seq=1 ttl=53 time=19.2 ms
 64 bytes from 10.70.42.150: icmp_seq=2 ttl=53 time=20.0 ms
```

Note: The `chvrf` command can be used in front of any command in the system to use the desired VRF.

The guest shell has been populated with common package managers. The yum package manager is installed, and will pull packages from the default CentOS 7 repositories. The locations of package repositories can be changed by modifying the ".repo" repository files in the `/etc/yum/repos.d` directory. The command `yum list available` will show all available packages in the repositories.

Installing the git client via yum, using the management VRF:

```
[guestshell@guestshell ~]$ sudo chvrf management yum install git
Loaded plugins: fastestmirror
base                               | 3.6 kB  00:00
```

```

extras                               | 3.4 kB  00:00
updates                               | 3.4 kB  00:00
(1/4): extras/7/x86_64/primary_db     | 87 kB  00:00
(2/4): base/7/x86_64/group_gz        | 154 kB  00:00
(3/4): updates/7/x86_64/primary_db   | 4.0 MB  00:03

...

Transaction Summary
=====
Install 1 Package (+34 Dependent packages)

Total download size: 17 M
Installed size: 63 M
Is this ok [y/d/N]:

```

You may need to increase the partition size of the guest shell, which is an option available to you from the host CLI using `guestshell resize`.

Resizing the rootfs of guest shell:

```

n9k-sw-1# guestshell resize rootfs 600
Note: Please disable/enable or reboot the Guest shell for root filesystem to be resized

```

In addition to the yum package manager, the Python package manager (*pip*) is also available from within the guest shell. Python packages are installed by pip into the default Python repository. In order to view a listing of installed packages, run the `pip freeze` command:

```

[guestshell@guestshell ~]$ sudo pip freeze
iniparse==0.4
pycurl==7.19.0
pygpme==0.3
pyliblzma==0.5.3
pyxattr==0.5.1
urlgrabber==3.10
yum-metadata-parser==1.1.4

```

From this example, we see that there are certain packages already installed such as Python curl (`pycurl`). A common package needed when working with Python and HTTP is the `requests` module.

The command listed below can be used to install the `requests` Python module:

```
guestshell@guestshell ~]$ sudo chvrf management pip --proxy=proxy.my.customer.com:8080
install requests
```

The command was executed as root to ensure we go through the management vrf using the `chvrf` command. In the event that the guest shell requires a proxy server for external HTTP connectivity, the `--proxy` option can be used.

You can now start Python and see that the `requests` module can be imported.

```
[guestshell@guestshell ~]$ python
Python 2.7.5 (default, Jun 17 2014, 18:11:42)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-16)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
>>>
```

The `pip freeze` command will also show that the `requests` module has been installed.

Additional programming languages can be installed within the guest shell if desired, is a user wants to utilize a different scripting language like perl. Users can install programming environments as needed through the yum package manager, or manually via RPM:

Running perl

```
[guestshell@guestshell ~]$ perl --version

This is perl 5, version 16, subversion 3 (v5.16.3) built for x86_64-linux-thread-multi
(with 28 registered patches, see perl -V for more detail)

Copyright 1987-2012, Larry Wall
```

Perl may be copied only under the terms of either the Artistic License or the GNU General Public License, which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be found on this system using "man perl" or "perldoc perl". If you have access to the Internet, point your browser at <http://www.perl.org/>, the Perl Home Page.

Perl files can be executed directly within the guest shell:

```
n9k-sw-1# guestshell
[guestshell@guestshell ~]$ ./test.pl
This is a perl script!
```

Programs and scripts can be executed in the guest shell directly from Open NX-OS using `guestshell run`.

```
[guestshell@guestshell ~]$ exit
logout
n9k-sw-1#
n9k-sw-1#
n9k-sw-1#
n9k-sw-1#
n9k-sw-1# guestshell run /home/guestshell/test.py
This is a Python script!
n9k-sw-1#
n9k-sw-1#
n9k-sw-1# guestshell run /home/guestshell/test.pl
This is a perl script!
n9k-sw-1#
```

Application Persistence within the Guest Shell Environment

Applications and scripts can be stopped and started automatically in the guest shell environment using `systemd`.

The below script named `/home/guestshell/datecap.sh` will save a file in the `/tmp` named datecap:

```
#!/bin/bash

OUTPUTFILE=/tmp/datecap

rm -f $OUTPUTFILE

while true
do
    echo $(date) >> $OUTPUTFILE
    echo 'Hello World' >> $OUTPUTFILE
    sleep 10
done
```

This script can be tied into the `systemd` infrastructure with the following file named `/usr/lib/systemd/system/datecap.service`:

```
[Unit]
Description=Trivial "datecap" example daemon

[Service]
ExecStart=/home/guestshell/datecap.sh &
Restart=always
RestartSec=10s

[Install]
WantedBy=multi-user.target
```

After creating the two files, the user can use `systemctl` to start and stop the `datecap` process:

```
[guestshell@guestshell tmp]sudo systemctl start datecap

[guestshell@guestshell tmp]$ sudo systemctl status -l datecap
datecap.service - Trivial "datecap" example daemon
   Loaded: loaded (/usr/lib/systemd/system/datecap.service; disabled)
   Active: active (running) since Wed 2015-09-30 02:52:00 UTC; 2min 28s ago
   Main PID: 131 (datecap.sh)
```



```

CGroup: /system.slice/datecap.service
      ??131 /bin/bash /home/guestshell/datecap.sh &
      ??164 sleep 10

Sep 30 02:52:00 guestshell systemd[1]: Started Trivial "datecap" example daemon.

[guestshell@guestshell tmp]$ sudo systemctl stop datecap

```

When the device is rebooted, `systemd` will automatically start the `datecap` daemon.

Working with Sockets in the Guest Shell

Linux sockets are available to applications running within the guest shell. Sockets can be used for communication between applications residing within the guest shell, as well as remote applications. An example would be an echo server, written in Python. This will echo back any text that it receives on a socket:

```

#!/usr/bin/env python

import socket

host = ''
port = 50000
backlog = 5
size = 1024
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host,port))
s.listen(backlog)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
while 1:
    client, address = s.accept()
    data = client.recv(size)
    if data:
        if (data == "done"):
            print "received a done message, exiting"
            client.send("server done, exiting")

```

```

        client.close()
    s.close()
    break
else :
    print data
    client.send(data)
client.close()
s.close()

```

The code above will use Python to create a socket stream and listen on port 50000. Any text it receives will be sent back to the sender. If `done` is entered the server will close its socket and exit.

This can be used with an echo client application, which can reside remotely. A sample echo client may look like the following:

```

#!/usr/bin/env python

import socket

host = '<ip address of echo server>'
port = 50000
size = 1024
while(True):
    test = raw_input('Enter Data to send to server: ')
    if (test == 'q') or (test == 'quit'):
        break
    else:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((host,port))
        s.send(test)
        data = s.recv(size)
        print 'Received:', data
        s.close()
print 'done'

```

The Python code above will take input from the command line through the `raw_input` call and send it to the echo server through the socket connection.

Here is a sample run of both the client and server:

Open NX-OS provides guest shell, a native LXC container for switch management and hosting applications. Other LXC containers can be created in Open NX-OS for third-party or custom applications.

Server side (Running inside of Guest Shell using the management VRF):

```
[guestshell@guestshell gs]$ sudo chvrf management Python echoserver.py
Hello Server
Please echo this text
received a done message, exiting
[guestshell@guestshell gs]$
```

Client side (Running on an external Linux Server):

```
./echoclient.py
Enter Data to send to server: Hello Server
Received: Hello Server
Enter Data to send to server: Please echo this text
Received: Please echo this text
Enter Data to send to server: done
Received: server done, exiting
Enter the Data to send to server: quit
done
```

The server can also accept multiple incoming sockets and below is an example where two clients have connected simultaneously:

```
[guestshell@guestshell gs]$ sudo chvrf management Python echoserver.py
hello from client 1
hello from client 2
```

Open NX-OS Architecture

An Extensible Network OS

The Cisco Open NX-OS software is designed to allow administrators to manage a switch such as a Linux device. The Open NX-OS software stack addresses a number of functional areas to address the needs of a DevOps-driven automation and programmability framework.

- **Auto Deployment (Bootstrap and Provisioning):** Cisco Open NX-OS supports a robust network bootstrapping and provisioning capability with Power-On Auto Provisioning (POAP). Open NX-OS can utilize Pre-boot eXecution Environment (PXE) to facilitate the boot process and initial configuration of a Nexus switch.
- **Extensibility:** Open NX-OS enables access to the Linux bash shell as well as the use of package managers. The user can install native RPMs and third-party applications running processes as they would on a Linux server. Supporting RPM-based packages provides the ability to load only the services or packages required. The level of extensibility in Open NX-OS ensures that third-party daemons and packages (such as routing protocols) can be supported. Third-party monitoring tools like `tcollector` are supported on the platform.
- **Open Interfaces:** Open NX-OS adds the ability to leverage Linux tools for configuration, monitoring and troubleshooting. Front panel ports of a switch can be manipulated as native Linux interfaces. Tools like `ifconfig` and `tcpdump` can be used as they would be in a server environment for troubleshooting and configuration.
- **Application Development (Adaptable SDK):** Open NX-OS provides a tool chain to build custom packages and agents. Open NX-OS has published an extensive SDK to enable a build environment that can be installed on a Linux server. This provides the ability to download a build agent that will incorporate the source code in the local directory structure. The SDK allows administrators to build and package binaries for use on Open NX-OS. Applications have two deployment options: they can be installed natively into the Linux filesystem, or deployed in an LXC container.

Flexible Programming Options

Open NX-OS provides flexibility in tool choice for programmability tools and languages.

- **Python Libraries:** There is embedded Python shell support with native Open NX-OS libraries that can be utilized for development.

- **NX-API REST** interacts with network elements through RESTful API calls. This allows for a data model-driven approach to network configuration and management.
- **NX-API CLI** provides the ability to embed NX-OS CLI commands in a structured data format (JSON or XML) for execution on the switch via an HTTP/HTTPS transport. The data returned from the calls will also be formatted in JSON/XML, making it easy to parse the data with modern programming languages. A sandbox environment (NX-API Developer Sandbox) also exists as a development tool, which is covered in *Section 7.3, Development and Testing Tools*.

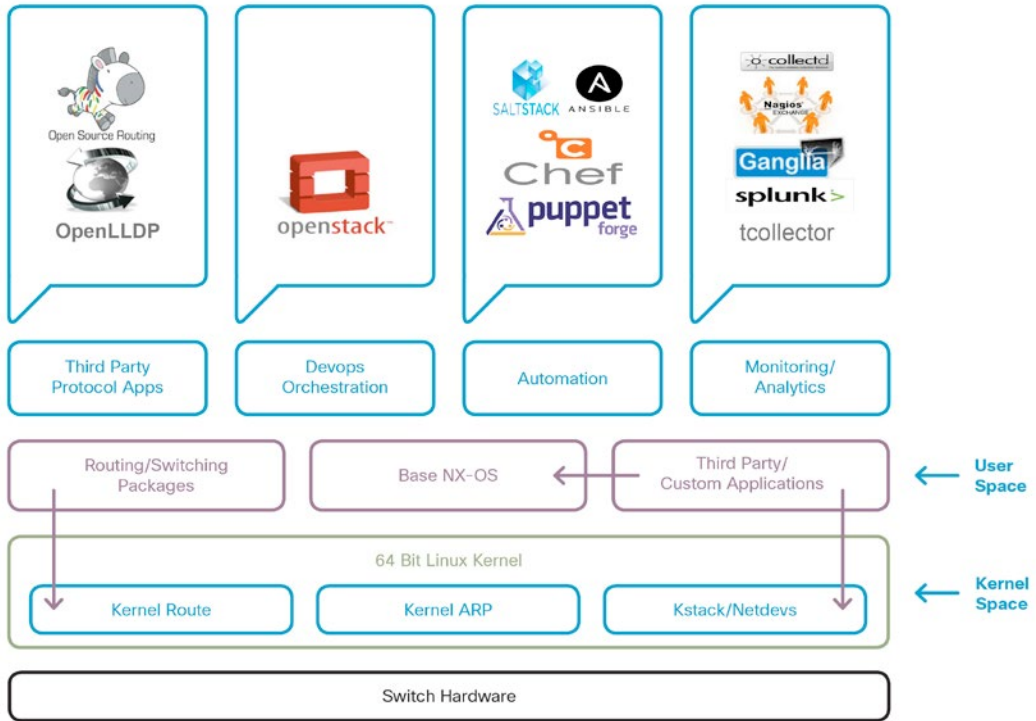
Support for Automation Tools

- **Configuration management tools** like Ansible, Chef and Puppet are orchestration engines that function with Open NX-OS.
- **Cloud Orchestration platforms** such as OpenStack integrate seamlessly with Neutron plugins.
- Cisco-sponsored and community-contributed **open source management tools** and packages are available on Github: <http://github.com/datacenter>

Integrating Third-party Applications

The Open NX-OS modular architecture provides a flexible framework to integrate third-party application software as depicted in the figure below.

Figure: Open NX-OS Third-party Application Integration - Software Architecture



The modular framework allows for the support of monitoring and analytics applications like `tcollector` and Splunk amongst others. The framework supports automation tools and agents, DevOps orchestration and third-party protocol support like openLLDP.

Supported third-party applications, open source tools and publish RPMs that integrate with Open NX-OS are all available at <http://developer.cisco.com/opennxos>

Open NX-OS Modular Architecture

Cisco Open NX-OS is a unique multi-process state-sharing architecture that separates an element's state from parent processes. This reflects Cisco's core software design philosophy and enables fault recovery and real-time software updates on a process-level basis without affecting the running state of the system.

Protocol routing and switching processes, security functions, management processes, and even device drivers are decoupled from the kernel. These modules and processes run in user space, not in kernel space, which ensures process control system stability. The modular nature of the system allows for the update and restart of individual switch processes without requiring a switch reload.

The same binary image of NX-OS can be deployed across any family of Nexus 9000 and Nexus 3000 Series Switches. This improves the feature compatibility across platforms and ensures consistency in defect resolution. It also makes it much simpler for users to deploy, certify and validate new releases in their data center environment, and makes code portable across the environment.

Process Isolation and Scheduling

The Linux kernel backing Cisco Open NX-OS is a multi-tasking kernel leveraging the Linux Completely Fair Process Scheduler. The process scheduler within the kernel coordinates which processes are allowed to run at any given time, scheduling CPU equally amongst all user and NX-OS system processes. By taking scheduling class/policy and process priorities into account to balance processes between multiple CPU cores in SMP systems, the CPU cycles are maintained in fair access to maintain system stability.

Shell Environment

Cisco Nexus switches support direct Bourne Again SHell (Bash) access. With Bash, you can access the underlying Linux system on the device to manage the system. Most importantly, by providing users unrestricted access to the Linux shell, users can now leverage data center automation tools, which can utilize bash scripting and Linux interfaces natively. Access to the bash shell is controlled through RBAC. Users who are able to gain access can write shell scripts which leverage the network in a similar fashion for other parts of the IT organization.

Process Patching

The Open NX-OS Linux kernel's process isolation allows patching and modification of software independent of the traditional Cisco software release cycles. Features and fixes can be delivered in a more agile fashion to the end user. Modifications to the system can be released to users in the form of patches which can be installed without the need to reload the device being patched. An example of this might be the installation of security fixes for packages such as OpenSSL or OpenSSH

Process Restartability

Processes within Open NX-OS can be restarted on-demand without affecting other processes, and will automatically be restarted in the event of an unexpected exit condition.

Process restart via NX-OS:

```
n9k-sw-1(config)# router bgp 65000
n9k-sw-1(config-router)# restart bgp 65000
%BGP-5-ADJCHANGE: bgp-65000 [9224] (default) neighbor 192.168.1.2 Up
```

For example, we can kill the BGP process and see that it is automatically restarted by NX-OS.

Automatic process restartability:

```
root      17073   5900   0 00:11 ?          00:00:00 /isan/bin/routing-sw/bgp -t 65000
admin     17137  17132   0 00:13 pts/2      00:00:00 grep bgp

bash-4.2$ sudo kill -9 17073
bash-4.2$ ps -ef | grep bgp
root      17221   5900  34 00:13 ?          00:00:01 /isan/bin/routing-sw/bgp -t 65000
admin     17258  17132   0 00:13 pts/2      00:00:00 grep bgp
bash-4.2$
```

Interactive Programmability with Python

In Open NX-OS, the Python programming language is supported directly on the device. An interactive Python interpreter is available just by typing `python` at the CLI prompt:

```
n9k-sw-1# python
Python 2.7.5 (default, Oct  8 2013, 23:59:43)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The Python interpreter on the Open NX-OS platform can also interpret script files directly, in this case the Python interpreter is running "non-interactively". To run non-interactively, users

can enter the name of the script to be run after the `python` command at the CLI prompt. Here is an example run of the script `hello_world.py`

```
n9k-sw-1# cd bootflash:
n9k-sw-1# python hello_world.py
hello world!
n9k-sw-1# show file hello_world.py
#!/usr/bin/python

print "hello world!"
n9k-sw-1#
```

For more information about Python, see *Section 7.2 Programmability Tools for Network Engineers*.

Cisco provides a set of helper library packages designed specifically for Python. These packages are pre-installed on the NX-OS platform and are named "cli" and "cisco".

The `cli` Python package is used to allow Python scripts running on the Open NX-OS device to interact with the CLI to get and set configuration on the device. This library has one function within it named `cli`. The input parameters to the function are the CLI commands the user desires to run, the output is a string representing the parser output from the CLI command.

Here is an example of using the CLI library to gather hardware version information

```
import cli

result = cli.cli("show version | beg Hardware")
print result
```

Running the example in Python results in the following output:

```
n9k-sw-1# python get_hardware.py
Hardware
cisco Nexus9000 C9396TX Chassis
Intel(R) Core(TM) i3-3227U CPU @ 2.50GHz with 16402136 kB of memory.
Processor Board ID SAL18370NXE
```

```

Device name: n9k-sw-1
bootflash: 51496280 kB
Kernel uptime is 98 day(s), 21 hour(s), 16 minute(s), 43 second(s)

Last reset at 180873 usecs after Tue Jun 23 18:15:50 2015

Reason: Reset Requested by CLI command reload
System version: 7.0(3)I2(1)
Service:

plugin
Core Plugin, Ethernet Plugin

Active Package(s):

```

The cli package will accept both show commands and configuration commands in the cli function. In addition to the cli package, a "cisco" package is provided. The cisco package provides the following functionality:

Function	Description
acl	Adds the ability to create, delete, and modify Access Control Lists
bgp	Functions around configuring BGP
cisco_secret	Adjust Cisco passwords
feature	Get information about supported and enabled features on NX-OS
interface	Functions around manipulating interfaces
nxcli	Contains useful CLI interaction utilities
ospf	Functions around configuring OSPF
ssh	Generate SSH key information
tacacs	Runs and parsers TACACS+ status information
vrf	Creates and deletes virtual routing and forwarding (VRF) tables

The following code sample uses the cisco package to change the state of all interfaces to up.

```
import cisco

tp = cisco.Interface.interfaces()
for tpint in tp:
    intf = cisco.Interface(tpint)
    intf.set_state()
```

The first line above imports the cisco package into the Python script so functions within the package can be used. A for loop is then used to loop through each interface and set_state(). set_state with no options will default to setting the state of the interface to up.

ASIC-level Shell Access

Cisco Nexus Series switches architecture enables access to the ASIC shell. You can use an ASIC shell to access a shell prompt specific to the device front panel and fabric module line cards.

The following examples describe how you can access the command-line Broadcom based shell (*bcm-shell*) and how to read from these ASICs, although it need not be limited to these ASICs.

Accessing the Broadcom Shell with the CLI (*bcm-shell*)

Syntax: `bcm-shell module module_number [instance_number:command]`

Query the Broadcom Shell for detailed VLAN information:

```
n9k-sw-1# sh vlan id 101

VLAN Name                               Status    Ports
-----
101  VLAN0101                               active    Po1, Po22, Eth1/1, Eth1/2
                                           Eth1/47, Eth1/48

VLAN Type      Vlan-mode
-----
-----
```

```

101 enet          CE

n9k-sw-1# bcm-shell module 1
Warning: BCM shell access should be used with caution
Entering bcm shell on module 1
Available Unit Numbers: 0
bcm-shell.0> vlan 101
Current settings:
    VRF=3
    OuterTPID=0x8100
    LearnDisable=1
    UnknownIp6McastToCpu=0
    UnknownIp4McastToCpu=1
    Ip4Disable=0
    Ip6Disable=0
    Ip4McastDisable=0
    Ip6McastDisable=1
    Ip4McastL2Disable=0
    Ip6McastL2Disable=0
    L3VrfGlobalDisable=0
    MplsDisable=0
    CosqEnable=0
    MiMTermDisable=0
    Cosq=-277706854
    Ip6McastFloodMode=MCAST_FLOOD_UNKNOWN
    Ip4McastFloodMode=MCAST_FLOOD_UNKNOWN
    L2McastFloodMode=MCAST_FLOOD_UNKNOWN
    IfClass=17
bcm-shell.0>

```

Query the Broadcom Shell for detailed MAC table information:

```

n9k-sw-1# show mac address-table
Legend:
    * - primary entry, G - Gateway MAC, (R) - Routed MAC, O - Overlay MAC
    age - seconds since last seen, + - primary entry using vPC Peer-Link,
    (T) - True, (F) - False
VLAN      MAC Address      Type      age      Secure NTFY Ports

```

```

-----+-----+-----+-----+-----+-----+-----
* 2501    5087.89d4.5495    static -      F      F      Vlan2501
* 2502    5087.89d4.5495    static -      F      F      Vlan2502
G   -     5087.89d4.5495    static -      F      F      sup-eth1 (R)
G   1     5087.89d4.5495    static -      F      F      sup-eth1 (R)
G  101    5087.89d4.5495    static -      F      F      sup-eth1 (R)
G  102    5087.89d4.5495    static -      F      F      sup-eth1 (R)
G 2501    5087.89d4.5495    static -      F      F      sup-eth1 (R)
G 2502    5087.89d4.5495    static -      F      F      sup-eth1 (R)
G  201    5087.89d4.5495    static -      F      F      sup-eth1 (R)

n9k-sw-1#
n9k-sw-1# bcm-shell module 1
Warning: BCM shell access should be used with caution
Entering bcm shell on module 1
Available Unit Numbers: 0
bcm-shell.0> l2 show
mac=50:87:89:d4:54:95 vlan=31174 GPORT=0x80000202 port=0x80000202 (vxlan) Static Hit
mac=50:87:89:d4:54:95 vlan=31173 GPORT=0x80000201 port=0x80000201 (vxlan) Static Hit
bcm-shell.0>

```

Summary

The Open NX-OS framework allows users to leverage the capabilities of an underlying Linux distribution. It allows the installation and management of a wide-variety of third party tools and applications that extend the capabilities of the switch, enabling powerful new use-cases. Open NX-OS is designed following the latest software development best practices and provides best of breed capabilities such as process isolation, restartability and modularity - characteristics that make the system more resilient.

Third-party Application Integration

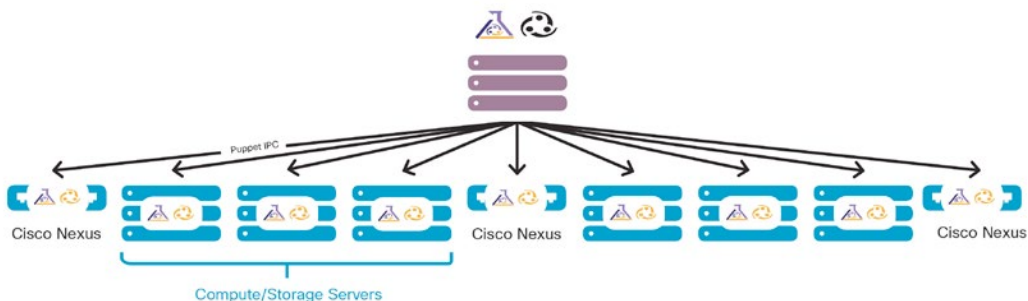
The Open NX-OS architecture allows users to expand base functionality on the Cisco Nexus switching platforms through the installation of compatible packages. Third-party applications or tools may address:

- Configuration Management
- Network Telemetry and Analytics
- Network Monitoring
- Custom Network Requirements

Configuration Management Tools

Cisco Open NX-OS supports intent-based automation through integration of agent software for Puppet and Chef. Automation of various network provisioning, configuration, and management tasks from a central server will enable a dramatic reduction in network deployment and configuration times, while eliminating manual tasks that are repetitive and error-prone.

Figure: Scalable Network Management with Configuration Management Tools

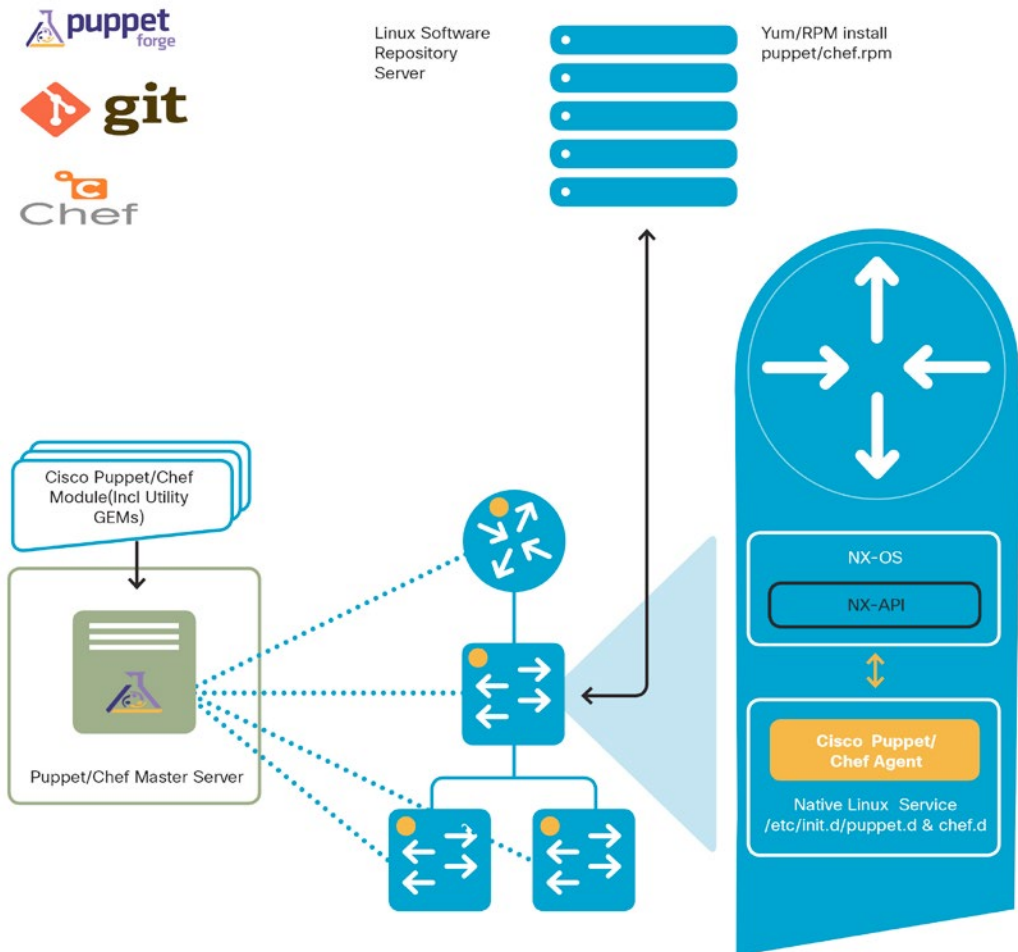


The provisioning of network constructs like VLANs, ports, network routes, quality of service (QoS) parameters, and access control can be optimized with automation tool integration.

Lifecycle management operations such as firmware and configuration management, compliance auditing, and performance monitoring are made substantially easier.

Puppet and Chef Agents

Figure: Using Standard Tools like YUM to install Puppet and Chef Agents



The graphic depicts the workflow for the Puppet/Chef agent support.

- The Puppet agent is installed via yum as an RPM.
- Configure agent to talk to proper server / master.
- User installs Chef cookbook / Puppet manifest as an example along with Cisco utility libraries on the server/master.
- User creates a recipe or defines a manifest using the resources available in the cookbook/module.
- Switch agent stays in sync with Puppet/Chef master for updated catalog/cookbooks.
 - If agent is configured to run periodically, it will obtain and download the cookbook/catalog and attempt to remediate the network element/switch to the desired state.
- The Puppet/Chef agent utilizes NX-API to apply changes as defined by the Puppet/Chef master server to the switch.

Figure: Types and Providers for Puppet and Chef supported out-of-the-box with Open NX-OS

Puppet Agent Types/Providers
cisco_vtp
cisco_tacacs_server
cisco_tacacs_server_host
cisco_snmp_server
cisco_snmp_community
cisco_snmp_group
cisco_ospf
cisco_ospf_vrf
cisco_vlan
cisco_bgp*
cisco_bgp_vrf*
cisco_interface
cisco_interface_ospf
cisco_interface_vlan

The graphic list samples supported agents / provider capabilities. The list can be extended, leveraging the puppet workflow via the utility classes. The user can develop extensions to Cisco Puppet modules on GitHub. The agent is also extensible by passing CLI commands.

```
cisco_command_config resource:
cisco_command_config { " feature-portchannel1":
command => " interface port channel1\n
           description nwk1-0106-ic4-gw1|Po2407\n
           no switchport\n
           ip address 10.1.1.51/31\n  }
```

Sample Puppet Manifest

The Puppet manifest for interface configuration sets Ethernet 1/1 with an IP address of 10.1.43.43/24.

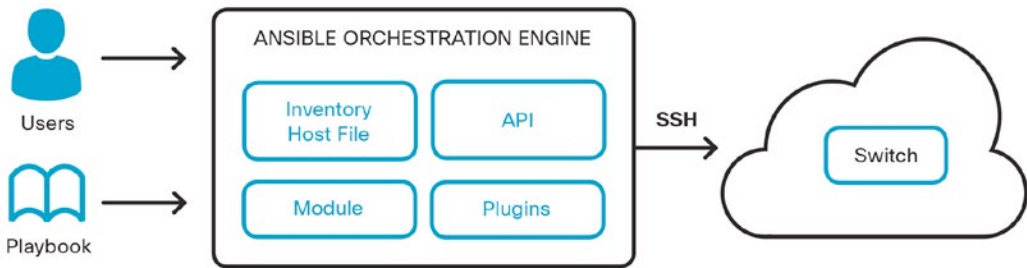
```
cisco_interface { "ethernet1/1" :
  shutdown          => false,
  description       => 'managed by puppet',
  switchport_mode  => disabled,
  ipv4_address      => '10.1.43.43',
  ipv4_netmask_length => 24,
}
```

Sample Puppet manifest for the installation of `tcollector`

```
class tcollector {
  package { ['tcollector']:
    ensure => present,
  }
  service { ["tcollector"]:
    ensure => running,
  }
}
```

Ansible

Ansible is an automation tool for cloud provisioning, configuration management and application deployment. It is agent-less, and uses the YAML markup language in the form of Ansible Playbooks.

Figure: Ansible Configuration Management Workflow

Here is a sample Ansible playbook that creates a static route on a NX-OS device. Within this sample a static route for 10.10.1.0/24 is created with a next hop of 10.20.2.2.

Sample Ansible Playbook

```
# Static route with tag and preference set
- nxos_static_routes: prefix=10.10.1.0/24 next_hop=10.20.2.2 tag=90 pref=80 host={{
inventory_hostname }}
```

Open NX-OS supports a range of configuration management tools like Puppet, Chef, and Ansible and is extensible to support Salt, CFEngine and others.

Telemetry Applications

Open NX-OS supports a wide range of third-party telemetry applications, and can support a pull or push model for telemetry data to be extracted from the devices.

Splunk

Splunk is a web-based data collection, analysis, and monitoring tool. Splunk Enterprise helps you gain valuable operational intelligence from your machine-generated data. It comes with a full range of powerful search, visualization and pre-packaged content for use-cases where any user can quickly discover and share insights. The raw data is sent to the Splunk server using the Splunk Universal Forwarder. Universal Forwarders provide reliable and secure data collection from remote sources while forwarding data into Splunk Enterprise for indexing and consolida-

tion. A Splunk Enterprise infrastructure can scale to tens of thousands of remote systems, collecting terabytes of data with minimal impact on performance.

For additional information, see http://www.splunk.com/en_us/download/universal-forwarder.html.

NX-OS with Splunk enables network operators to:

- Gain visibility into their infrastructure
- Track detailed network inventory
- Track power usage and temperature
- Authenticate and audit configuration changes
- Collect performance data from network devices

tcollector

tcollector is a client-side process that gathers data from local collectors and pushes the data to OpenTSDB. The tcollector agent is installed in the native Linux filesystem.

For additional information, see http://opentsdb.net/docs/build/html/user_guide/utilities/tcollector.html.

Collectd

collectd is a daemon which collects system performance statistics periodically and provides mechanisms to store the values in a variety of ways as for example in RRD files.

collectd gathers statistics about the system it is running on and stores this information. Statistics can be used for performance analysis and capacity planning.

For additional information, see <https://collectd.org>

Ganglia

Ganglia is a scalable, distributed monitoring system for high-performance computing systems such as clusters and grids. It is based on a hierarchical design targeted at federations of clus-

ters. It leverages widely used technologies such as XML for data representation, XDR for compact, portable data transport, and RRDtool for data storage and visualization.

For additional information, see <http://ganglia.info>.

Nagios

Nagios is an open source application which provides monitoring of network services (through ICMP, SNMP, SSH, FTP, HTTP etc), host resources (CPU load, disk usage, system logs, etc.) and notification for servers, switches, applications, and services. Nagios provides remote monitoring through the Nagios remote plugin executor (NRPE) and through SSH or SSL tunnels.

For more information, see <https://www.nagios.org>

Open Source Protocols

Open NX-OS comes packaged with a comprehensive suite of protocols. Customers may be interested in augmenting or replacing default modules and packages with custom-developed or open source modules. Examples of these modules could be OpenLLDP, OpenSSH, etc.

Open LLDP

LLDP is an industry standard protocol designed to supplement proprietary Link-Layer discovery protocols such as EDP or CDP. The goal of LLDP is to provide an inter-vendor compatible mechanism to deliver Link-Layer notifications to adjacent network devices.

For more information, see <https://vincentbernat.github.io/lldpd/index.html>

Custom Applications on Open NX-OS

Cisco SDK

The Cisco SDK is a development-kit based on Yocto 1.2. It contains all of the tools needed to build applications for native installation on a Cisco Nexus switch beginning with NX-OS Release 7.0(3)I2(1). The basic components are the C cross-compiler, linker, libraries, and header files that are commonly used in many applications. The SDK can be used to develop applications and

package them as RPMs to be installed on a switch. The SDK should be downloaded and installed on your Linux build server in your data center for your application development efforts.

Basic example of building an application into an RPM:

Essentially if the application successfully builds using "make", then it can be packaged into an RPM. The package should contain the meta data for use with autoconf and RPM packaging tools. Assuming all of the dependencies are met, you can just use the standard RPM build procedure:

```
bash$ mkdir rpm
bash$ cd rpm
bash$ mkdir BUILD RPMS SOURCES SPECS SRPMS
bash$ cp ../example-app-1.0.tgz SOURCES
bash$ cp ../example-app.spec SPECS
bash$ rpmbuild -v --bb SPECS/example-app.spec
```

The SDK is available for download at: <http://developer.cisco.com/opennxos> . This will also contain detailed documentation around the SDK installation and custom application building process, to assist user development efforts.

Many commonly used applications are already pre-built and available at <http://developer.cisco.com/opennxos>

Conclusion

Cisco Open NX-OS allows network operators and developers to integrate third-party and custom applications directly onto Nexus switch platforms. These applications can address a wide variety of configuration management, telemetry and advanced network monitoring challenges.

Network Programmability Fundamentals

Introduction

The drivers for network programmability and software-defined networking are numerous and growing. Network infrastructure components represent one of the last elements of the IT service delivery chain that require manual provisioning.

Automating and orchestrating network infrastructures through programmatic interfaces can provide many benefits, including reduced provisioning time and increased service velocity. A more dynamic, agile, repeatable, and reliable approach to network device configuration, operation and monitoring is necessary to keep pace with the rapid change of business today.

Automation and orchestration methodologies that have been successfully applied to compute, virtualization, and storage platforms in the data center can also be applied to network fabrics. Perhaps the most impactful of these methodologies is the utilization of Application Programming Interfaces (APIs). When network platforms expose APIs as a means for configuration, control, and monitoring, it becomes possible to replace existing manual processes with automated workflows.

The extensibility of an API allows for network components to be controlled and managed in a centralized fashion. Some practical examples involving centralized management through APIs include:

- **Network controllers** leverage APIs to manage and operate the network inclusive of monitoring key performance indicators (KPI) such as latency, jitter and delay. The controllers' knowledge allows for routine change actions to be performed, such as dynamic traffic movement away from identified saturated links.
- **Policy controllers** leverage APIs to apply policies to meet the security and performance requirements of a given application, service, or user - all based on high-level policy definitions.

This section will provide an overview of non-programmatic management interfaces available on Cisco Nexus switching platforms, as well as introducing the new open NX-OS programmatic interfaces for Cisco Nexus switches.

Conventional Network Interfaces

Network devices have traditionally been managed using the Command Line Interface (CLI) and with protocols such as Simple Network Management Protocol (SNMP) and Network Configuration Protocol (NETCONF).

Command Line Interface (CLI)

CLI has been the primary interface to interact with network devices, used to manage, operate and troubleshoot the network device throughout its lifecycle. CLI is a very comprehensive interface to leverage, but has limitations when used as the interface for automation:

- The CLI was designed as a human-readable interface, returning unstructured text data to the operator.
- This unstructured text data requires post-processing (screen scraping) to transcode to machine-friendly formatting.
- CLI does not return error/exit codes which can be programmatically acted upon.
- CLI is a single-threaded serial interface, reducing the ability to manipulate multiple objects at the same time.

"Screen Scraping" is the process of using text-processing tools or interpreters to examine the results of CLI commands executed via script. This approach was developed as a means to provide some rudimentary levels of automation, but has several drawbacks:

- Requires domain-specific scripting skills and often requires the use of regular expressions.
- Increased operational costs due to script repository maintenance, particularly with CLI syntax changes.
- Lack of interoperability / applicability across heterogeneous network devices.

Simple Network Management Protocol (SNMP)

SNMP is an industry standard protocol for managing network devices, and includes an application layer protocol, a database schema, and a set of data objects. SNMP exposes management

data in the form of accessible variables on the system, describing the system configuration in its entirety. These variables can be queried (and sometimes set) by management applications.

SNMP is widely used for monitoring of network devices. While it can be leveraged for configuring network devices, it is not widely deployed for this purpose. This is due in part to the fact SNMP lacks a defined discovery process which makes it challenging to find the correct MIB (Management Information Base) modules and elements for each platform. SNMP also suffers from limitations inherent to use of the UDP protocol which is stateless and considered a less-than-reliable transport. It has also been burdened by a lack of effective MIB standardization efforts.

NETCONF

NETCONF is another configuration management protocol, and continues to be developed as an IETF standard. The NETCONF protocol defines a simple mechanism through which a network device can be managed, configuration data can be retrieved, and new configuration data can be uploaded and manipulated.

The NETCONF protocol uses remote procedure calls for communication. The data payload is encoded within XML for NETCONF RPC calls. The data is sent to the server over a secure, connection-oriented protocol - secure shell (SSH) is an example of this. The server response is also encoded in XML. The key part of this mechanism is the request, and both the request and the response are fully described in an agreed upon communication model, meaning both parties understand the syntax that is being exchanged.

NETCONF has a set of transport protocol requirements which include:

- Connection-oriented communication
- Authentication
- Connection data integrity

Although Cisco and NX-OS platforms have extensive support for NETCONF, it will not be discussed in detail in this book.

CLI and SNMP have presented challenges for network automation. To address these limitations, Cisco has introduced NX-API with the launch of the Nexus 9000 Series. The following

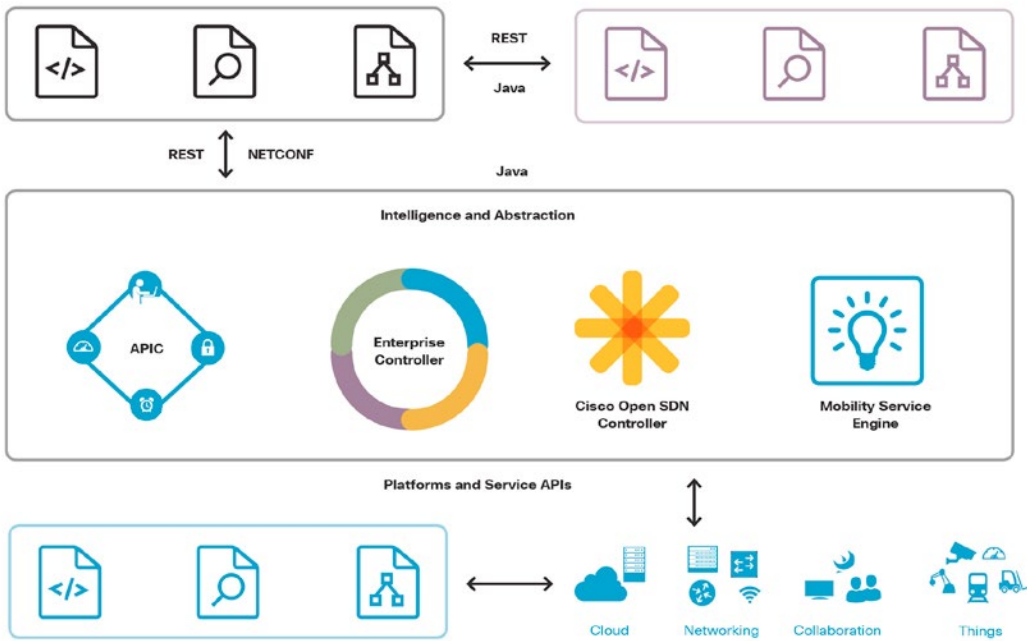
section introduces NX-API CLI and the NX-API Sandbox. This provides a new way to interact with the network - using APIs and programmability paradigms.

Programmable Network Elements

The previous chapter presented some of the key API-related concepts and technologies. In this chapter we will explore where APIs can be exposed within a network.

Just as there is a hierarchy associated with most network architectures, there is a hierarchy associated with the elements that can be controlled via APIs. The hierarchy illustrated below reflects the nature of the elements and APIs exposed at each layer, focusing specifically on two key factors: abstraction and scope of influence.

Figure: Various Programmable Elements in the Network



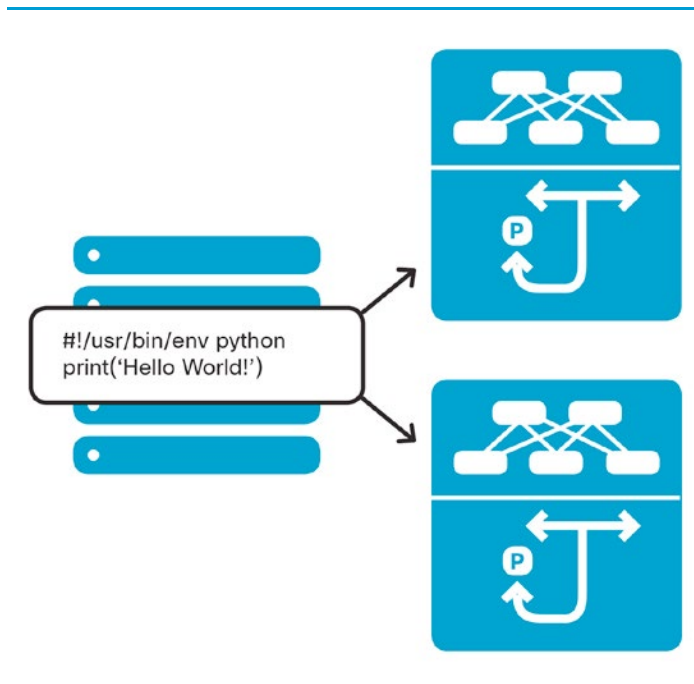
Device Layer

In the above example, devices reside at the lowest layer of the hierarchy, with applications and controllers using APIs to communicate with the device to achieve some desired functionality.

An important thing to consider at this layer is that visibility and scope of influence is generally limited to a single device. APIs at this layer are used to control device configurations and functions, as well as to capture information about things that are controlled by or visible to the device.

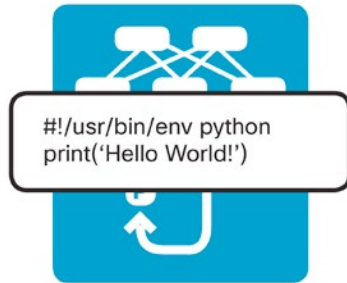
Because NX-OS is based on Linux and provides access to a Bash shell, it's possible for APIs to be leveraged not only by devices external to the switch, but also directly on the switch. In the following example, a Python script using an API library is executed "off-box" to perform a function on one or more switches:

Figure: Running the User Logic outside of the Switch on a Server or VM



In this example, a Python script using an API library is executed "on-box" to perform the same function. The target of this script is the local switch, but there is nothing preventing a script executed on one Nexus switch from using APIs to manipulate other switches in the network.

Figure: Running the User Logic within the Switch, Natively or in a Container



The ability to execute a script "on-box" can offer significant value, particularly when actions must be taken while a switch is not reachable by traditional access methods.

For example, it might be desirable that the access ports on a switch be held in a down state until uplink connectivity has been established. A Python script can configure these access ports in an administratively down state at boot. When the uplinks become active, a Python script triggered by the Embedded Event Manager (EEM) could then bring the access ports online.

Low-Level Device APIs

It is also possible for applications to control a subset of switch functionality by interacting with an autonomous subsystem within the device. For example, some of the ASICs that perform low-level packet switching may expose an API that can be leveraged by scripts to expose low-level counters or affect low-level packet forwarding functions:

Figure: Using APIs Exposed by ASICs on Network Devices

```
#!/usr/bin/env python
print('Register: '+ reg014.x4483 )
```



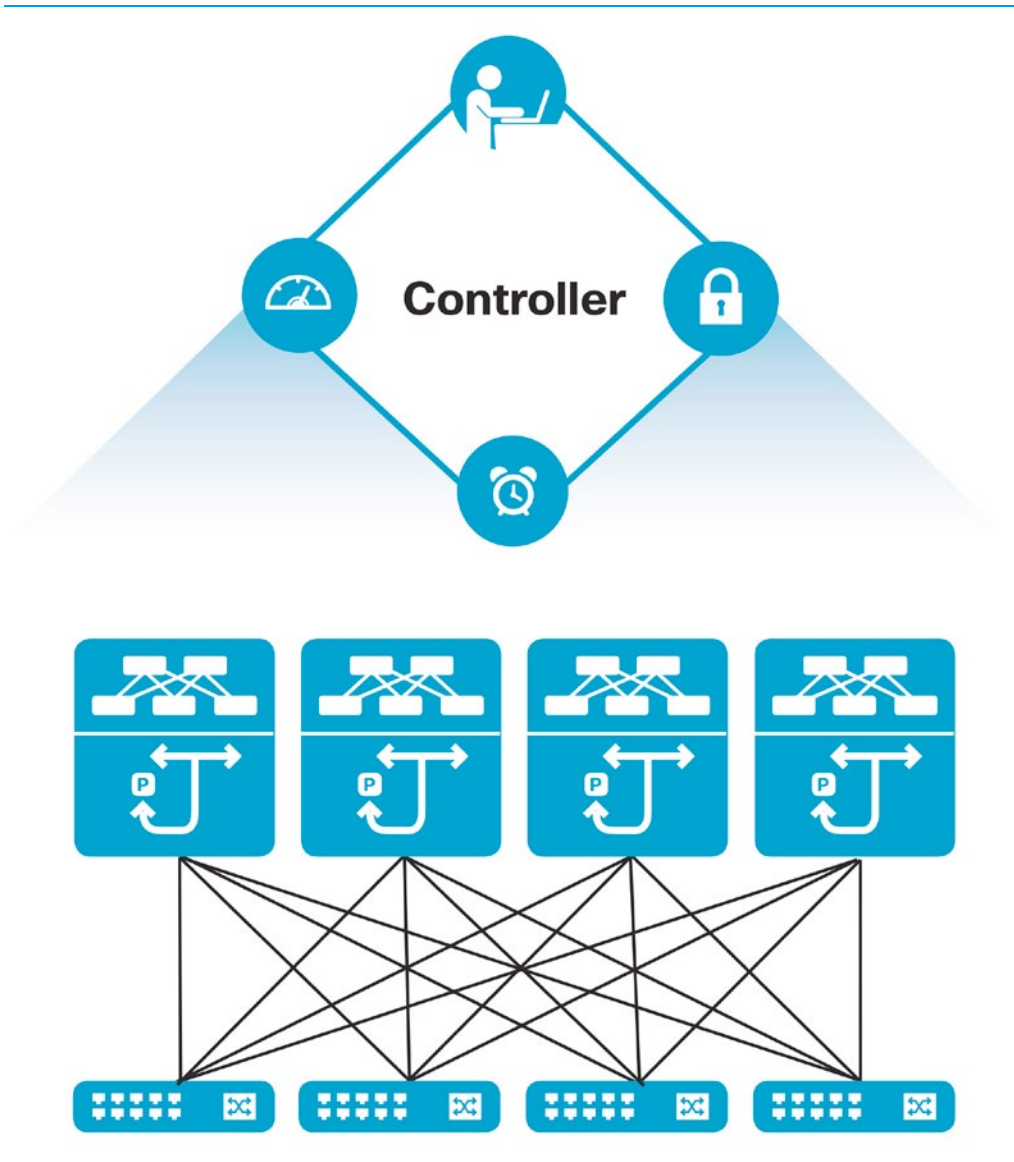
Controller Layer

The control layer resides above the device layer. Controllers are typically compute devices that:

- Provide network layer abstraction by hiding away details of individual devices and their connectivity. Applications communicate with this abstraction layer.
- Expose and use APIs in two directions. "Northbound" APIs are consumed by applications that need to use the controller's services; "southbound" APIs communicate with managed devices.
- Provide increased visibility through the capture, consolidation and correlation of data describing device-state and status, path and link characteristics, and many other types of information from southbound devices.
- Analyze and interpret data for consumption by northbound applications.
- Implement policy by accepting a policy definition, and manipulating the configuration of managed devices.

In general, controllers provide a increased level of visibility, intelligence, abstraction, and scope of influence than is possible at the device layer.

Figure: Controller Scope of Influence



Some examples of API-driven Cisco controllers include:

- **APIC** provides policy-driven management of data center network fabrics to streamline application deployment and security.
- **Enterprise Controller** extends policy-driven management capabilities to routers and switches throughout enterprise networks.
- **Cisco Open SDN Controller** provides a protocol conversion, network abstraction, and application delivery platform for Cisco and third-party developers.
- **Mobility Services Engine** enables the development of sophisticated, secure applications built around high-fidelity location tracking and analysis.

Higher Layer Orchestration

Some environments utilize orchestration tools or applications that reside above the controller layer or directly above the device layer. This layering creates a "stack" of API providers. Each layer has the capability of exposing "northbound" APIs that allow other applications to interact with the stack.

To summarize, APIs are offered at each layer of the infrastructure. At each successive layer throughout the architecture, the scope of influence and level of abstraction will increase.

NX-API CLI

The NX-API CLI is a web interface through which commands, traditionally entered via the CLI, can be encoded using either XML or JSON, and transmitted via HTTP or a secure transport (HTTPS) to the device. The commands are executed on-box and responses are returned in either XML or JSON format.

The NX-API CLI can be characterized as an "encapsulated CLI" supporting both NX-OS commands and those available in bash. In contrast, NX-API REST interface provides a means through which device configurations can be manipulated via RESTful API calls. Both the NX-API CLI and NX-API REST are serviced by an nginx web server on the back end.

NX-API CLI Security

NX-API CLI leverages HTTPS to secure and encrypt data. NX-API CLI provides a session-based cookie `nxapi_auth` when users first authenticate. The session cookie is used to avoid re-authentication during communication.

If the session cookie is not included with subsequent requests, another session cookie is required and is achieved through a full authentication process. Avoiding unnecessary use of the authentication process helps to reduce the workload on the device.

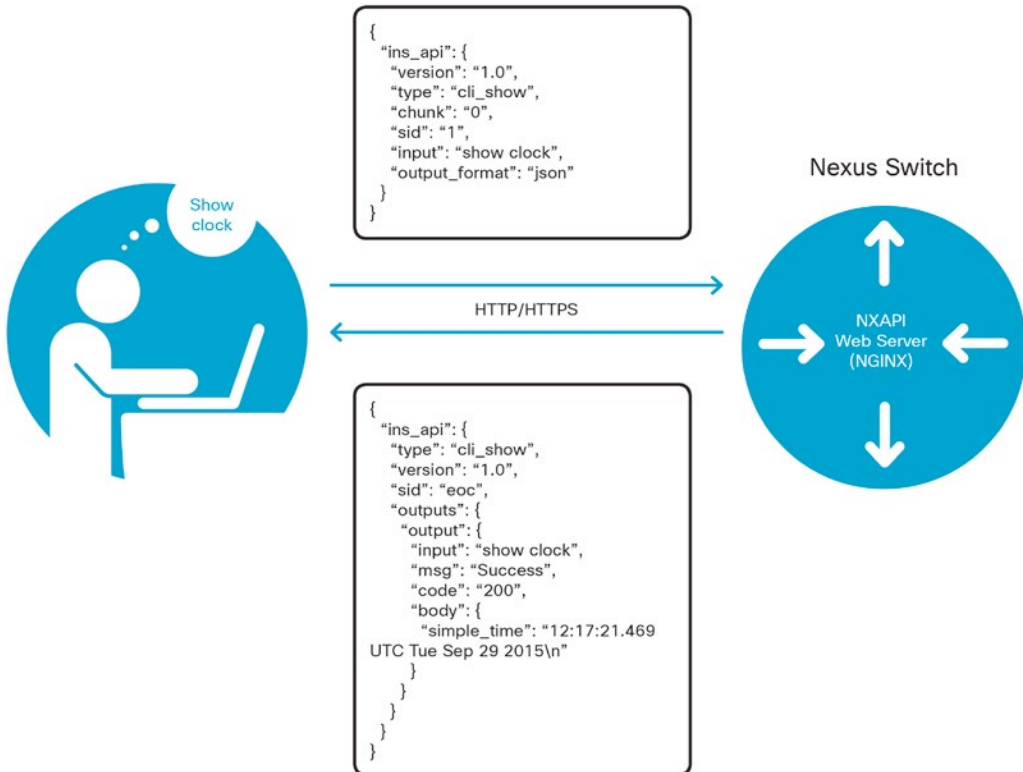
Note: An `nxapi_auth` cookie expires after 600 seconds (10 minutes). This value is not configurable.

Working with NX-API CLI

To enable NX-API CLI, enter the following command:

```
feature nxapi
```

The following diagram shows an NX-API call, demonstrating a `show clock` request.

Figure: Request and Response Sequence of NX-API CLI

NX-API CLI Developer Sandbox

Cisco has developed a tool to help customers become familiar with the NX-API CLI by creating the NX-API CLI Developer Sandbox. This is a web interface that allows users to make NX-API interface calls, and it can transcode traditional Nexus CLI commands into JSON or XML format.

The NX-API CLI Developer Sandbox is divided into three components:

- Post Pane:
 - This Pane is composed into three major categories:

- Text Area: This is where the user can enter show, configuration, or bash commands.
- Message format: XML, or JSON
- Command Type:
 - **cli_show** represents the CLI show commands (such as `show version` or `show clock`)
 - **cli_show_ansi** represents the CLI show command that expects ASCII output. This aligns with existing scripts that parse ASCII output. Users are able to use existing scripts with minimal changes.
 - **cli_conf** represents the CLI for the configuration commands
 - **bash** represents the Bash command
- Post
- Request Pane:
 - Provides a brief description of the request elements that are displayed
 - Includes a Python code output option
- Response Pane:
 - The POST response will be displayed in this pane

In order to access the NX-API CLI Developer Sandbox, point your browser to the IP or host-name of a Nexus switch.

```
http://<<switch ip>>
```

The authentication page will prompt for a username and password. Once authenticated, the sandbox interface will be displayed:

Figure: NX-API CLI Developer Sandbox Web Interface

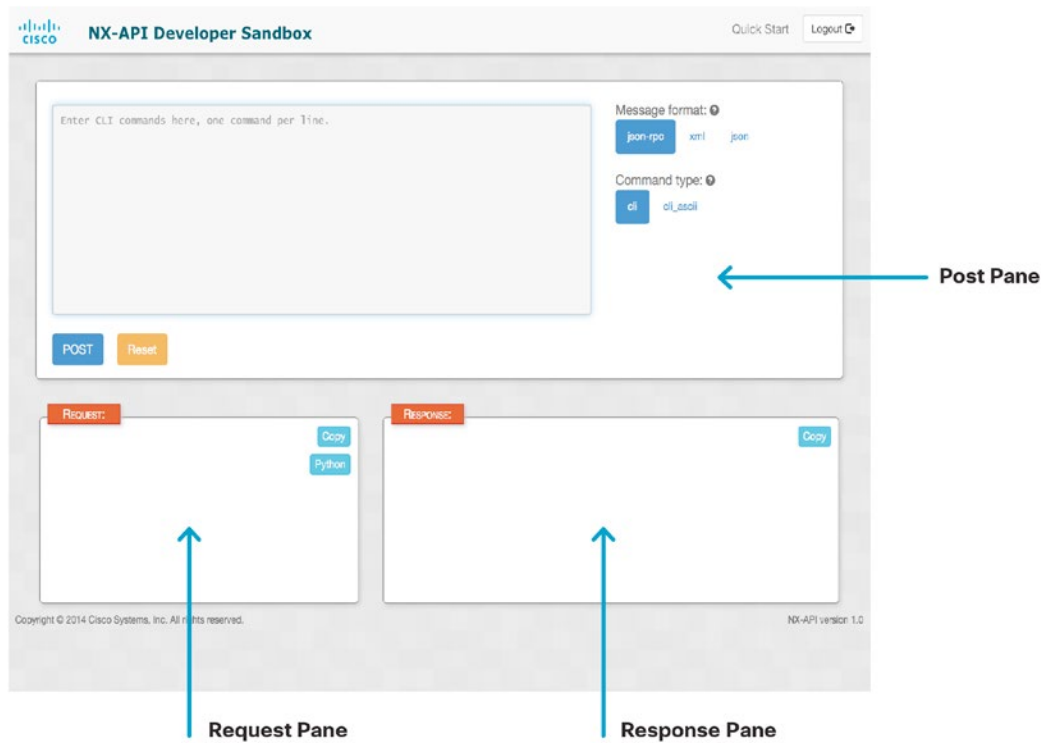


Figure: Screenshot of the 'show clock' Example Workflow

The screenshot displays the NX-API Developer Sandbox interface. At the top, the Cisco logo and 'NX-API Developer Sandbox' are visible, along with 'Quick Start' and 'Logout' links. The main area is divided into several sections:

- Command Input:** A text box contains the command 'show clock'. A blue arrow points from this box to a callout labeled 'Show clock'.
- Message format:** Radio buttons are present for 'json-rpc', 'xml', and 'json'. The 'json' option is selected.
- Command type:** Radio buttons are present for 'cli_show', 'cli_show_ascii', 'cli_conf', and 'bash'. The 'cli_show' option is selected.
- Buttons:** 'POST' and 'Reset' buttons are located below the command input area.
- Request:** A JSON object is displayed in a box with 'Copy' and 'Python' buttons. The JSON is:


```
{
  "ins_api": {
    "version": "1.0",
    "type": "cli_show",
    "chunk": "0",
    "sid": "1",
    "input": "show clock",
    "output_format": "json"
  }
}
```
- Response:** A JSON object is displayed in a box with a 'Copy' button. The JSON is:


```
{
  "ins_api": {
    "type": "cli_show",
    "version": "1.0",
    "sid": "e0c",
    "outputs": {
      "output": {
        "input": "show clock",
        "msg": "Success",
        "code": "200",
        "body": {
          "simple_time": "13:17:32.815 UTC Tue Sep 29 2015\n"
        }
      }
    }
  }
}
```

 A blue arrow points from this response box to a callout labeled 'Output clock'.

The user can also leverage the Python function in NX-API CLI Developer Sandbox in order to generate Python code.

Figure: Automatic Python Code Generation with NX-API CLI

The screenshot displays the NX-API Developer Sandbox interface. At the top, there's a header with the Cisco logo and 'NX-API Developer Sandbox'. Below this, a text area contains the command 'show clock'. To the right, there are dropdown menus for 'Message format' (set to 'json') and 'Command type' (set to 'cli_show'). Below these are 'POST' and 'Reset' buttons. The main area is divided into 'Request' and 'Response' sections. The 'Request' section shows a Python script using the 'requests' library to send a POST request to an NX-API endpoint. The 'Response' section shows the JSON output of the request. Annotations include a 'Python' callout pointing to the request code, a 'Python Function' callout pointing to the 'ins_api' object in the response, and a 'Python Output' callout pointing to the response JSON.

```

Request:
import requests
import json

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

myheaders={'content-type':'application/json'}
payload={
  "ins_api": {
    "version": "1.0",
    "type": "cli_show",
    "chunk": "q",
    "sid": "1",
    "input": "show clock",
    "output_format": "json"
  }
}

response = requests.post(url,data=json.dumps(payload)
, headers=myheaders,auth=(switchuser,switchpassword))
.json()

Response:
{
  "ins_api": {
    "type": "cli_show",
    "version": "1.0",
    "sid": "eac",
    "outputs": {
      "output": {
        "input": "show clock",
        "msg": "Success",
        "code": "200",
        "body": {
          "simple_time": "13:17:32.815 UTC Tue Sep 29 2015\n"
        }
      }
    }
  }
}

```

Python code generated by the NX-API CLI Developer Sandbox can be copied into an editor for execution using the Python interpreter.

Summary

NX-API CLI is a starting point for network engineers to familiarize themselves with network programmability via the NX-API.

NX-API REST provides the next level of programmability with a RESTful interface to a data model, enabling model-driven programmability. NX-API REST is discussed in more detail in the next chapter.

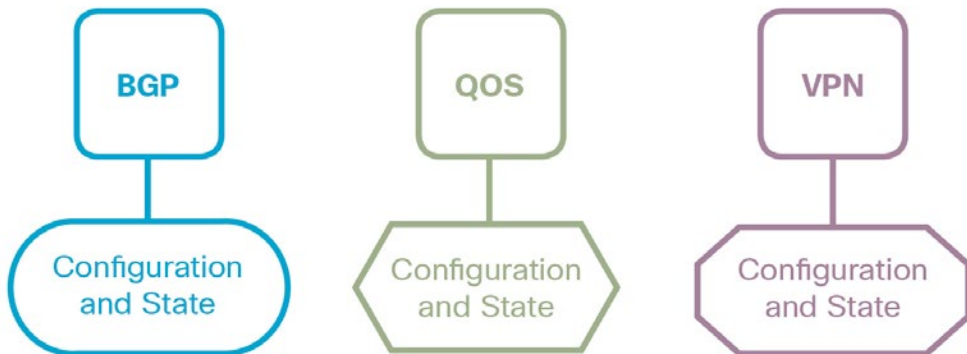
Model Driven Programming

Introduction

When routing platforms were first introduced in the 1980s, the pace of new feature development was frenetic. It was common practice for new development teams to be formed almost daily and for them to work in relative isolation. These teams developed every element of a new feature, including the command line interface (CLI) and the data structures that were necessary for its configuration, operation, and monitoring.

This practice persisted for years and was prevalent industry-wide, therefore the syntax associated with features such as BGP, QOS, or VPNs varied widely across platforms and software releases.

Figure: Traditional Information Store Model - Internal to each Process



This software development model was productive for customers as it supported rapid feature velocity. Over time, however, this model inhibited the ability to configure and manage networks at scale. Configuring and operating a single feature within a large network could require the use of several different CLIs.

Legacy scripting tools such as Tcl and Expect were leveraged to optimize configuration workflows and deliver more repeatable, less error-prone results, but they still had to be developed and maintained for potentially many disparate CLI interfaces.

As the networking industry adopts programmatic network device access methods, APIs and model-driven software interfaces offer a structured way to access and automate the network device.

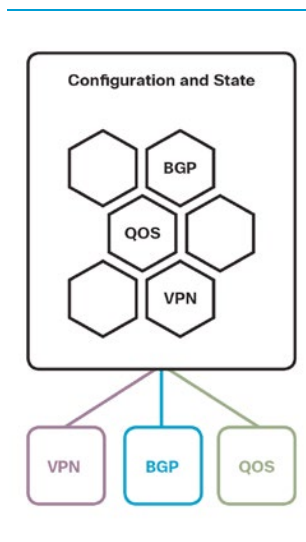
This chapter will explore solutions to these legacy network management issues, which can be addressed with data model concepts and the NX-API REST interface.

Model-driven Programming

Data models and application programmable interfaces (APIs) have shown enormous promise addressing the previously described problems and have been available in a number of forms since the early 2000s.

A data model is a schema or specification that describes the configuration and operational state associated with the elements and features of a routing or switching platform. Network devices that implement a data model, like the Cisco Open NX-OS and IOS-XR platforms, typically store the configuration and operational data in a centralized data store, and expose model via an API framework.

Figure: Modern, Centralized Information Store Model in Cisco Open NX-OS



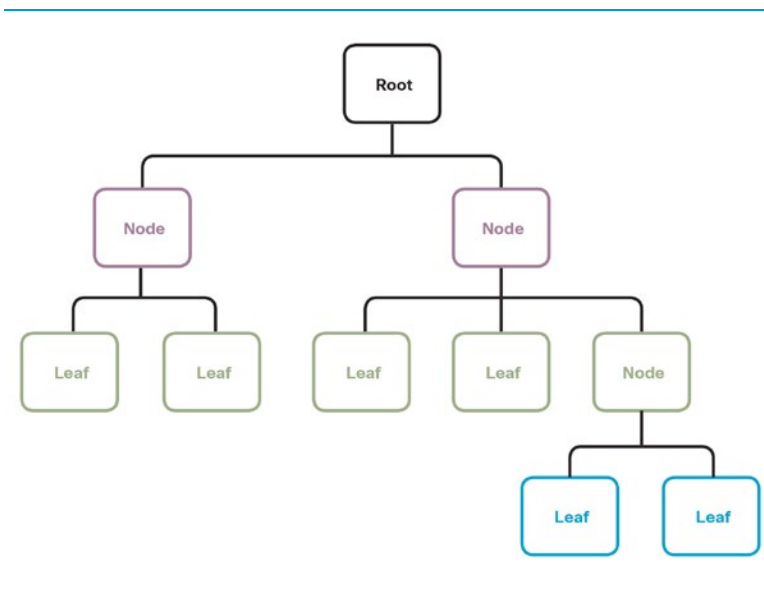
In a model-based architecture, CLI manipulates the model rather than individual features or elements. After an update to the data model, applicable network features or elements will react to any changes in the model. Every feature individually maintains its operational state and

performance data within the data model. The presence of a data model implementation in a network platform enables programmatic interfaces, such as NX-API REST.

The Nature of Data Models

Data models can be implemented using numerous data representation and storage formats, including arrays, linked lists, stacks, and graphs (e.g hierarchical trees). The hierarchical tree is very efficient in representing repetitive and hierarchical data and is typically associated with routing or switching platform configurations. Therefore it is the most common data model format used for networking platforms.

Figure: Information Store Model - Hierarchical Tree of Objects

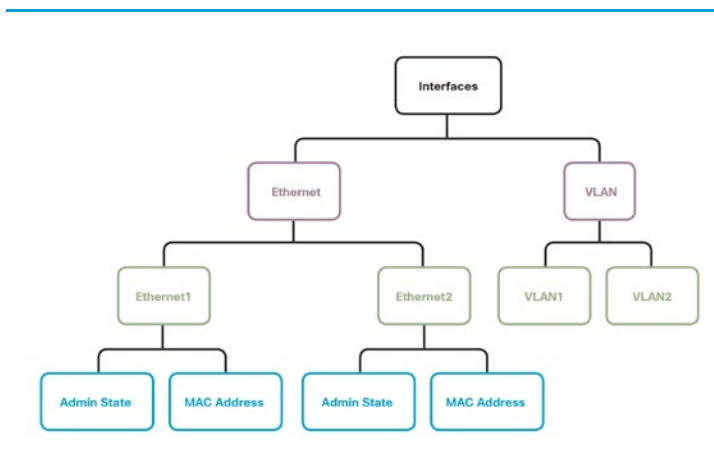


Tree-based data models are characterized by nodes that can have parents, children, or both. The root of the tree has child nodes but no parent. Leafs have parents but no children, and nodes have both; as illustrated above. In many data model implementations there are potentially one or more trees, each containing information about some major group or important division of information pertaining to a modeled object.

Data models **enable data to be easily structured, grouped, and replicated** to represent information related to network devices, features, and solutions. The example below represents a data model of network interfaces, where there is:

- a root node described by the category "interfaces"
- child-nodes for various interface types and discrete interfaces
- leaves containing information that pertains to specific object instances (interfaces), including configuration and operational state

Figure: Inheritance Relationship between Parent-Child Objects in Object Store



Data models can be used to **enforce data consistency and validity**. Rules govern data model structures, including the manner in which information can be inserted, modified, accessed, or deleted, and who is able to manipulate the data model. These rules ensure that data is maintained in a known, valid state.

Since data models represent the complete state of the network platform at any point in time, it is possible to implement and utilize recovery features such as backups and snapshots, allowing configuration changes to be validated and rolled-back if necessary.

Data models are **extensible**; they can grow and adapt to accommodate additions or changes to features and elements, provided any rules governing the structure of the data model are observed.

Data models are **flexible**; once a model structure is chosen, it can be used to encode more than one model simultaneously to meet the needs of multiple administrative audiences.

The Value of Data Models

A data model's structure, consistency, data validation, flexibility, and extensibility enable a shift from human- or script-based methods of device configuration to machine-based, **programmatic** methods.

Once the structure and rules governing the use of a data model are defined, it is possible to expose a programmatic interface that enables access to, and manipulation of, the information contained therein.

These programmatic interfaces leverage technologies already discussed, such as data interchange formats (XML, JSON), transports (HTTP, RPC), and protocols (NETCONF, RESTconf) to enable remote applications and controllers to manage and monitor the network platforms.

Programmatic interfaces form the foundation of software-defined networks (SDN) and allow network platforms to be controlled in a more rapid, dynamic, and repeatable manner.

Data Models and YANG

Data models have been in use on the Cisco IOS-XR platform for over a decade. In the following section, a data model specific to the Open NX-OS operating system, the Data Management Engine-Data Model (DME-DM) will be introduced.

There are industry-wide efforts, including those in the IETF NETMOD working group and the OpenConfig organization, to build common data models in YANG. Cisco and Tail-f pioneered the data-modeling language presently available in the industry. YANG models can be mapped to network platform data models, and present an opportunity to deliver common interfaces for a wide variety of end-to-end network use cases.

More specific data models, such as the the Cisco Open NX-OS managed information tree, will be discussed in the next section. Data models and APIs are essential to successful automation.

Cisco Open NX-OS MDP Architecture

The Cisco Open NX-OS Model-Driven Programmability (MDP) architecture is an object-oriented software framework aimed at development of management systems. The MDP object model is an abstract representation of the capabilities, configuration and operational state of elements and features on a Cisco Nexus switch. The object model consists of various classes that represent different functions and their attributes on the switch.

As an example, a switch has physical network interfaces, and those interfaces have characteristics. These characteristics include the mode of operation (Layer 2 or Layer 3), speed, and connector type(s). Some of these characteristics are configurable while others are read-only. The object model is responsible for representing each element's state and hierarchy.

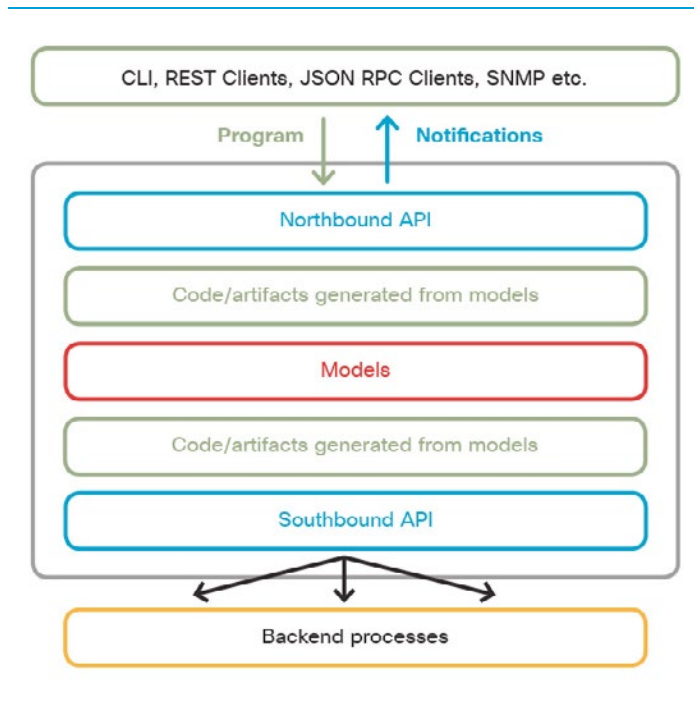
The Cisco Open NX-OS MDP framework provides users with many advantages:

- Ability to automate network configuration using a programmatic method
- Backed by a data model with abstraction suitable for network programmability
- Support for a variety of management agents
- Ability to extract configuration and operational data from devices
- Security through Role-Based Access Control (RBAC)

This section will provide an overview of the data management engine, managed objects, and object relationships and dependencies.

Data Management Engines

The data management framework consists of the Data Management Engine (DME), clients of the DME (“northbound” interface), and back-end processes and applications.

Figure: Automated Generation of Code and Artifacts from Data Model Definition

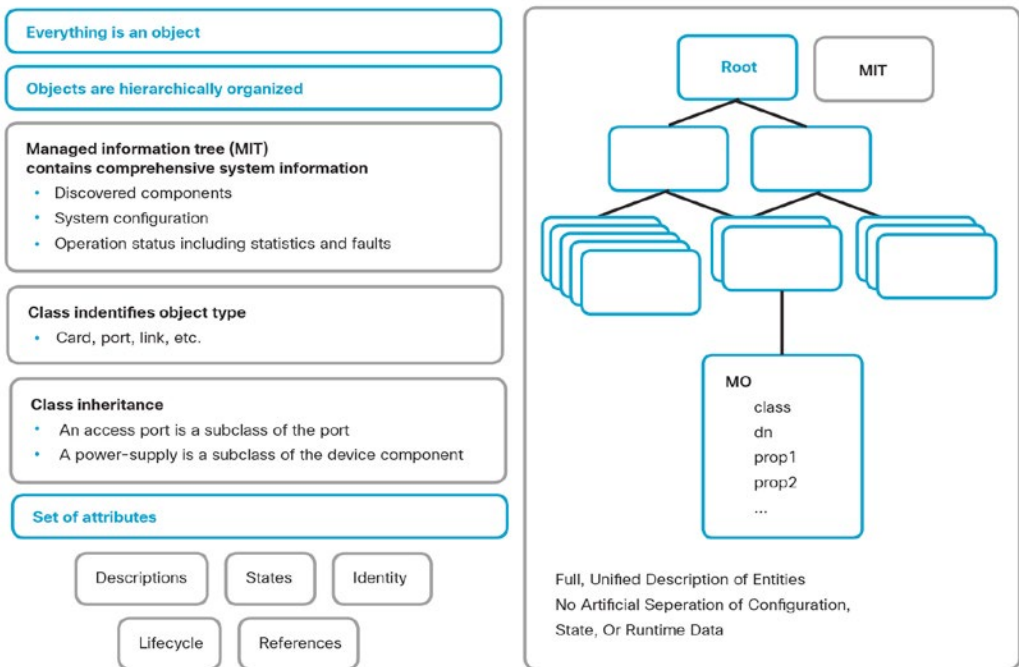
The DME has the following properties:

- Transactional, with each transaction following ACID semantics with regard to objects they affect:
 - Atomicity
 - Consistency
 - Isolation
 - Durability
- Model-driven architecture
- Multi-threaded
- Secure
- Upgradable
- Ultimately consistent across multiple objects affected by a transaction

Management Information Tree

The DME holds the repository for the state of the managed system in the Management Information Tree (MIT). The MIT manages and maintains the whole hierarchical tree of objects on the switch, with each object representing the configuration, operational status, accompanying statistics and associated faults for a switch function. The MIT is the single source of truth for the configuration and operational status of NX-OS features and elements. Object instances, also referred to as Managed Object (MOs), are stored in the MIT in a hierarchical tree, as shown below:

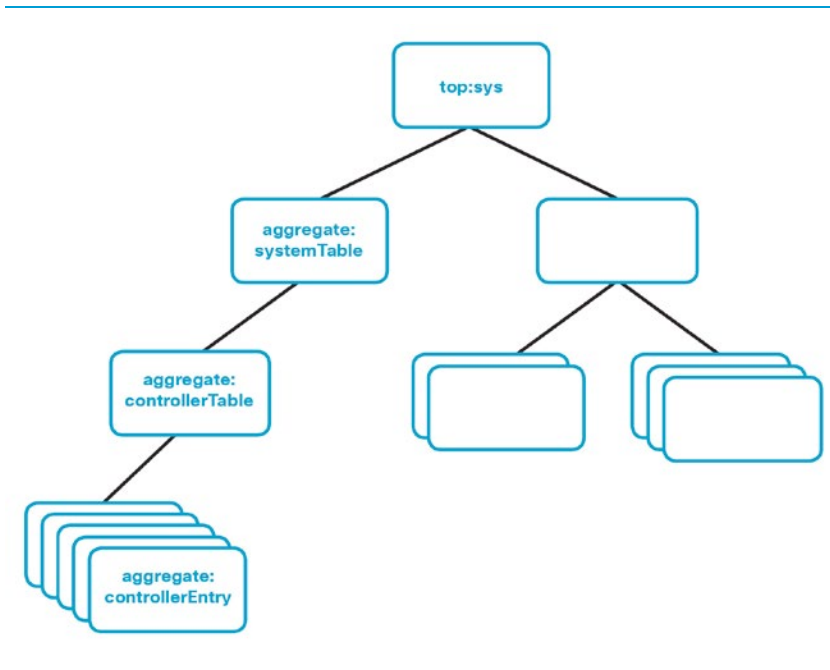
Figure: Characteristics of Open NX-OS Object Store - Management Information Tree (MIT)



The general concept is similar to the tree-based hierarchy of a file system. The MO database is organized in such a way that there is a parent-child hierarchy, meaning there is a root node followed by a hierarchical group of children.

One such parent-child hierarchy is illustrated using the following example:

Figure: Parent-Child Containment Relationship in Open NX-OS Data Model



In the above example:

- top:sys is the root of the entire MIT
- aggregate:systemTable is a child of top:sys
- aggregate:controllerTable is a child of aggregate:systemTable
- parent/child relationship continues through the table

When there are user-provided distinguishers, values for the naming property of the object and multiple objects of the same class can exist in the same subtree. In the example above, multiple instances of the controller class can exist since it has a naming property ([id]) associated with it in which each instance will have a unique user supplied [id].

Everything is an Object

Cisco Open NX-OS is object-oriented, and everything within the model is represented as an object. Objects store the configuration or operational state for Open NX-OS features associated with the data model. Within the model, objects can be created in reference to other objects. References may be among various networking constructs, such as interfaces and VLANs, as well as relationships between these components. Trunked VLAN interfaces represent an example of related, hierarchical objects.

The following section outlines how MOs are defined in Open NX-OS during the software development process.

MO Definition

Open NX-OS software developers define models for various classes in an XML format, and use a modeling schema with various tags to denote specific attributes of the class. This relationship is defined prior to compile-time and is enforced during run time. However, it is only the relationship that we create when defining the model and properties of each MO.

Class model definition

```
<model>
  <package name="aggregate">
    <objects>
      <!-- Section1 -->
      <mo name="SystemTable"
        concrete="yes"
        label="System Table"
        read-access="access-protocol-util"
      >
<!-- List of other properties -->
      </mo>
      <rn mo="SystemTable">
        <item prefix="systemTable"/>
      </rn>
      <contains parent="top:System"
        child="SystemTable"
      />
    </objects>
  </package>
</model>
```

```

<mo name="ControllerTable"
  concrete="yes"
  label="Controller Table"
  read-access="access-protocol-util"
  >
</mo>
<rn mo="ControllerTable">
  <item prefix="controllerTable"/>
</rn>
<contains parent="SystemTable"
  child="ControllerTable"
  />

<mo name="ControllerEntry"
  concrete="yes"
  label="Controller Entry"
  read-access="access-protocol-util"
  >
  <! Section2 >
  <property name="id"
    type="scalar:Uint32"
    owner="management"
    mod="implicit"
    label="Controller ID"
  />

  <rn mo="ControllerEntry">
    <item prefix="controller" property="id" />
  </rn>
  <! Section3 >
  <contains parent="ControllerTable"
    child="ControllerEntry"
    />
  <! Section4 >
  <chunk target="ControllerEntry"
    owner="vlanmgr"
    type="primary"
  />

```

MO Properties

On a Cisco Nexus switch, the configuration and/or operational state of the network can be stored in the MO as properties of the MO. Some properties can be used to store configuration state, such as enabled/disabled state and attributes of protocols, while other properties can be used to store operational state, such as statistics, faults, events, and audit trails.

Properties within MOs will have an attribute.

```
propertyname="attribute"
```

Values can be expressed in terms of regular expressions, etc. Thus, one can actually specify that a certain property can have a string as its value only if that string matches a particular regular expression. All type/regex checking is done prior to creating an MO and storing the value as one of its properties.

MO property definition

```
<property name="speed"
  type="Speed"
  owner="management"
  mod="implicit"
  label="Speed"
/>
```

MO Grouping

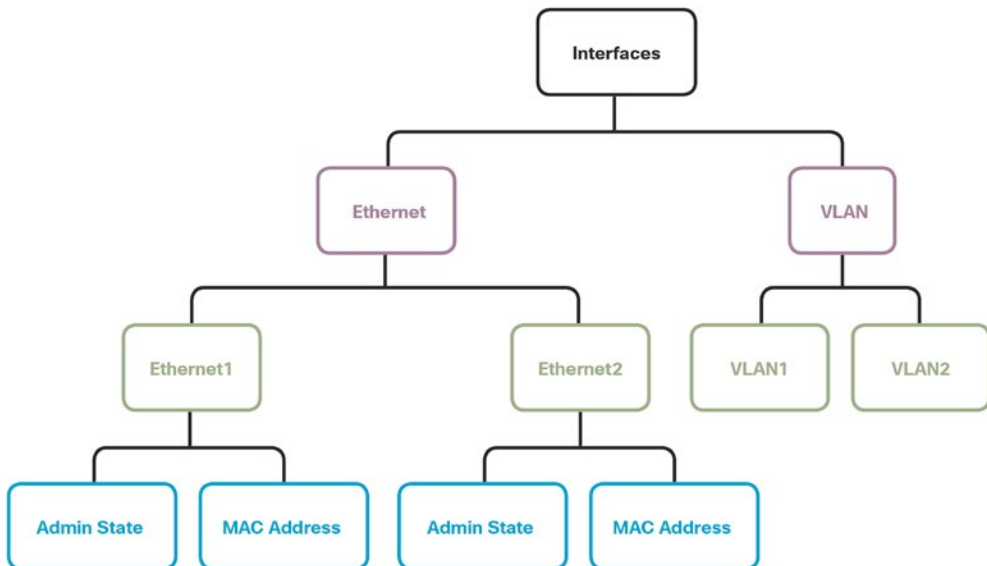
Each class is part of a package, and the definition of the class includes the **package name**. Each class has a **mo name** that identifies the class. References to an MO in the current package can be made by directly specifying the name of the MO. Any reference to an MO in a different file/package will have to prefix the reference with the package name of the MO being referred. Any reference from an external package to the **controllerTable** MO, in the Class model definition example above, should be formatted as **aggregate: controllerTable**, where **controllerTable** is the class name and **aggregate** is the namespace qualifier. This ensures you are using the right class while referencing it in some other package.

MO Inheritance

Each object within the data model is an instance of its associated class. The objects in the model can use the concept of inheritance, which allows for new objects to take on properties from existing, more abstracted base objects. For example, a physical interface can be a data port or a management port; however, both of these still have the same basic properties, so they can inherit from a single interface base class. Rather than redefine the same properties many times, inheritance can be used to define them in one base class, and then customize them for a specific child class.

A class can be defined as a purely abstract class by setting the `concrete=yes/no` attribute to `concrete=no`. Such classes are purely abstract - no objects of the class can be instantiated. Other non-abstract, concrete MO can reference the abstract MO as a “super class” and inherit the class properties.

Figure: Parent-Child Inheritance Relationship in Open NX-OS Data Model



Access Control

An important attribute of the MO is its access control attribute, defined by `access=admin/user`. This basically denotes the access privileges for the specific MO and determines which user-access-role is allowed to access and modify this particular MO. This ability to specify per-MO permissions enables a programmability model with granular access control to meet the security requirements of the infrastructure.

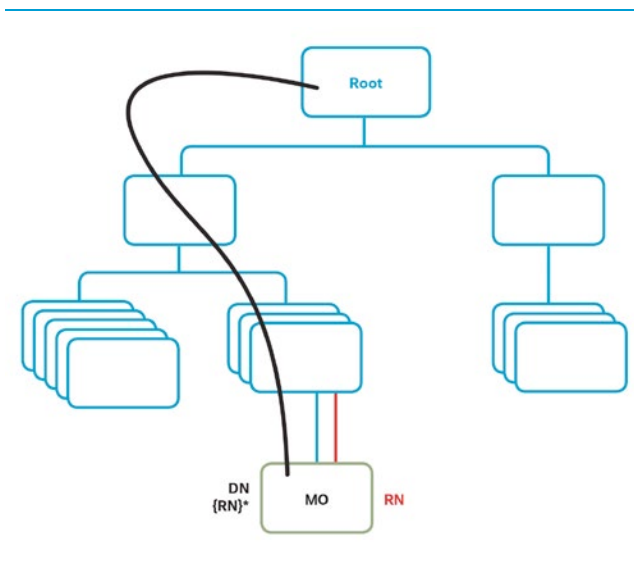
Identifying Objects in the MIT

Managed objects make up the management information tree. Each MO, other than “top:Sys” has the following attributes:

- A parent object
- A relative name (RN) that uniquely identifies the object among its siblings
- A distinguished name (DN) that uniquely identifies the object globally

The RN is immutable; it is set once at MO creation time. The DN is the concatenation of relative names along the path from the root to the MO, with RNs separated by “/”

Figure: Identifying Objects with their Distinguished Name (DN) and Relative Name (RN)



Distinguished Name

Every object in the object store will have a DN. The distinguished name enables you to unambiguously identify the target object. The distinguished name has the following format consisting of a series of relative names:

```
dn = {rn}/{rn}/{rn}/{rn}...
```

In the following example, the DN provides a fully qualified path for peer-[192.168.0.2] from the top of the object tree to the object. The DN specifies the exact managed object on which the API call is operating.

```
< dn = "sys/bgp/inst/dom-default/peer-[192.168.0.2]" />
```

DN_of_MO is the concatenation of **Parent_DN** and **RN_of_MO**

top:Sys is the only MO that has RN ~= DN. RN of **top:Sys** specified in the model is **sys** and its DN **/sys**

In the example above:

- The RN of **aggregate:SystemTable** is **systemTable**. Thus its DN is **/sys/systemTable** because it does not have a naming property. The RN of **aggregate:ControllerTable** is **controllerTable**. The DN of **aggregate:ControllerTable** MO is **/sys/systemTable/controllerTable**
- The RN of **aggregate:ControllerEntry** is **controllerEntry-[id]**, where id is the naming property of the MO.
- The DN of **aggregate: ControllerEntry** MO will become **/sys/systemTable/controllerTable/controllerEntry-[id]**. Since this MO has a naming property, we can have multiple instances of this MO under its parent MO **aggregate:ControllerTable**, with each instance being associated with a unique [id] value.
- The DN **/sys/systemTable/controllerTable/controllerEntry-1** refers to one particular instance of controllerEntry class.

Relative Name

The relative name identifies an object within the context of its parent object. The distinguished name is composed of a sequence of relative names. The following distinguished name is composed of the following relative names:

Distinguished name

```
<dn="sys/bgp/inst/dom-default/peer-[192.168.0.2]"/>
```

Relative name

```
'peer-[192.168.0.2]'
```

Relative name and Distinguished name

Object Name	DN	Parent DN	RN
topSystem	'sys'	N/A	'sys'
BGP	'sys/bgp'	'sys'	'bgp'
BGP Instance	'sys/bgp/inst'	'sys/bgp'	'inst'
BGP Domain	'sys/bgp/inst/dom-default'	'sys/bgp/inst'	'dom-default'
BGP Peer	'sys/bgp/inst/dom-default/peer-[192.168.0.2]'	'sys/bgp/inst/dom-default'	'peer-[192.168.0.2]'

The next chapter, which discusses the REST API, provides more information about how to use an object's distinguished name and relative name to form a REST URI.

Accessing Objects with Queries

The MIT allows operations such as search, traversal, insertion, and deletion. One of the most common operations is a search to query information from the MIT.

The following types of queries are supported:

- tree-level query: search the MIT for objects of a specific subtree.
- class-level query: search the MIT for objects of a specific class.
- object-level query: search the MIT for a specific DN.

Each of these query types support numerous filtering and subtree options, but the primary difference is the way that each type is used.

A class-based query is useful for searching for a specific type of information without knowing all the details, or only knowing partial details. Because a class-based query can return a range of results from zero to many, it can be a helpful means of querying the fabric for information when the full details are not known. A class-based query combined with filtering can be a powerful tool for extracting data from the MIT. For example, a class-based query can be used to find all interfaces that are functioning as uplink interfaces on leaf switches in a datacenter fabric and extract their CDP/LLDP information, for a way to rapidly create a cable plan of the fabric.

An object-based (DN) query returns a single match, and the full DN for an object must be provided for a match to be found. Combined with an initial class query, a DN query can be helpful for finding more details about an object referenced from another object, or for updating a local copy of information.

Both query types support tree-level queries with scope and filtering criteria. Thus, you can query the MIT for all objects of a specific class or DN and then retrieve the children or complete subtree for the returned objects. Furthermore, the data sets can be filtered to return only records of interest for the current purpose.

The next chapter, which discusses the REST API, provides more information about how to build and run these queries.

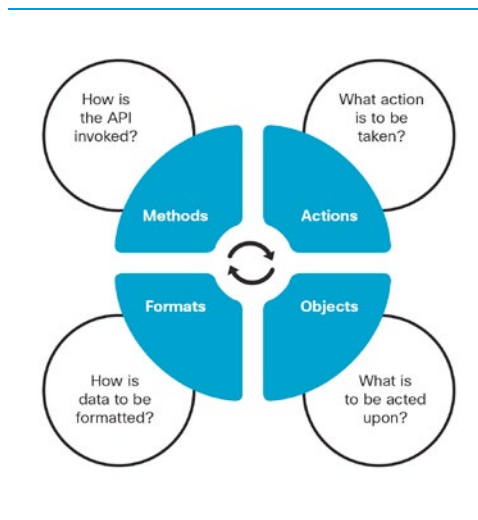
REST API Primer

Application Programming Interface - API

Most applications expose some sort of API that governs how an application can be accessed by other applications. APIs provide a set of routines, protocols, tools and documentation which can be leveraged for programmatic interaction. They represent a means through which elements or applications can be programmatically controlled and describe how external applications can gain access to capabilities and functions within another application. APIs have four primary components:

- **Methods:** Describes the mechanism of the API implementation including how resources communicate, provide encapsulation, etc
- **Actions:** This is the intent of the API call, often referred to as a "verb". It describes the operations available such as GET, PUT, POST, and DELETE
- **Objects:** This is the resource the user is trying to access. This is often referred to as a noun and it is typically a URI.
- **Formats:** This is how the data is represented. e.g., JSON, XML, etc.

Figure: API Framework



The following characteristics of an API enable users to build a more efficient, manageable and reliable network via automation.

- **Modularity:** Applications can be built leveraging clearly defined and reusable modules
- **Abstraction:** APIs abstract the details of the underlying implementation from the higher level logic that invokes it
- **Stability:** APIs provide a stable and consistent interface

Cisco Open NX-OS exposes three primary APIs:

- NX-API REST - HTTP-based RESTful API
- NX-API CLI - RPC-based API
- NETCONF API

Each of these APIs can be used with multiple language bindings. There are Python bindings for both NX-API REST and NX-API CLI.

This chapter will explore the Open NX-OS RESTful APIs in more detail.

HTTP

Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, and hypermedia information systems and is one of the primary methods for communication with the API. In this case, our focus will be leveraging HTTP from a programmatic perspective rather than as an access mechanism, which is the case with the World Wide Web. Hypertext is structured text using logical links between nodes containing text (also called "hyperlinks") and HTTP is the protocol to enable exchange or transfer of hypertext.

HTTP has two types of messages:

- Requests - from a client to a server, consisting of:
 - Request Line: The request line begins with the method token, followed by the Request-URI and the protocol version.
 - Request Uniform Resource Identifiers (URI): The URI is a set of string of characters used to identify the name of a resource. The most commonly used URI is in the form of the Uniform Resource Locator (URL). For example www.cisco.com

- Request Method: The request method indicates the method to be performed on the resource identified by the given Request-URI. Here is a list of supported methods:
 - GET - The action of "getting" the information which is identified by the Request-URI
 - HEAD - Similar to the GET method but it includes the transfer of the status line and the header section only
 - POST - Used to send data to the server. Examples include loading a new configuration or querying the state of the network element
 - PUT - Replaces all the current representations of the target resource with the uploaded content
 - DELETE - Requests the origin server delete the resource identified by the Request-URI
 - OPTIONS - Requests for information about the communication options available on the Request-URI
 - TRACE - Invokes a remote, application-layer loop-back of the request message
 - CONNECT - Establishes a tunnel to the server identified by a given Request-URI

- Responses - from a server to a client in response to a request, consisting of:
 - Status Line: This is the first line of the Response Message which contains the protocol version followed by the status code.
 - 503 – Services Unavailable
 - 500 – Internal Server Error
 - 404 – Not Found
 - 403 – Forbidden
 - 400 – Bad Request
 - 301 – Moved Permanently
 - 201 – Created
 - 200 – OK
 - Message Body

The following diagram describes a Client Request and the Server Response communication

Figure: HTTP Client Request and Server Response

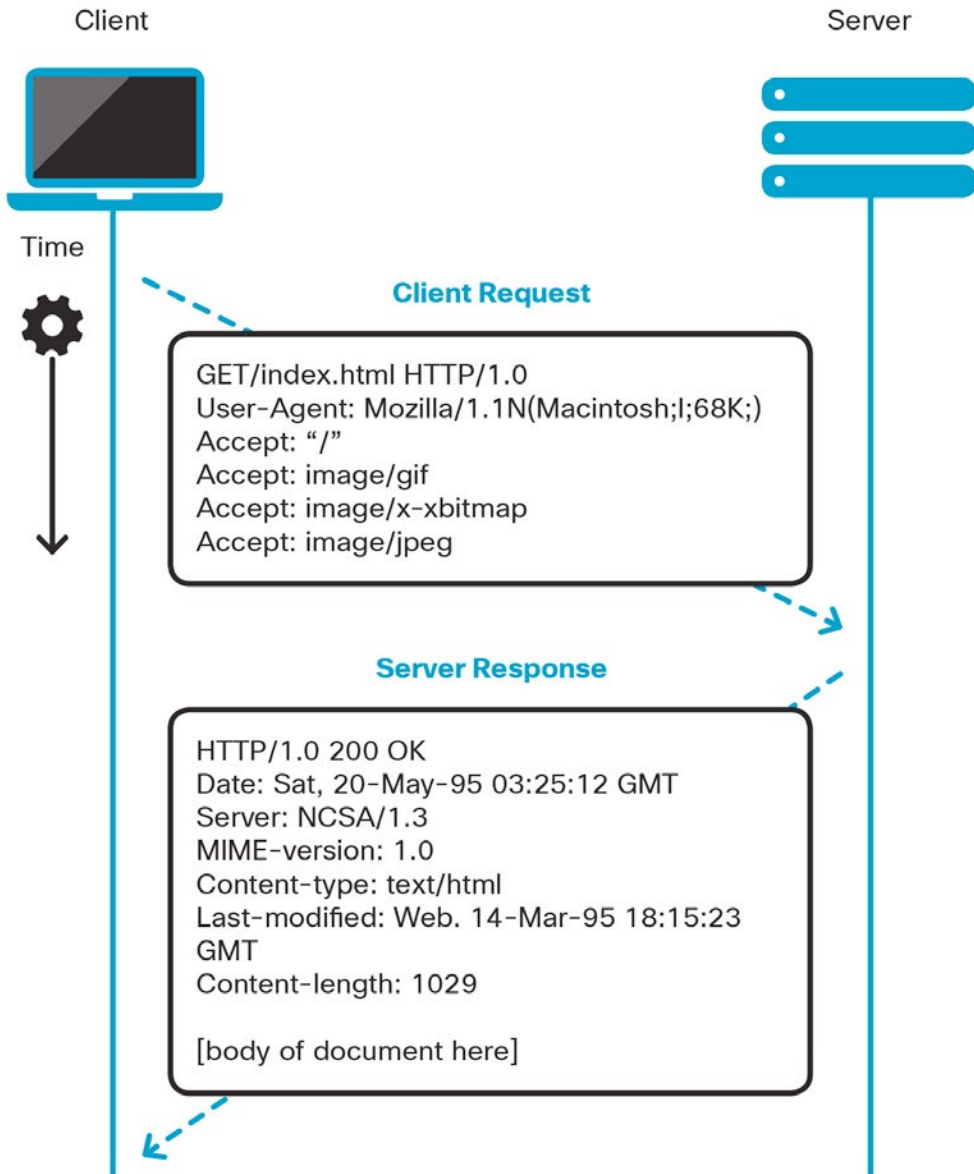
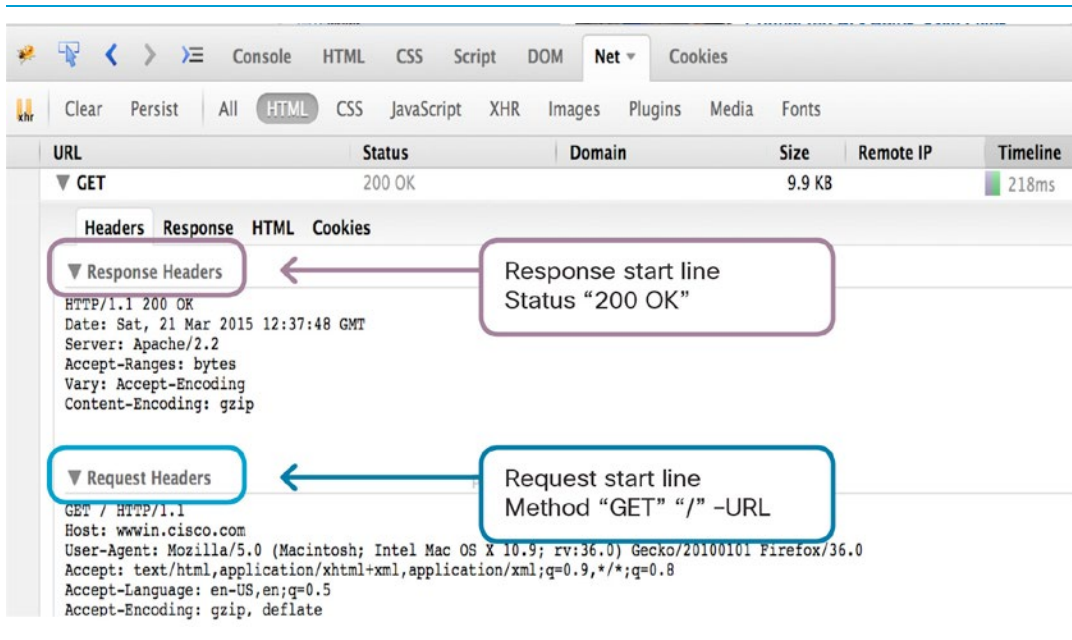


Figure: HTTP Client Request with Response Code

Representational State Transfer - REST

Given an understanding of APIs and their importance, let's explore REST - **RE**presentational State Transfer. REST is a software architecture style for designing scalable networked applications, specifically web services. By providing a coordinated set of constraints applied to component design in distributed systems, REST facilitates higher levels of performance and more maintainable architectures.

RESTful constraints are described as follows:

- 1 **Client server** - clients and servers are fully separated and communicate only via the RESTful interface.
- 2 **Stateless** - no client context or state is stored on the server between requests, and each client request must contain all of the information needed for the server to service the request.

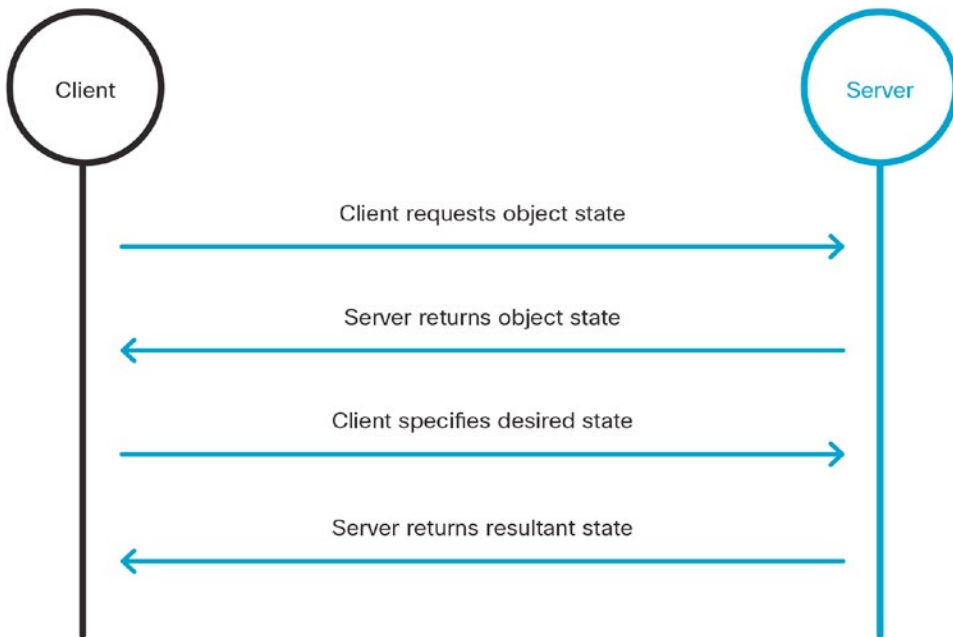
- 3 **Cacheable** - clients can cache responses, and servers must define the cacheability of the response.
- 4 **Layered** - a client should not be able to tell whether it is connected to a server or to an intermediate that provides functionality such as security, caching, or load-balancing.
- 5 **Code on Demand (Optional)** - servers can at times extend the capabilities of a client through the transfer of executable code or scripts.
- 6 **Uniform Interface** - both client and server must adhere to a uniform interface that allows for the independent development of functionality.
- 7 **Resource Identification** - individual resources are identified using URIs in requests. Representations of resources are distinct from the actual resources and may be provided in formats such as HTML, XML, or JSON.

REST relies on standards protocols HTTP or HTTPS to transmit calls between entities, and within that leverages unique URL identifiers, either a verb or a noun. The specified HTTP methods or verbs for REST are as follows:

- GET - List the URI's in a collection, or a representation of an individual member
- POST - Create a new entry in a collection. The new entry's URI is assigned automatically and returned by the operation
- PUT - Replace an entire collection with a collection, or individual member with another. If a member does not exist, create one
- DELETE - Delete an entire collection or an individual member

The two behaviors of REST operations are:

- Idempotent - the operation has the same effect no matter how many times it is performed (PUT and DELETE)
- Nullipotent - the operation does not affect the resource (GET)

Figure: REST Communication Flow

URI

The URI is a string of characters used to identify the name of a resource. Two types of URI's exist:

- Uniform Resource Locator (URL) - what we often refer to as a web address
- Uniform Resource Name (URN) - less frequently utilized, but intended to compliment URLs by offering a way to identify specific namespace resources

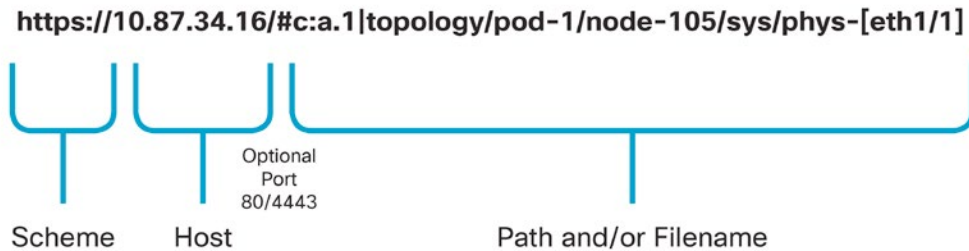
A REST URL contains:

- Protocol/schema
- Resource IP or hostname

- Path and filename

An important distinction and concept to understand is the difference between absolute and relative. In absolute we provide the **exact** path, whereas in relative there is a layer of indirection where we give the path to the actual location. The following is a sample URI:

Figure: URI Model



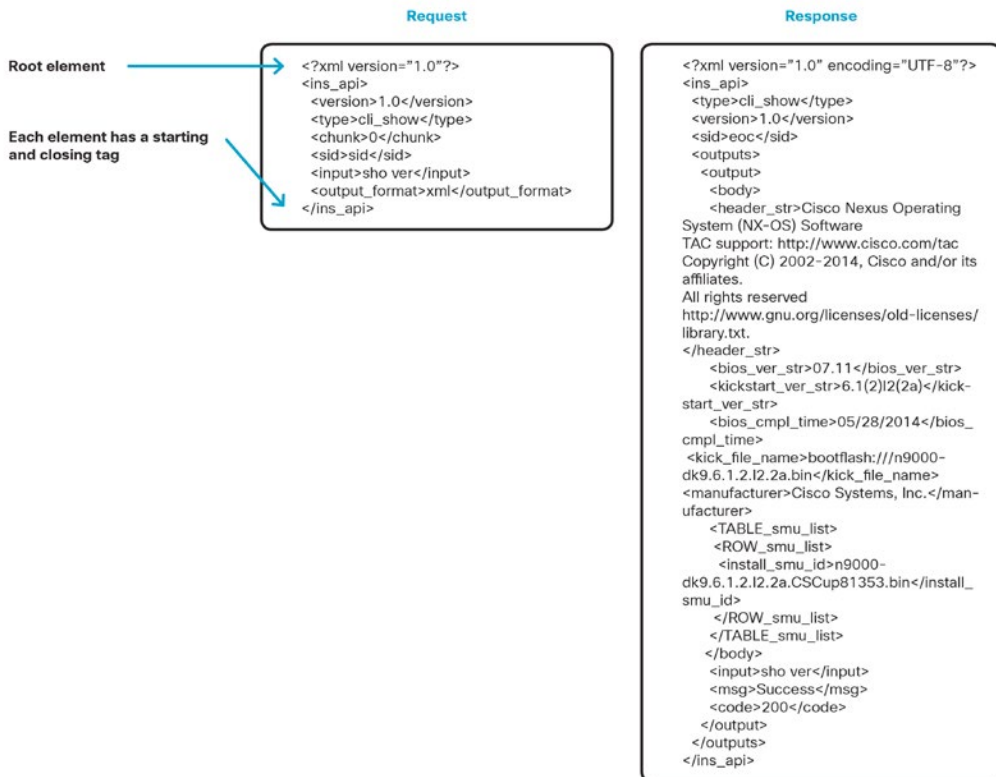
API Security

REST uses HTTPS for encrypted transport. Several widely-accepted industry practices to provide API security are utilized today, including OAuth, BasicAuth, and API Keys.

Data Formats

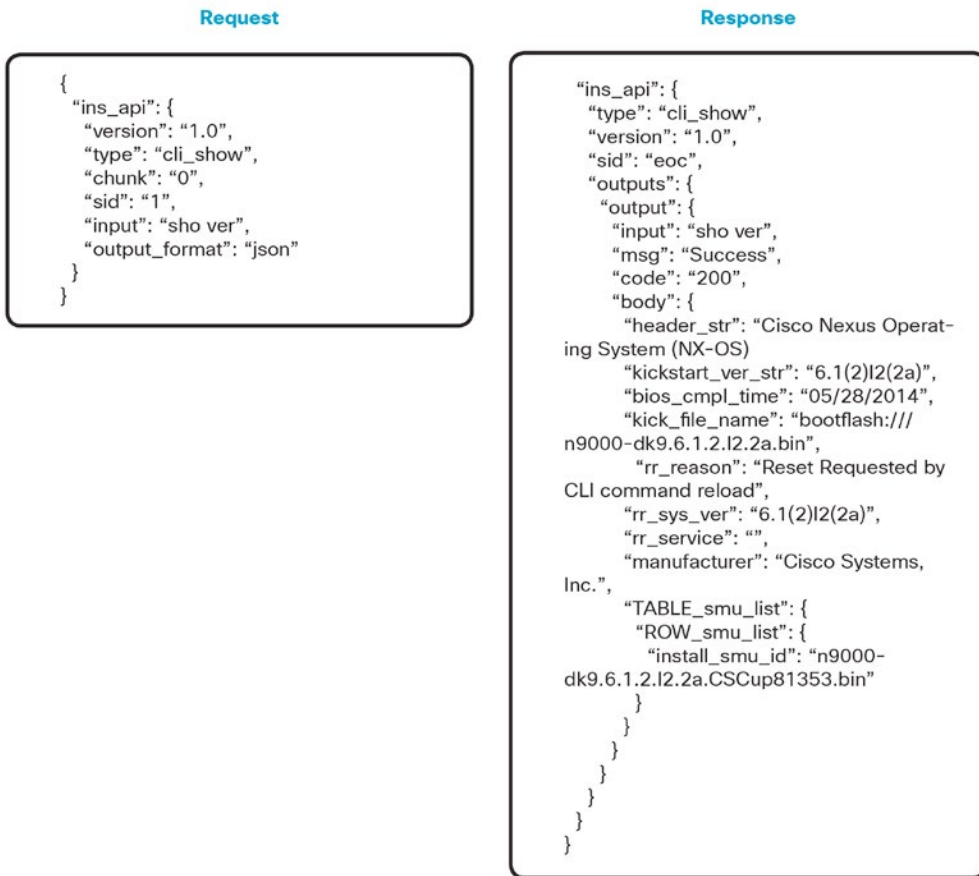
Data formats represent different ways we render output information to the user or application. Two primary data formats we'll cover here are **JavaScript Object Notification (JSON)** and **eXtensible Markup Language (XML)**.

XML is similar to HTML, but designed to encode structured data. Tags are self-defined rather than standardized.

Figure: XML Request and Response Format

JSON is focused on being more human readable and uses attribute-value pairs. The encoding format utilizes:

- a collection of name/value pairs
- an ordered list of values

Figure: JSON Request and Response Format

The REST API's structure is one of the most prevalent API design types available. It provides a language-independent easy-to-structure interface based on well known HTTP web concepts that are familiar to most users.

Cisco NX-API REST Interface

NX-API REST is a RESTful programmatic interface for Cisco Open NX-OS. In the previous sections, we discussed how NX-OS stores configuration and operational data in a centralized object store, the Management Information Tree (MIT). The nodes in the MIT store the configuration and state for a switch element or feature (interfaces, protocols, etc.). NX-API REST provides access to objects stored in the MIT. Managed objects (MOs) are associated with a well-defined REST URI, and can be queried or configured from NX-API REST using their URI.

In the sections below, we examine some of the characteristics of Cisco's NX-API REST interface.

Transactional

NX-API REST operates in a forgiving mode, meaning missing attributes are substituted with default values (if applicable) that are maintained in the internal data management engine (DME). The DME validates and rejects incorrect attributes. NX-API REST is also atomic; if multiple MOs are being configured simultaneously, the API has the ability to stop its operation in the event any of the targeted MOs cannot be configured. It will return the configuration to its previous state, stop the API operation, and return an error code.

NX-API REST is transactional and terminates on a single data model. Developers are relieved from the task of programming and interfacing with individual component configurations.

The API model includes the following programmatic entities:

- **Classes** are templates that define the properties and states of objects in the management information tree.
- **Methods** are actions that the API performs on one or more objects.
- **Types** are object properties that map values to the object state such as `equipmentPresence`.

A typical request comes into the DME and is placed in the transactor queue using a first in, first out (FIFO) scheduler. The transactor retrieves the request from the queue, interprets the re-

quest, and performs an authorization check. After the request is confirmed, the transactor updates the MIT. This complete set of steps is executed for every transaction the DME processes.

Backwards Compatible

Updates to MOs and properties conform to the existing object model, which ensures backward compatibility. If existing properties are changed during a product upgrade, they are managed during the database load after the upgrade. New properties are assigned default values.

Event-Driven

Full event subscription is enabled. When any MO is created, changed, or deleted due to a user or system-initiated action, an event is generated. You can subscribe to notifications for changes that occur to an object through a websocket, and receive proactive notifications back from DME, showing the change that occurred. This is covered in the next section.

Secure

Currently, the following controls for API security functions within NX-API REST are supported and provided by Cisco:

- REST API password-based authentication uses a special subset of request URIs, including `aaaLogin`, `aaaLogout`, and `aaaRefresh` as the DN targets of a POST operation.
- Data payloads are formatted in XML or JSON, and contain the MO representation of an `aaaUser` object with attributes defining the username and password.
- The response to the POST operation will contain an authentication token as both a Set-Cookie header and an attribute to the `aaaLogin` object in the response.
- Subsequent operations on the REST API can use this cookie to authenticate future requests.

Flexible

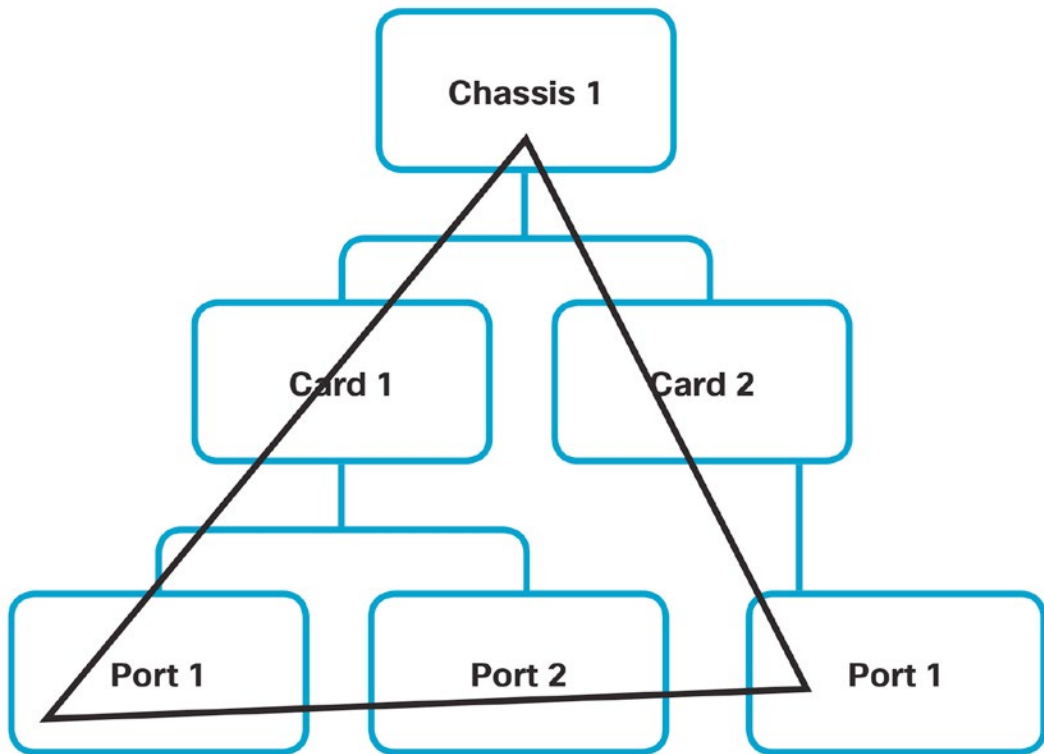
NX-API REST supports a wide range of flexible filters which are useful for narrowing the scope of a search, allowing information to be located quickly. The filters themselves are appended as query URI options starting with a question mark (?) and concatenated with an ampersand (&). Multiple conditions can be joined together to form complex filters.

The [Cisco NX-API REST API User Guide](#) discusses in detail how to use filters and filter syntax while providing examples. Using some of the tools discussed in the following sections, you can build your own query strings and discover those being used by the native Cisco NX-API REST interface.

Tree-Level Queries

The following figure shows a switch chassis that is queried at the tree level.

Figure: Tree-Level Queries

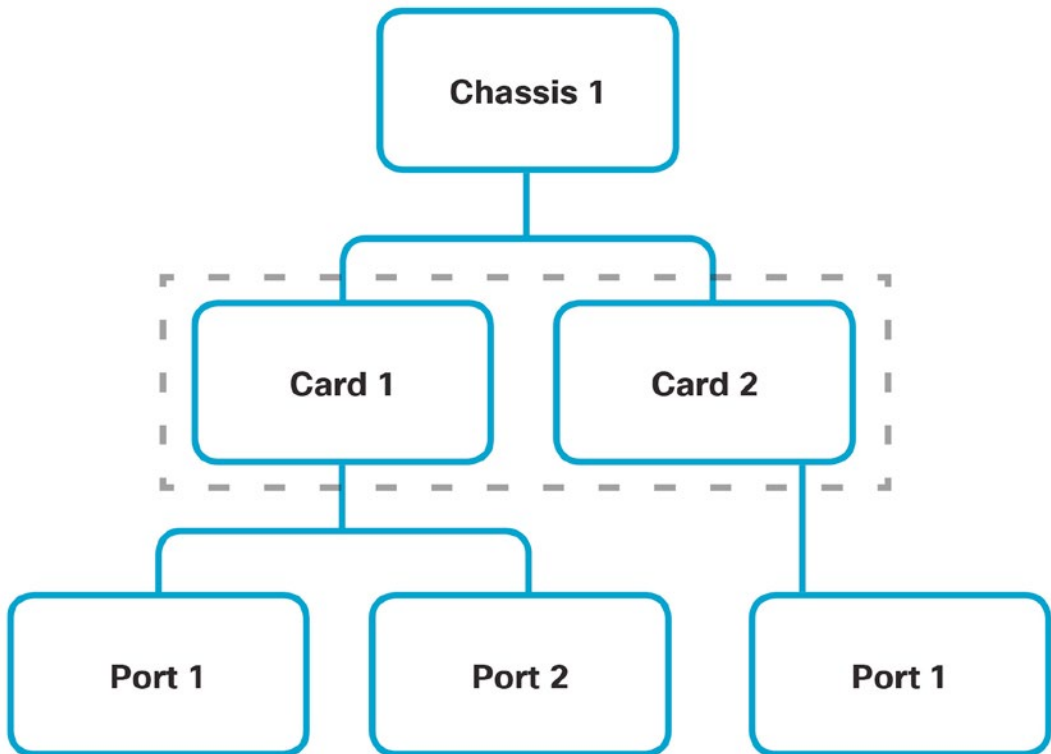


Tree queries return the referenced object and its child objects. This approach is useful for discovering the components of a larger system. In this example, the query discovers the cards and ports of a given switch chassis.

Class-Level Queries

The following figure shows the second query type: the class-level query.

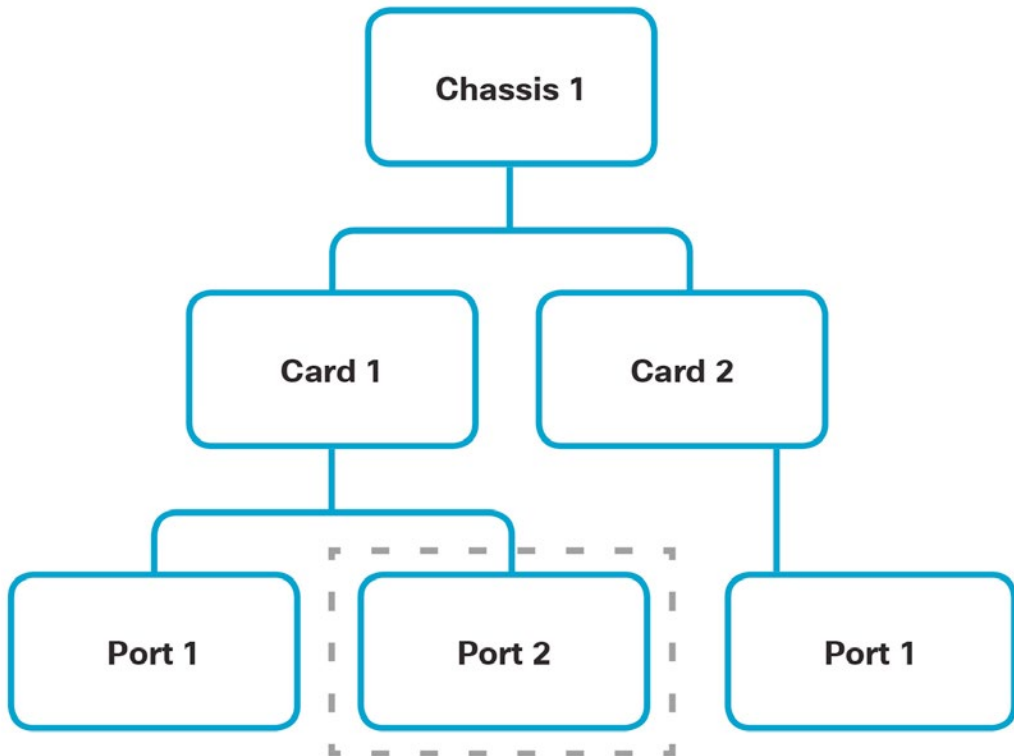
Figure: Class-Level Queries



Class-level queries return all objects of a given class. This approach is useful for discovering all the objects of a certain type that are available in the MIT. In this example, the class used is Cards which returns all objects of type Cards.

Object-Level Queries

The third query type is an object-level query. In an object-level query, a distinguished name is used to return a specific object.

Figure: Object-Level Queries

For all MIT queries, an administrator can optionally return the entire subtree or a partial subtree. Additionally, the role-based access control (RBAC) mechanism in the system dictates which objects are returned; only the objects that the user has rights to view will be returned.

Standards-based

Standard REST methods are supported on the API, which include POST, GET, and DELETE, operations through HTTP.

REST HTTP and HTTPS-Based Methods

Method	Action	Behavior
POST	Create/Update	Idempotent
GET	Read	Nullipotent
DELETE	Delete	Idempotent

The POST and DELETE methods are **idempotent**, meaning they have no additional effect if they are called more than once with the same input parameters. The GET method is **nullipotent**, meaning it can be called zero or more times without making any changes (or that it is a read-only operation).

Payload Encapsulation

Payloads to and from the NX-API REST interface can be encoded with either XML or JSON. The XML encoding operation uses the element tag as the name of the package and class, with the respective properties of that object being specified as attributes of the element. Containment is defined by creating child elements.

The following example creates a BGP instance in an XML payload.

Figure: Create BGP - XML

```
POST http://n9k-sw-1/api/mo/sys/bgp.xml
```

```
<?xml version="1.0" encoding="UTF-8"?>
<bgpEntity adminSt="enabled">
  <bgpInst adminSt="enabled" asn="65000">
    </bgpInst>
  </bgpEntity>
```

JSON encoding requires definition of certain entities to reflect the tree-based hierarchy.

- All objects are described as JSON dictionaries. The key is the name of the package and class while the value is another nested dictionary with two keys: attribute and children.

- The attribute key contains a further nested dictionary describing key-value pairs that define attributes on the object.
- The children key contains a list that defines all the child objects. The children in this list are dictionaries containing any nested objects which are defined as described in the MIT.

The following example creates a BGP instance in a JSON payload.

Figure: Create BGP Peer - JSON

```
POST http://n9k-sw-1/api/mo/sys/bgp/inst/dom-default.json
```

```
{
  "bgpPeer": {
    "attributes": {
      "addr": "192.168.0.2",
      "asn": "65000"
    }
  }
}
```

Both examples have been abbreviated to simplify visual understanding.

Read Operations

After the object payloads are properly encoded as XML or JSON, they can be used in create, read, update, or delete (CRUD) operations through the REST API (Figure 5).

Figure: Read Operations

```
GET http://n9k-sw-1/api/mo/sys/bgp.json
```

Response to the above query:

```
{
  "totalCount": "1",
  "imdata": [
    {
      "bgpEntity": {
```

```

        "attributes": {
            "adminSt": "enabled",
            "childAction": "",
            "dn": "sys/bgp",
            "lcOwn": "local",
            "modTs": "2015-09-30T03:28:52.083+00:00",
            "monPolDn": "uni/fabric/monfab-default",
            "name": "",
            "operErr": "",
            "operSt": "enabled",
            "status": ""
        }
    }
}
]
}

```

Because NX-API REST uses HTTP, defining the universal resource identifier (URI) to access a certain resource type is important. The first two sections of the request URI simply define the protocol and access details (hostname, IP address) of a Cisco Open NX-OS device. Next in the request URI is the literal string "/api", followed by the DN of the object or class being queried. The final part of the request URI is the encoding format: JSON or XML.

The optional part of a request URI consists of any query options. Query options provide filtering capabilities to developers, and are explained extensively in the NX API REST documentation available at <http://developer.cisco.com/open-nxos>.

```
GET http://n9k-sw-1/api/class/l1PhysIf.json
```

Response to the above query:

```

{
  "totalCount": "54",
  "imdata": [
    {
      "l1PhysIf": {
        "attributes": {
          "accessVlan": "vlan-1",

```

```

        "adminSt": "up",
        "autoNeg": "on",
        "descr": "",
        "dn": "sys/phys-[eth1/33]",
        "dot1qEtherType": "0x8100",
        "duplex": "auto",
        "ethpmCfgFailedBmp": "",
        "ethpmCfgFailedTs": "00:00:00:00.000",
        "ethpmCfgState": "0",
        "id": "eth1/33",
        "inhBw": "unspecified",
        "layer": "Layer2",
        "linkDebounce": "100",
        "linkLog": "default",
        "modTs": "2015-06-26T16:04:10.748+00:00",
        "mode": "access",
        "monPolDn": "uni/infra/moninfra-default",
        "mtu": "1500",
        "name": "",
        "speed": "auto",
        "trunkVlans": "",
    }
},
{
    ...
}
]
}

```

The example above shows a query for all objects with class "l1PhysIf". For a complete reference to the various objects and their properties and possible values, please refer to the [Cisco NX-API REST documentation](http://developer.cisco.com/open-nxos) at <http://developer.cisco.com/open-nxos>.

Write Operations

Create and update operations in the REST API are implemented using the POST method, so if an object does not already exist, it will be created. If the object already exists, it will be updated to reflect any changes between its existing state and desired state.

Both create and update operations can contain complex object hierarchies. A complete tree can be defined in a single command as long as all objects are within the same context and root, and are under the 1MB limit for the REST API data payloads. This limit is in place to guarantee performance and protect the system under high load. Very large operations may need to be broken into smaller pieces.

Example: Write Operations

```
POST http://n9k-sw-1/api/mo/sys/bgp/inst/dom-default.json
```

```
{
  "bgpPeer": {
    "attributes": {
      "addr": "192.168.0.2",
      "asn": "65000"
    }
  }
}
```

Create and update operations use the same syntax as read operations, except they are always executed at an object level, as you cannot make changes to every object of a specific class. The create or update operation should target a specific managed object; the literal string `/mo` in a URL indicates the DN of the managed object will be provided, followed by the DN. Filter strings can be applied to POST operations. As an example, if you want to retrieve the results of your POST operation in the response, you can pass the ***rsp-subtree=modified*** query string to indicate you want the response to include any objects that have been modified by the POST operation.

The payload of the POST operation will contain the XML or JSON encoded data representing the managed object being queried.

In summary, REST-based APIs are the most popular APIs today. They are easy to use, well documented and language independent. NX-API REST exposes these benefits while presenting a comprehensive data model for managing network infrastructures.

Cisco NX-API WebSocket Notifications

The Cisco Open NX-API REST interface, described in the previous section, is a very powerful interface for pushing configuration changes or pulling information from the Cisco Nexus switches. However, there might be instances where it might be desirable to receive notifications from the switch directly- for example, when a counter representing unexpected packet errors increments.

Cisco Open NX-OS provides an interface capability to enable the switch to push notifications to interested subscribers. Through the NX-API WebSocket interface, programs and end-users can receive notifications about various state changes on the switch, eliminating the need for periodic polling. The interface establishes full-duplex communication on a single session with a remote entity.

Subscribing to Query Results

When you perform an API query using the Cisco NX-API REST interface, you have the option to create a subscription to any future changes in the results of a given query. When any management object (MO) is created, changed, or deleted, because of a user-initiated or system-initiated action, an event is generated. If the received event changes the results of a subscribed query, the switch generates a push notification to the API client that created the subscription.

Opening a WebSocket

The API subscription feature uses the WebSocket protocol (RFC 6455) to implement a two-way connection with the API client. This way, the API can send unsolicited notification messages to the client itself. To establish the notification channel, you must first open a WebSocket connection with the respective API. Only a single WebSocket connection is needed to support multiple query subscriptions within each switch. The WebSocket connection is dependent on your API session connection, and closes when your API session ends.

Creating a Subscription

To create a subscription to a query, perform the query with the option `?subscription=yes`. This example creates a subscription to a query of the `fv:Tenant` class in the JSON format:

```
GET https://n9k-sw-1/api/class/l1PhysIf.json?subscription=yes
```

The query response contains a subscription identifier, **subscriptionId**, that you can use to refresh the subscription and identify future notifications from the given subscription.

```
{
  "subscriptionId" : "72057611234574337",
  "imdata" : [{
    "l1PhyIf" : {
      "attributes" : {
        "instanceId" : "0:0",
        "childAction" : "",
        "dn" : "sys/phys-[eth1/1]",
        "lcOwn" : "local",
        "monPolDn" : "",
        "description" : "uplink to core-1",
        "replTs" : "never",
        "status" : ""
      }
    }
  ]
}
```

Receiving Notifications

An event notification from the subscription delivers a data structure that contains the subscription ID and the MO description. In this JSON example, a new user has been created with the name "sysadmin5":

```
{
  "subscriptionId" : ["72057598349672454", "72057598349672456"],
  "imdata" : [{
    "aaaUser" : {
      "attributes" : {
```

```

    "accountStatus" : "active",
    "childAction" : "",
    "clearPwdHistory" : "no",
    "descr" : "",
    "dn" : "sys/userext/user-sysadmin5",
    "email" : "",
    "encPwd" : "TUxISkhH$VHyidGgBX0r7N/srt/YcMYTEn5248omnFhNFzZghvAU=",
    "expiration" : "never",
    "expires" : "no",
    "firstName" : "",
    "intId" : "none",
    "lastName" : "",
    "lcOwn" : "local",
    "name" : "sysadmin5",
    "phone" : "",
    "pwd" : "",
    "pwdLifeTime" : "no-password-expire",
    "pwdSet" : "yes",
    "replTs" : "2013-05-30T11:28:33.835",
    "rn" : "",
    "status" : "created"
  }
}
]
}

```

As multiple active subscriptions can exist for a given query, a notification can contain multiple subscription IDs; similar as shown in the example above. Notifications are supported in either JSON or XML format.

Refreshing the Subscription

In order to continue receiving event notifications, you must periodically refresh each subscription during your API session. To refresh a subscription, send an HTTP GET message to the API method **subscriptionRefresh** with the parameter **id**, equal to the **subscriptionId** shown in the example:

```
GET https://n9k-sw-1/api/subscriptionRefresh.json?id=72057611234574337
```

The API will return an empty response to the refresh message unless the subscription has expired.

Note: The timeout period for a subscription is one minute per default. To prevent loss of notifications, you must send a subscription refresh message at least once every 60 seconds.

In summary, WebSocket provides a powerful tool for allowing publisher-subscriber communication for event subscription within the Open NX-OS REST API.

Configuration Management and Automation

Introduction

The concept of Development Operations (DevOps) relates to optimizing the lifecycle of applications and services through collaboration and cooperation between development and operations teams. The desired outcome for DevOps is to facilitate rapid development cycles, increase application scalability and stability, and enable a flexible and agile infrastructure.

Fundamental aspects of DevOps include:

- Simplification of processes and workflows used to deploy infrastructure
- Workload scalability
- Integrated lifecycle management

Infrastructure automation, driven by programmability, is a key enabler for the DevOps transformation. Open NX-OS introduces a broad set of tools, features, and capabilities to facilitate network automation.

The rest of this section will discuss Open NX-OS features that enable automation and DevOps, including:

- Native Linux-based management of Open NX-OS
- Agent-based Configuration Management Systems (Puppet, Chef, etc.)
- Agentless Management Systems (Ansible)
- NX-API REST Programmability
- Automation of switch provisioning operations (POAP, Ignite)

Device Power-On Automation

Open NX-OS provides foundational elements for the automation and configuration life cycle management of a network device. This is essential to initial bootstrapping and provisioning of the NX-OS device. Ongoing life cycle management of device configurations can be accomplished using configuration management agents, programming and open source tools.

The network build and operation lifecycle is divided in three main phases or stages:

- Day-0 – Initial device and network startup
- Day-1 – Incremental configuration, including provisioning of new end-points and workloads
- Day-2 – Monitoring and Visibility

Starting at Day-0 (zero), the network device is brought up with an initial configuration. In general, the network device could be provisioned with all relevant configuration at Day-0, but the focus for initial startup should be on features and functions which change the least over the lifecycle of the network element. Device name, management IP address, and routing process configuration are some examples.

While Host interface and Port-Channel configuration could be part of the Day-0 configuration, most likely not all information will be available at initial network device setup. Configuration of these elements can be automated in later phases.

Day-1 provisioning covers incremental and ongoing configuration changes. During this phase, flexible configuration management and automation allows changes to be accomplished in an efficient way. Management of end-points and segmentation are examples.

The division between Day-0 and Day-1 configuration can be very fluid as the initial configuration can span from simple management access to an extensive configuration to enable a network device to participate in a data center network fabric.

Sample Minimal Switch Configuration (Day-0):

- Switch name

- Admin username and password
- Out-of-Band management interface and routing
- Console access

Extended Switch Configuration (Day-0)

- Inband management
- AAA - Authentication, Authorization and Accounting
- Enabling NX-OS features
- Global switching parameters
- Common routing protocol parameters
- vPC - Virtual Port-Channel domain
- VXLAN VTEP parameters
- Network interfaces

Day-0 or Day-1 configuration

- Access/host interfaces configurations including vPCs
- Tenant/workload configs: VRFs, routes, host facing VLANs
- Additional features

At Day-2, visibility and monitoring become extremely important. In most environments, Day-1 and Day-2 operations run in parallel and extend through the entire lifecycle of the network device, and appropriate tooling is necessary to achieve these tasks efficiently.

Day-0 (zero) Provisioning

Zero-Touch device provisioning is commonly associated with compute devices, but network devices have had this capability for years. However, this capability has been fairly limited until now. Cisco's Power on Auto Provisioning (POAP) was designed to provide advanced Day-0 provisioning capabilities using an extensible framework.

POAP includes the ability to execute Python scripts as part of its workflow - this offers an unparalleled level of flexibility. Today, POAP can download and install additional management

agents and apply specific configurations based on information such as location in a network topology.

A similar approach is achieved by using PXE – Preboot Execution Environment, which uses a process well known in compute environments. PXE has extended its presence into the network as infrastructure devices are increasingly managed more like servers. NX-OS uses iPXE which leverages an open source network firmware based on gPXE/Etherboot. With PXE, we can leverage existing skillsets and infrastructures developed for compute environments to simplify initial device start-up.

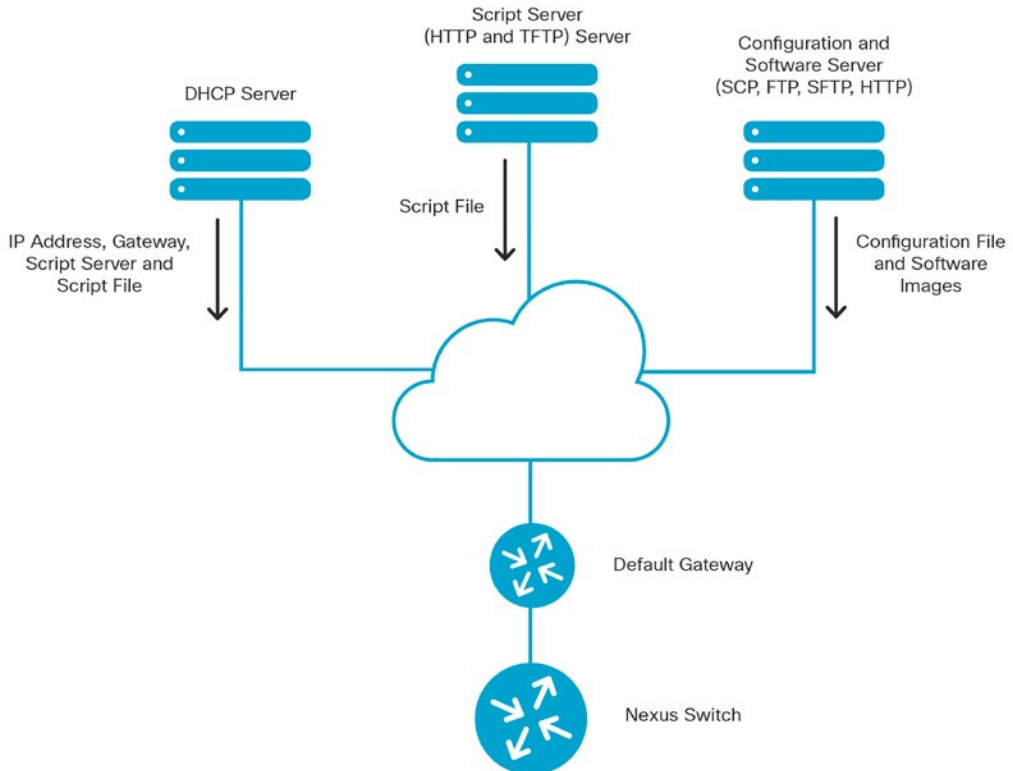
Open source tools like Ignite can also make it easier to automate the Day-0 provisioning.

POAP/PXE Components and Architecture

POAP/PXE can be the first configuration management tools leveraged in the network device lifecycle (Day 0). The initial startup of the network device contains basic access configurations such as the IP connectivity for out-of-band management interfaces, console information, and setting of usernames and passwords.

It could contain more extensive configurations, as described above. POAP and PXE startup processes depends on multiple services that provide different functions. It is possible to host all of these services on the same server, if desired:

- DHCP - Dynamic Host Configuration Protocol to provide necessary information to the device being set up
- A script server for providing the initial configuration script download. TFTP- and HTTP-based script download mechanisms are supported with Open NX-OS
- A configuration and software server hosts a repository of software images, configuration, and additional components. Various transport protocols such as Secure Copy (SCP), File Transfer Protocol (FTP), Secure File Transfer Protocol or Hypertext Transfer Protocol (HTTP) could be supported here

Figure: POAP / Component Architecture

POAP Process

The POAP process starts by assigning a temporary IP address to the switch via the DHCP protocol. Additional DHCP scope options are also provided to facilitate the configuration script download.

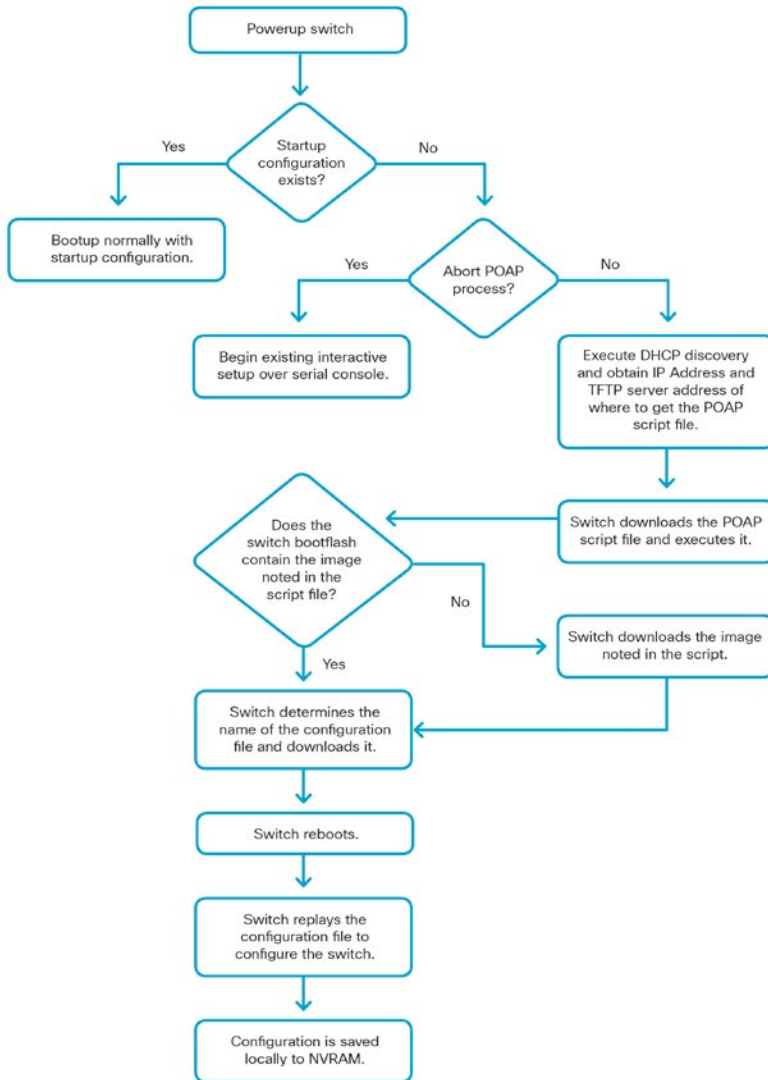
- **Option 66 or Option 150:** References the Script Server. IEEE Option 66 allows a single IP address or Server name. Cisco's Option 150 allows provision of a list of IP addresses for accessing the TFTP-Server

- **Option 67:** References the Configuration Script or Bootfile Name.

The Open NX-OS switch, acting as a DHCP client, will use this information to contact the TFTP server to obtain the configuration script file.

The configuration script (e.g., poap.py) will be executed. The logic of the configuration script will download the software image, switch configuration, agent information and any other additional requirements from the network. POAP provides multiple mechanisms to flexibly identify switches, based on their serial number or system MAC address or their location in the network, as determined by its directly connected neighbors. The downloaded image and configuration is 'scheduled' to be applied after a reboot.

Below is a flowchart representing the POAP process:

Figure: POAP Process Flow Chart

PXE Process

Similar to POAP, the PXE process starts by assigning a temporary IP address to the switch via the DHCP protocol. Additional DHCP scope options are also provided to facilitate the configuration script download.

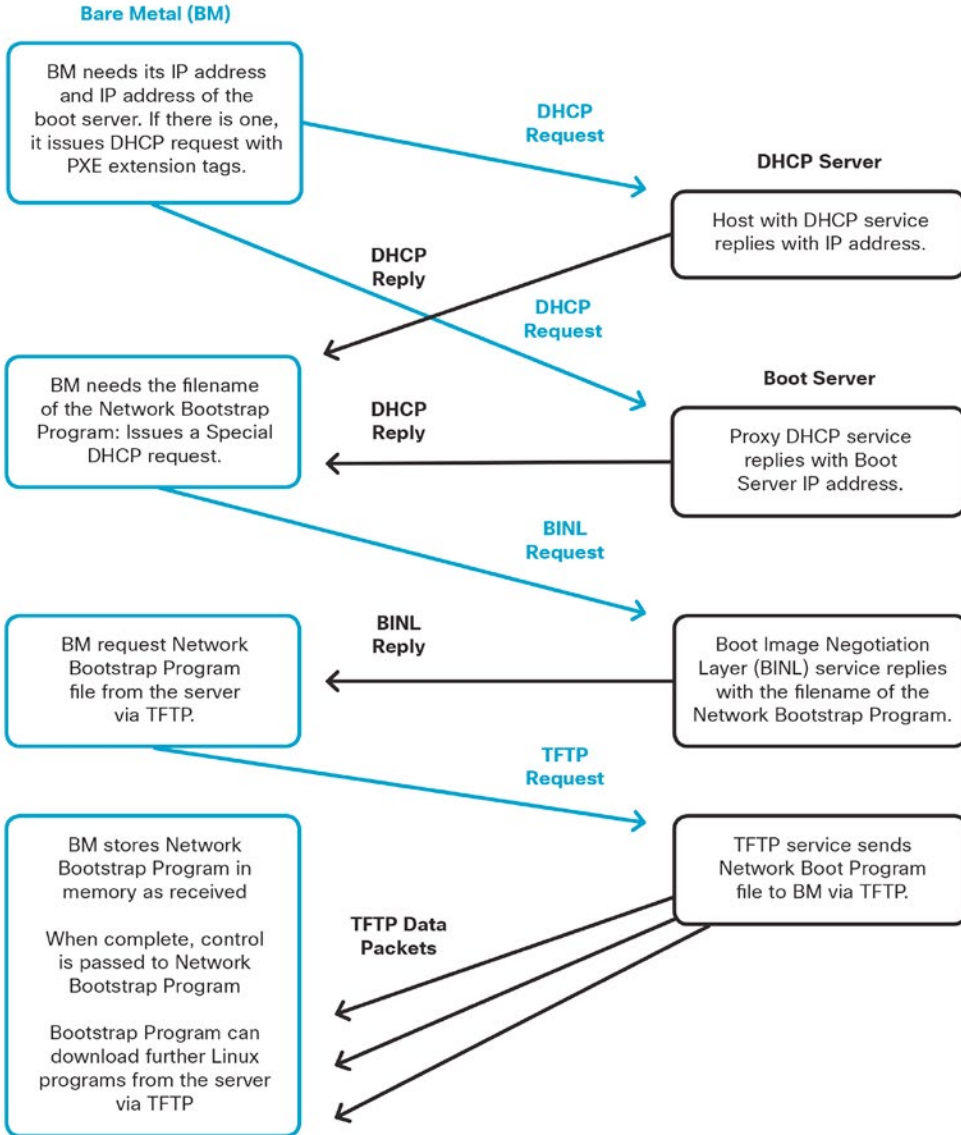
- **Option 66:** This "next-server" options provide the information of a TFTP Server Name that acts as a script server
- **Option 67:** Provides the Configuration Script or Bootfile Name

The Open NX-OS switch, acting as the DHCP client, will download the configuration script and execute it.

PXE based bootstrapping of network devices behaves very similarly to the process on a compute system. While in normal PXE we will start a mini-OS (Network Bootstrap Program), in NX-OS we are using "NETBOOT" to execute the configuration script and respectively fulfill the task of downloading the software image and configuration replay onto the network device.

The new software image and configuration will be used on the next reboot of the network device.

Below is a flowchart representing the PXE process:

Figure: PXE Process Flow Chart

POAP and PXE are great tools for initial start-up of computing systems and network devices. Using these tools at the beginning of a device's lifecycle helps ensure consistency and mitigates erroneous typing or mistakes. Using a streamlined process through network programmability can reduce the risk of such outages.

Further advantages with automated network device on-boarding can be seen during replacement scenarios where archived configurations can be applied to a new device after replacing the unique identifier (ie serial number) in the POAP or PXE definitions.

Loading RPMs and agents using POAP/PXE

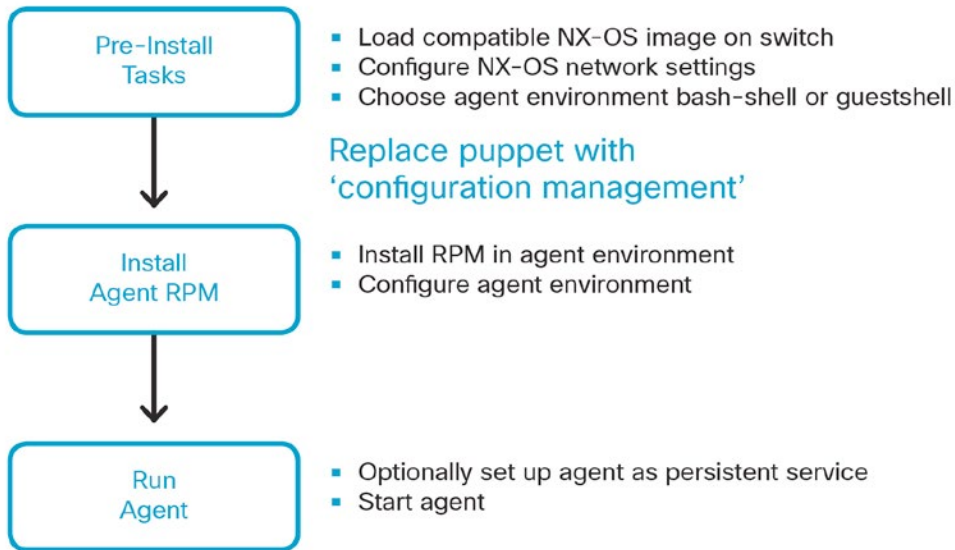
While some Dynamic Configuration Management tools are agent-less and rely on the CLI (Command Line Interface) or REST-based APIs (i.e., NX-API REST), others require a specific agent to be present in order to configure the network device. For the agent-based tools, we want to provide a simplified way to on-board the agent autonomously during startup. The extensibility of POAP and PXE enables going beyond the software image and configuration download for a network device, and can provide the installation of such an agent during initial start-up.

As part of POAP/PXE, we have to extend the process to add configuration management agents to the network device. To include the agent installation in the overall start-up process, a post-processing command-file is required to facilitate the execution and installation of the agent environment.

The following are examples of the post-processing steps needed for enabling agent installation in the network device's bash shell.

- Enable feature bash-shell
- DNS configuration of bash-shell
- RPM installation
- Installation of configuration management agent

Depending on whether the agent is installed in the bash shell or in the guest shell (container), the post-processing command file content will vary.

Figure: POAP/PXE for Dynamic Configuration Management "Agent" onboarding

Day-0 Automation Tool - Ignite

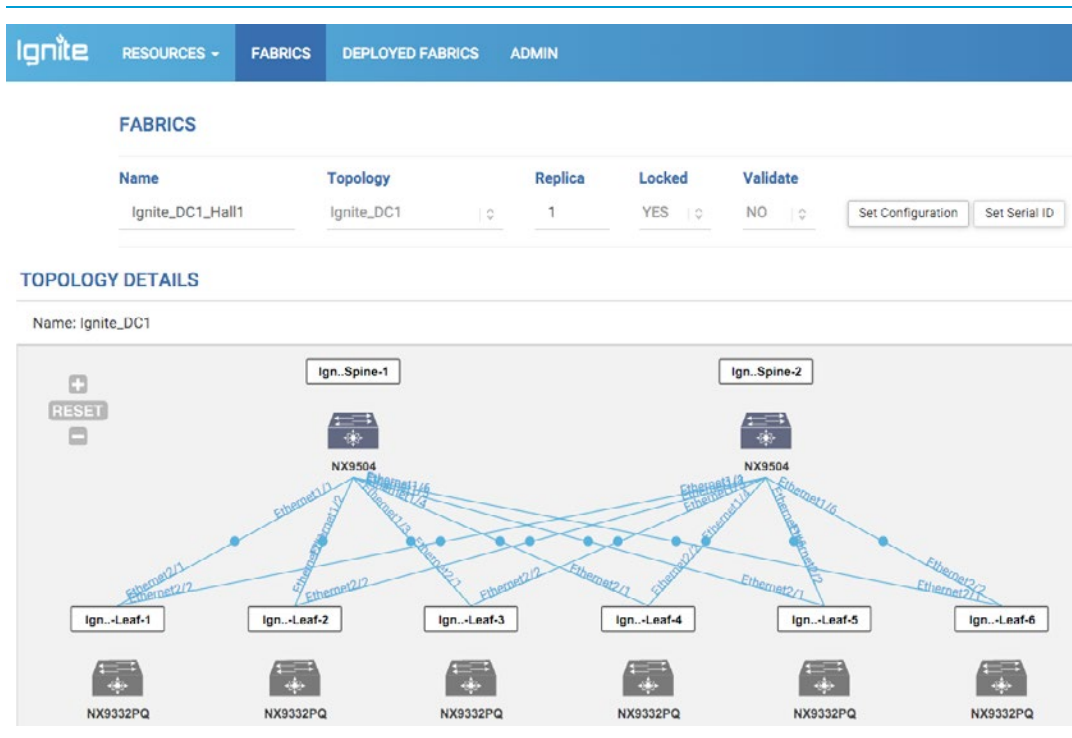
Ignite is an open source tool designed to facilitate initial network boot-strapping and provisioning. It supports Cisco Nexus switches leveraging POAP and, in the near future, PXE capabilities of NX-OS. Ignite provides a powerful and flexible framework through which a data center network administrator can define the data center fabric in terms of its topology, configuration templates (configlets) and resource pools.

Ignite provides the following functionality to help with Day-0 automation:

- Topology design
- Configuration design
- Image and configuration store for POAP
- POAP request handler

Ignite provides a flexible mechanism to identify switches based on their serial number, MAC address, or its location in the network as determined by examining its relationship to peers. Ignite, can then generate a configuration unique to the switch using rules provided by the administrator, configuration templates and configlets, scripts, and resource pools.

Figure: Topology Design with Ignite



Getting Started with Ignite

Information for getting started with Ignite can be found at <http://github.com/datacenter/ignite>

In summary, POAP is an essential tool for Day-0 automation. This can facilitate the deployment of networks of any size. The use of POAP with python scripting provides additional Day 0-1 functionality for further manipulation of configurations.

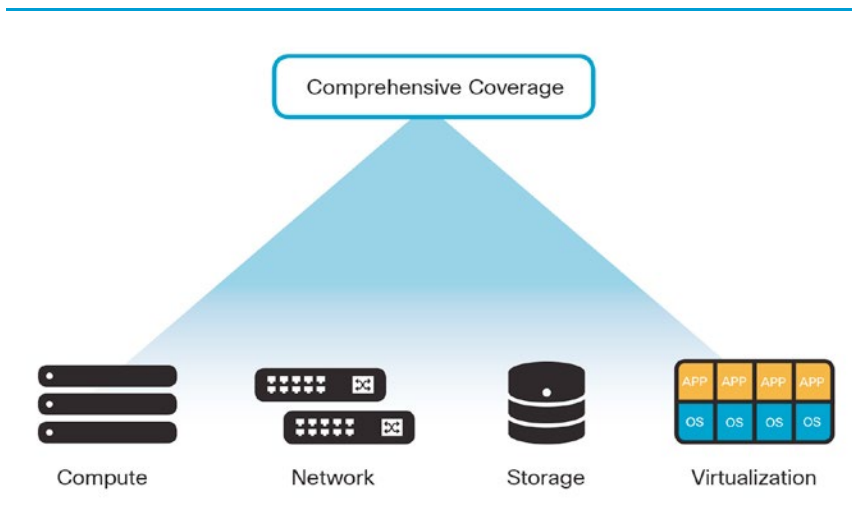
Configuration and Lifecycle Management

Configuration management is part of the larger process of device lifecycle management - from planning and implementing, to operations, and eventual device decommissioning.

Configuration management is one of the most tedious and repetitive operations that will occur throughout the lifecycle of a device, and it is tightly integrated with organizational change management processes. The purpose of configuration management is to build consistent and repeatable processes to implement and verify changes, as well as remediate exceptions or violations found within the infrastructure.

All of this is done to ensure compliance with the best-practice configurations and any organizational or regulatory standards. Configuration management enables consistent configuration of compute, virtualization, networking and storage resources and services. Multiple facets of a company's infrastructure can benefit from these processes, as shown in the the following illustration.

Figure: The Broad Applicability of Configuration Management Infrastructures



Configuration management tools, typically, have a hierarchically-distributed structure built on a declarative approach to management. They most often implement a client-server or master-agent framework, where policies are maintained on the server, and agents ensure the policies are applied on the devices being managed.

Examples of tasks that can be automated include:

- Auto-discovery and provisioning of devices
- Granular discovery of device components
- Policy-based and role-based management of infrastructure
- Configuration, event and power management
- Availability management (redundancy)
- Inventory, operation and licensing status report
- Continuous monitoring of resource utilization and capacity
- Dynamic provisioning of device resources (interface, VLANs, routes, switch/port profiles etc)

There are numerous security-related checks that can be implemented. For example:

- Configuration and policy compliance
- Disabling clear-text access mechanisms
- VLAN and trunk management

Network automation that started at Day-0 with POAP and PXE, can be extended by tools like Puppet, Chef, Ansible, Salt Stack, etc. Leveraging tools for day-to-day management, monitoring and configuration changes, IT automation with dynamic configuration management, can optimize the work of infrastructure operations teams, at the same time mitigating the risk of error-prone keyboard input.

Models: Imperative vs. Declarative

Configuration changes in a network have traditionally been made on a per-device basis using a specific set of procedures and configuration line items.

These devices were configured *imperatively*, that is, the exact steps to achieve the desired end-state were specified. With the imperative model, administrators build a workflow every single

time they want to perform a task, and as more components are added, the workflow grows in size and complexity.

This differs from using a *declarative model* where an administrator models how they would like their environment to look, and the switches, configuration management tool, or a combination thereof decides on how best to implement the requested changes.

One of the main differences between the imperative and declarative models is that administrators in an imperative model are typically required to have deep syntax and context knowledge for the entities they are configuring. Syntax and context can differ based on operating code version. Using a declarative approach administrators can request a change using a broad statement of desired outcome, and the system is responsible for "translating" the desired outcome to the different network elements.

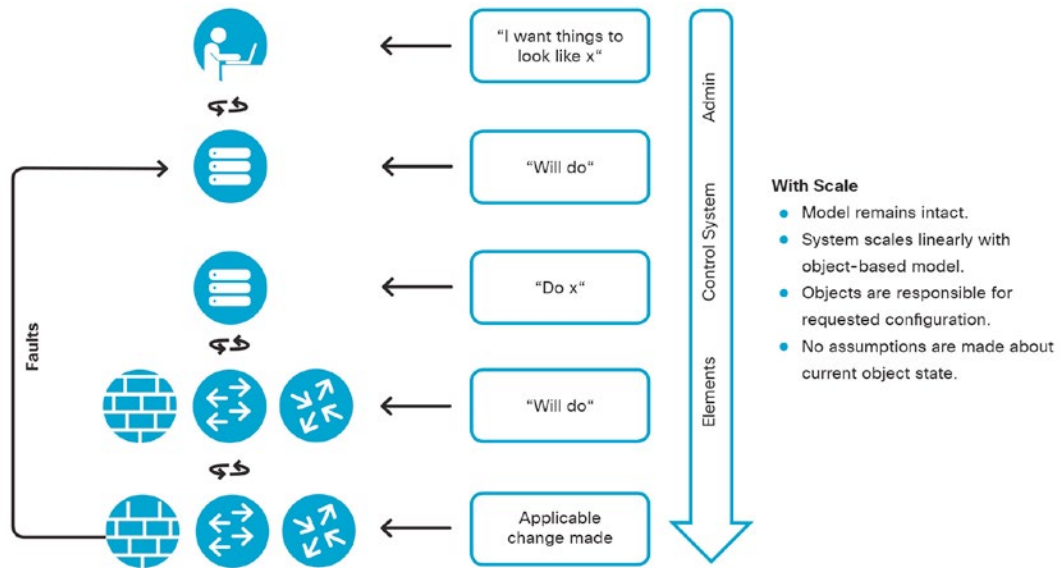
Declarative or model-based configuration management becomes important in the broader data center context, including compute, virtualization, network, storage, and other resources. Architects and engineers are not just modeling the network, they are modelling the entire infrastructure.

From a micro level, the network itself may be heterogeneous; it may have different switch types, levels of code, or even vendors. From a macro level, the network is not the only entity within the data center; configuring a network element on a virtual switch can be drastically different from configuring that same element on a physical switch.

To illustrate the difference between imperative and declarative models, the following workflow is an example of an imperative operation on a switch - configuring a VLAN.

Figure: Imperative Workflow Explicitly Expresses Each Instruction



Figure: Declarative Approach to Infrastructure Management (Promise Theory)

Configuration Management in Open NX-OS

For participation in a configuration management framework, switches must enable configuration management capabilities. Due to differences in configuration management tool methodologies, Open NX-OS supports a variety of access mechanisms, including:

- Secure Shell (SSH)
- Guest agents (installed via RPM), and
- NX-API CLI and REST APIs

Secure Shell (SSH)

Configuration management tools that utilize an embedded/direct payload use Secure Shell (SSH). The following command will confirm that SSH is enabled on a switch:

```
n9k-sw-1# show feature | inc ssh
sshServer          1          enabled
```

Guest Agent Installation using Bash/RPM

Open NX-OS allows installation and configuration of management agents through the bash or guest shell. To learn about the differences between bash and guest shells, and recommended implementations of each, please see the *Third-party Application Integration* chapter.

NX-API CLI and REST APIs

For configuration management tools and other architectures such as cloud management platforms, the NX-API may be utilized for device configuration, gathering device data, and performance monitoring.

For more information see <http://developer.cisco.com/opennxos>

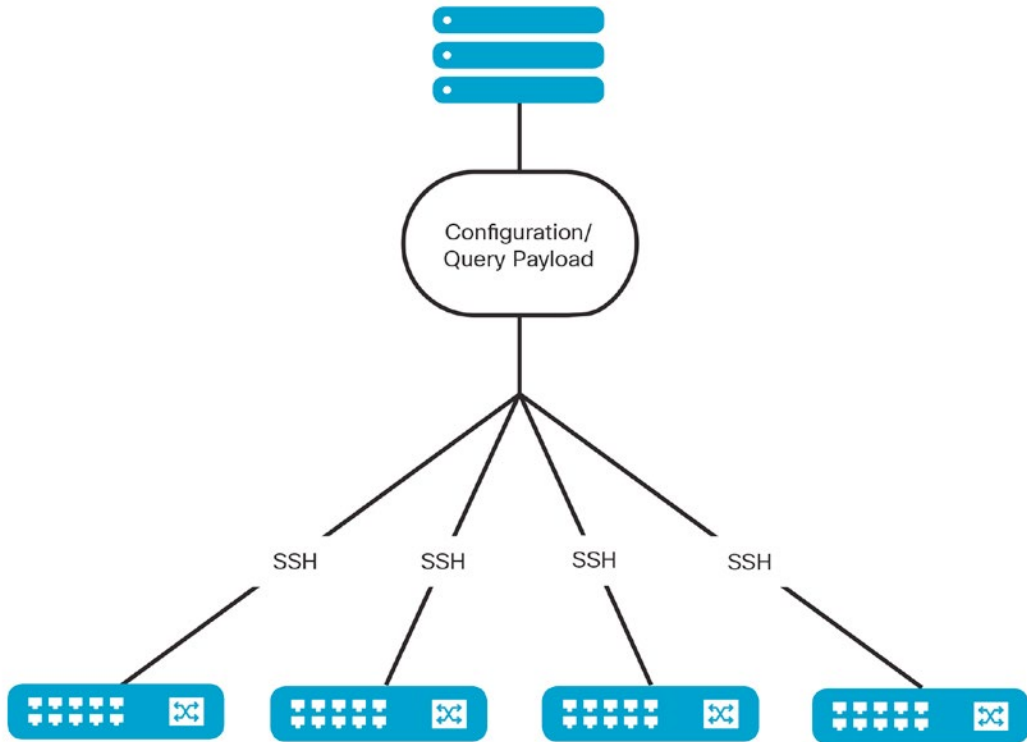
Agent-less Management

Native or agent-less management is accomplished through remote shell access or through the NX-API. With remote shell access, a configuration management server or master will utilize a push model to deliver a payload in the form of a script through the network. The configuration management scripts are stored on the master.

As an example, a Python script can be used to gather switch information and statistics and return results to the originating server. The script will run in a temporary space on the switch, and any relevant data will be streamed back or compressed and returned through the network.

A major advantage to the remote shell access method is that there is little to no configuration required on the switch device as there are no agents required for installation. A potential drawback is the need to ensure that the security configuration on the switch is kept synchronized, as any change can have a significant impact on the configuration management tool's ability to access the switch.

Ansible is a good example of an agent-less configuration management tool.

Figure: Agent-less Models Utilize Standard Device Interfaces like Secure SHell (SSH)

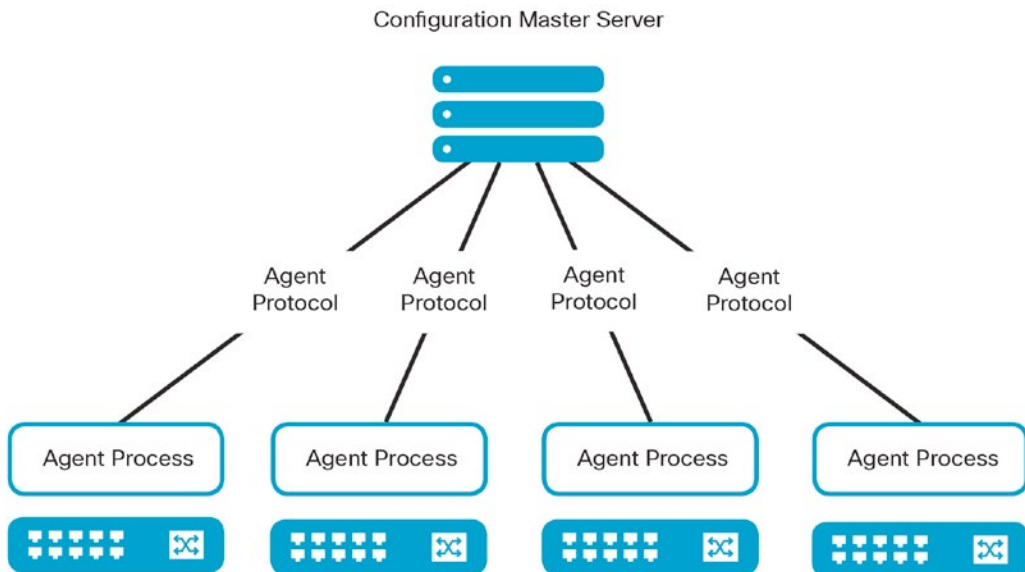
Agent-based Management

Agent-based configuration management is pull-based, and requires installation of an agent on the switch. When agents are used for switch management, they can be installed in the native Linux user-space, a Linux Container (LXC), or the guest shell. Communication between agents and master nodes is accomplished through the network, and agents should be configured for secure, encrypted communication between master nodes and agents.

While bash will run agents directly within the shell, guest shell has the capability to install agents within an LXC, or to run an open virtual format (OVF/OVA) machine, which can be launched from the bootflash directory. Guest shell ensures agent-based operations are isolated from the underlying host context to ensure security

Puppet and Chef are examples of agent-based configuration management tools.

Figure: Puppet and Chef Leverage an Agent on the Open NX-OS Switch



In summary, configuration management tools have been proven in server automation environments, and can be extended to network infrastructure components using a common configuration management framework.

IT Automation Tools

The open source community has several toolsets for configuration management. Many of these toolsets also offer enterprise-level product management and support. Cisco offers support for several open source configuration management tools, including:

- Ansible
- Chef
- Puppet

Open NX-OS also offers developers and administrators the ability to create and share their own code in Python, Ruby, Go or any programming capable of accessing NX-API REST.

Ansible

Introduction

Ansible is an open-source software platform for configuring and managing compute and switching infrastructure using “playbooks”. Ansible features a state-driven resource model that describes the desired state of computer systems and services. It is used to automate the configuration of a company’s compute and switching resources in an agent-less manner.

Ansible is a very straightforward and powerful tool for intent-based network automation, all you need to get started are playbooks, a server configuration file, and an inventory file.

Key Technical Concepts

Playbooks

Playbooks specify a list of tasks that are run in sequence across one or more hosts. Each task can also run multiple times with a variable taking a different value. Playbooks are expressed in YAML format.

Inventory

Inventory is the representation of information about hosts – what groups a host belongs to, the properties those groups and hosts have. A hierarchy of groups often results.

Templates

Templates allow you to generate configuration files from values set in various inventory properties. This means that you can store one template in source control that applies to many different environments.

Roles

Roles are a way to encapsulate common tasks and properties for reuse, if you find yourself writing the same tasks in multiple playbooks, turn them into roles.

Sample Playbook To Configure VLANs:

```
cisco@linux-dev:~/nxos-ansible/ansible_playbooks$ more vlans.yml
# vlans.yml
---

- name: VLANs
  hosts: all
  connection: local
  gather_facts: no

  tasks:
    - name: ensure VLANs 2-20 and 99 exist on all switches
      nxos_vlan: vlan_id="2-20,99" state=present host={{ inventory_hostname }}

    - name: config VLANs names for a few VLANs
      nxos_vlan: vlan_id={{ item.vid }} name={{ item.name }} host={{ inventory_hostname }}
state=present
  with_items:
    - { vid: 2, name: web }
    - { vid: 3, name: app }
```

```
- { vid: 4, name: db }  
- { vid: 20, name: server }  
- { vid: 99, name: native }
```

Ansible Reference Links

<https://github.com/datacenter/nxos-ansible>

<http://docs.ansible.com/ansible/>

Chef

Introduction

Chef is a powerful automation platform that transforms complex infrastructure into code, enabling your data center infrastructure automation using a declarative, intent-based model. Whether you're operating in the cloud, on-premises, or a hybrid, Chef automates how applications are configured, deployed, and managed across your network, no matter its size.

Chef is built around simple concepts: achieving desired state, centralized modeling of IT infrastructure, and resource primitives that serve as building blocks. These concepts enable you to quickly manage any infrastructure with Chef. These very same concepts allow Chef to handle the most difficult infrastructure challenges and customer use-cases, anything that can run the chef-client can be managed by Chef.

Key Technical Concepts

Chef Server

The Chef server acts as a hub for configuration data. It stores:

- Cookbooks
- Recipes (The policies that are applied to nodes)
- Metadata that describes each registered node that is being managed by the chef-client.

Node

Any physical, virtual, or cloud machine or switch configured to be maintained by a chef-client.

Chef Client

Runs locally on every node that is registered with the Chef server. Performs all configuration tasks specified by the run-list and brings client into desired state.

Chef Resources

Term used for a grouping of managed objects/attributes and one or more corresponding implementations. It describes the desired state for a configuration item and declares the steps needed to bring that item to the desired state. It specifies a resource type—such as a package, template or service, and lists additional details (also known as attributes), as necessary. These are grouped into recipes, which describe working configurations

The 2 core layers of a resource:

- Resource Type: Definition of managed objects.
- Resource Provider: Implementation of management tasks on objects.

Cookbook

A cookbook defines a scenario and contains everything that is required to support that scenario, and is used for device configuration and policy distribution:

- Recipes that specify the resources to use and the order in which they are to be applied
- Attribute values
- File distributions
- Templates
- Extensions to Chef, such as libraries, definitions, and custom resources

Recipe

A collection of resources, defined using patterns (resource names, attribute-value pairs, and actions); helper code is added around this using Ruby:

- Must be stored in a cookbook
- May use the results of a search query and read the contents of a data bag
- May have a dependency on one (or more) recipes
- Must be added to a run-list before it can be used by the chef-client
- Is always executed in the same order as listed in a run-list
- The chef-client will run a recipe only when asked

Sample Cookbook Showing Configuration of Switch Interface as L3 or L2:

```
cisco_interface 'Ethernet1/1' do
  action :create
  ipv4_address '10.1.1.1'
  ipv4_netmask_length 24
  ipv4_proxy_arp true
  ipv4_redirects true
  shutdown true
  switchport_mode 'disabled'
end

cisco_interface 'Ethernet1/2' do
  action :create
  access_vlan 100
  shutdown false
  switchport_mode 'access'
  switchport_vtp true
end
```

Chef Reference Links

Cisco Chef Client: ([WRL5 Agent](#), [CentOS7 Agent for Guest Shell](#))

Cisco Chef Cookbook: (<https://supermarket.chef.io/cookbooks/cisco-cookbook>)

Cisco Chef Cookbook Source Repository (<https://github.com/cisco/cisco-network-chef-cookbook>)

List of Supported Cisco Resources (<https://github.com/cisco/cisco-network-chef-cookbook/>

<blob/develop/README.md#resource-by-tech>)

Puppet

Introduction

Puppet models desired system states, enforces those states, and reports any variances so you can track what Puppet is doing. To model system states, Puppet uses a declarative resource-based language - this means a user describes a *desired final state* (e.g. “this package must be installed” or “this service must be running”) rather than describing a series of steps to execute.

You use the declarative, readable Puppet DSL (Domain Specific Language) to define the desired end-state of your environment, and Puppet converges the infrastructure to the desired state. So if you have a pre-defined configuration that every new switch should receive, or need to make incremental configuration changes to your infrastructure, or even have a need to install third party software agents, the puppet intent-based automation solution can automate these repetitive configuration management tasks quickly. We support both open source and puppet enterprise with Open NX-OS.

Key Technical Concepts

Puppet Server

The puppet server acts as a central server or point of configuration and control for your data-center, both switching and compute:

- Manifests
- Resources

Node

Any physical, virtual, or cloud machine or switch configured to be maintained by a puppet client, be it server or switch.

Puppet Client

Runs locally on every node that is managed by the puppet master server. Performs all configuration tasks specified by the manifest and converges the node into desired state.

Resources

Puppet ships with a number of pre-defined *resources*, which are the fundamental components of your infrastructure. The most commonly used resource types are files, users, packages and services. You can see a complete list of the Cisco resource types at the Github repository link below.

Puppet revolves around the management of these resources. The following code sample ensures a third party agent service is always up and running.

```
service { 'tcollector':  
  ensure => running,  
}
```

Here, the “service” is the resource type, and “tcollector” is the managed resource. Each resource has attributes, and here, “ensure” is an attribute of the tcollector service. We’re setting the “ensure” attribute to “running” to tell Puppet that tcollector should always be running, and to start the service if it is not.

Manifests

Introduction

Manifests are files containing Puppet code. They are standard text files saved with the `.pp` extension. Most manifests should be arranged into [modules](#).

Resources

The core of the Puppet language is declaring **resources**. A resource declaration looks like this:

```
# A resource declaration:  
cisco_interface { "Ethernet1/2" :
```

```
description => 'default',  
shutdown   => 'default',  
access_vlan => 'default',  
}
```

When a resource depends on another resource, you should explicitly state the relationship to make sure they occur in the proper order.

Classes

A class is a set of common configurations – resources, variables and more advanced attributes. Anytime we assign this class to a machine, it will apply those all configurations within the class.

Modules

A Puppet module is just a collection of files and directories that can contain Puppet manifests, as well as other objects such as files and templates, all packaged and organized in a way that Puppet can understand and use. When you download a module from PuppetForge, you are downloading a top-level directory with several subdirectories that contain the components needed to specify the desired state. When you want to use that module to manage your nodes, you classify each node by assigning to it a class within the module.

The hierarchy is as follows:

- Resources can be contained within classes.
- Classes can live in a manifest.
- Manifests can live in a module.

Sample Manifest

Three types are needed to add OSPF support on an interface: `cisco_ospf`, `cisco_ospf_vrf`, and `cisco_interface_ospf`.

First, to configure `cisco_ospf` to enable ospf on the device, add the following type in the manifest:

```
cisco_ospf {"Sample":  
  ensure => present,
```

```
}

```

Then put the ospf router under a VRF, and add the corresponding OSPF configuration.

If the configuration is global, use 'default' as the VRF name:

```
cisco_ospf_vrf {"Sample default":
  ensure => 'present',
  default_metric => '5',
  auto_cost => '46000',
}
```

Finally, apply the ospf configuration to the interface:

```
cisco_interface_ospf {"Ethernet1/2 Sample":
  ensure => present,
  area => 200,
  cost => "200",
}
```

Puppet Reference Links

Puppet Module: (<https://forge.puppetlabs.com/puppetlabs/ciscopuppet>)

Puppet Module Source Repository (<https://github.com/cisco/cisco-network-puppet-module>)

List of Supported Cisco Resources (<https://github.com/cisco/cisco-network-puppet-module/blob/develop/README.md#resource-by-tech>)

Common Artifacts

Ruby Libraries (https://rubygems.org/gems/cisco_nxapi, https://rubygems.org/gems/cisco_node_utils)

Ruby Library Source Repository (<https://github.com/cisco/cisco-network-node-utils>, <https://github.com/cisco/cisco-nxapi>)

Any configuration management switch-resident agents including Puppet and Chef, are supported natively in Open NX-OS Linux, or in the guest shell, providing choice for automation agent installation.

In summary, configuration and lifecycle management are core tenets of the DevOps model, and there are multiple open source tools available for configuration management and automation that Open NX-OS can leverage.

Practical Applications of Network Programmability

Introduction

Up to now, the focus has been on network automation concepts, technologies, tools, and development methodologies. In the sections that follow, the focus will shift to practical use-cases which illustrate how network programmability and automation can be applied to drive innovation and address real issues affecting network operations teams.

In each of the use-cases described below, the challenge affecting the network operator will be described, and a solution that leverages network automation will be illustrated, along with sample APIs, scripts, tools, code-snippets, and configuration-snippets.

These use-cases and their solutions are drawn from real-world customer scenarios. Hopefully they will spark ideas for other use-cases and new solutions that can be shared with the Open NX-OS community.

Infrastructure Provisioning Automation

Problem Statement

Initial startup of a network involves distinct network planning, provisioning and validation phases. With the manual approach to these processes that has been prevalent to date, network operators have been forced to spend less time planning and more time performing tedious configuration and validation tasks.

Automating these tasks would deliver immediate benefits, including:

- **Faster initial setup:** Reducing the time needed to get infrastructure ready for the applications
- **Fewer errors:** Eliminating manual processes with well planned network blueprints
- **Repeatability:** Leveraging automation tools and scripts across multiple deployments

Network startup is a complex process involving numerous configuration and validation steps. The following is an abridged example of the configuration steps necessary to deploy a new network. The whole process could take multiple days extending to weeks, depending on the size and complexity of the network. In this example there are 100 nodes.

Infrastructure Setup (x1):

- Start-up and configure DHCP infrastructure
- (Optional) Configure and enable infrastructure services like AAA servers, SNMP servers, syslog server, etc

Switch Deployment (x 100):

- Physical Installation:
 - Rack the switch
 - Connect management (in-band or out-of-band) interfaces
 - (Optional) connect a console connection to a terminal server
 - Power up the switch

- Switch Configuration:
 - Configure management parameters on the switch
 - Validate management connectivity
 - Validate if the right image exists on the switch
 - Upgrade/downgrade the image, as needed
 - Configure AAA parameters
 - Validate AAA and user access
 - Configure infrastructure features SNMP, syslog, etc.
 - Validate infrastructure features
 - Configure fabric interfaces with appropriate VLANs or IP addresses
 - Validate connectivity to adjacent switches
 - Configure connectivity to external / default routers
 - Validate connectivity to external / core networks
 - Configure connectivity to the Layer 4-7 service devices
 - Validate Layer 4-7 service connectivity
 - Configure Layer 2/3 interfaces and protocols
 - Validate protocol reachability
 - Install configuration agents (Puppet, Chef, Splunk forwarder, etc.)

Fabric Wide (x1):

- *Validate node-to-node and end-to-end connectivity*

The problem with this manual configuration exercise is each of the configuration steps above must be done by hand - 100 times - which is extremely time-consuming and error-prone.

Solution

Cisco NX-OS includes a power-on automated provisioning (POAP) capability that aids in automating the initial start-up and configuration processes. In addition, Cisco has recently published an open source tool called Ignite that further enhances automation with tools for topology, configuration design and management.

Cisco Power-On Auto Provisioning (POAP)

Cisco's POAP implementation allows automated configuration of a network device by providing the required image and configuration via download.

While POAP was originally targeted at initial configuration, it is implemented using a Python-based scripting mechanism that is extremely flexible, so its functionality can be extended well beyond power-on provisioning. Currently POAP is already capable of handling complex requirements for intermediate boot orders, replay checkpoints, and configuration management agent installation.

A broader discussion on POAP is included in the *Configuration Management and Automation* chapter.

Open NX-OS Extensibility Support

POAP utilizes the Open NX-OS RPM (Puppet, Chef) capabilities to install configuration management agents during power-on installation.

Ignite

Ignite is a tool that automates the process of installing and upgrading software images and installing configuration files on Cisco Nexus switches that are being deployed in the network, working in conjunction with POAP on the switch.

Ignite provides a powerful and flexible framework through which a data center network administrator can define data-center network topologies, configuration templates (configlets), and resource pools. Ignite automatically identifies the switch from its neighbors or by its unique identifiers (Serial ID or System MAC address) and then delivers the the proper configuration template and image to the switch.

Using these tools, users can automate a significant portion of their switch start-up process, as illustrated below:

Solution Approach

Infrastructure setup (x1):

- Start and configure the DHCP infrastructure
- (Optional) Configure and enable infrastructure services like AAA servers, SNMP servers, syslog server, etc
- Install and configure Ignite
 - Create IP/VLAN pools
 - Design the configuration templates for the leaf and spine switches
 - One single template is needed per switch category (leaf, spine, core, etc)
 - Design the topology for the fabric
 - Assign image version and configuration templates to the switches
- Configure the DHCP server to redirect DHCP requests from the switches to Ignite

Figure: Sample DHCP-Server configuration for POAP (Linux dhcpd)

```
##ScopeStart:ScopeName:POAP_Scope
subnet 10.10.10.0 netmask 255.255.255.0 {
    range 10.10.10.40 10.10.10.50;
    max-lease-time 3600;
    option bootfile-name "poap.py";
    option domain-name-servers 10.10.10.250;
    option routers 10.10.10.1;
    option tftp-server-name "10.10.10.200";
}
```

Figure: Design Pools of Resources

Name	Type	Scope
LPBK0_IP	IP	global

Start Range	End Range
10.1.1.1	10.1.1.1
20.1.1.1	20.1.1.1
30.1.1.1	30.1.1.1
40.1.1.1	40.1.1.1
50.1.1.1	50.1.1.1
60.1.1.1	60.1.1.1
70.1.1.1	70.1.1.1

Figure: Design Configuration Template Leaf Switches with Ignite

Ignite | RESOURCES | FABRICS | DEPLOYED FABRICS | ADMIN

CONFIGURATION

Name: lableaf_fabric

LIST OF CONSTRUCTS

#	Construct	Template/Script	Parameters
1	append_configlet	adminuser	ADMIN_PASSWORD = cisco123
2	append_configlet	hostname	HOST_NAME = Instance.SWITCH_NAME
3	append_configlet	feature	
4	append_configlet	mgmt	MGMT_IP = Pool.MGMT_IPv4
5	append_configlet	vrf-management	
6	append_configlet	loopback0	LOOPBACK0 = Pool.LPBK0_IP
7	append_configlet	anycastp	
8	append_script	trunkportleaf	TRUNK_PORTS = Instance.TRUNK_PORTS

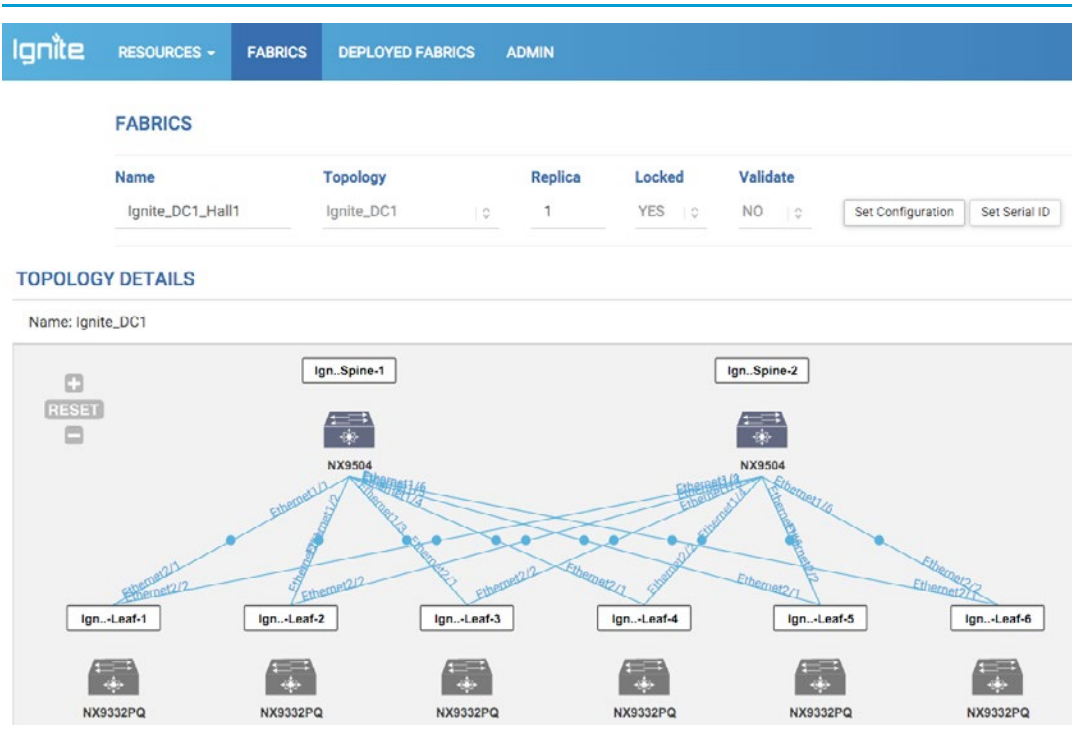
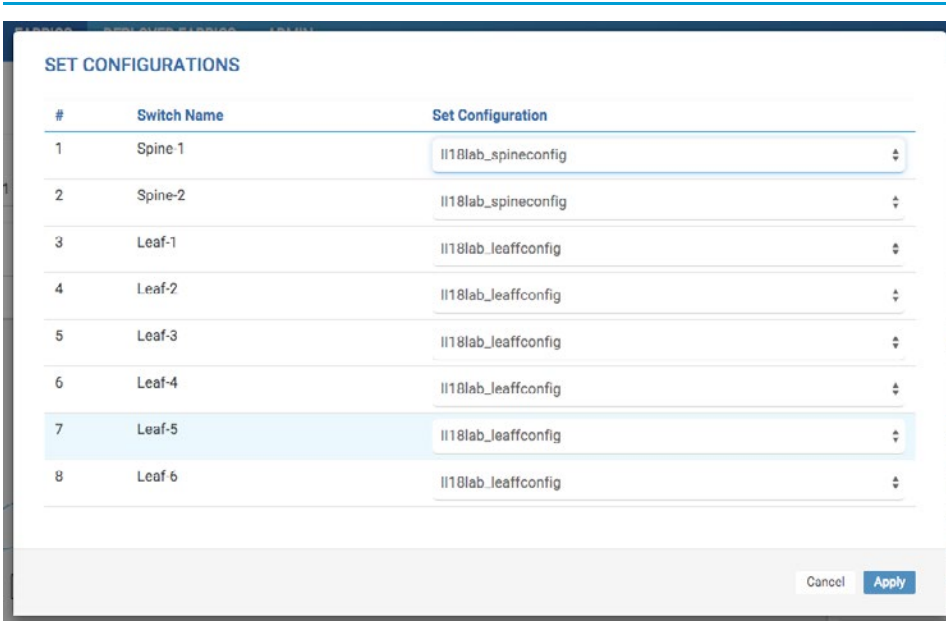
Figure: Design topology for the fabric with Ignite

Figure: Assigning Configuration Templates to Switches


The screenshot shows a web interface titled "SET CONFIGURATIONS" with a table listing switches and their assigned configuration templates. The table has three columns: "#", "Switch Name", and "Set Configuration". There are 8 rows of data. The 7th row is highlighted in light blue. At the bottom right of the interface, there are "Cancel" and "Apply" buttons.

#	Switch Name	Set Configuration
1	Spine-1	II18lab_spineconfig
2	Spine-2	II18lab_spineconfig
3	Leaf-1	II18lab_leaffconfig
4	Leaf-2	II18lab_leaffconfig
5	Leaf-3	II18lab_leaffconfig
6	Leaf-4	II18lab_leaffconfig
7	Leaf-5	II18lab_leaffconfig
8	Leaf-6	II18lab_leaffconfig

Switch setup (x100):

- Rack the switch
- Connect the management (in-band or out-of-band) interfaces
- Connect a console connection to a terminal server (Optional)
- Power-on the switch

Each switch, upon being booted, completes the following steps automatically:

- Issues a DHCP request and receives an IP address
- Contacts the Ignite server and downloads the assigned image and configuration
 - Ignite generates configurations based on the defined configuration templates and pools
- Reboots with the proper image and configuration (Figure 5)
- Installs and configures the configuration agents (Puppet, Chef, Splunk Forwarder)

Figure 5: Switch booting up with POAP

```

[7:51:26] - INFO: Get serial number: ABC1234ab
[7:51:26]S/N[ABC1234ab] - INFO: device type is n9k
[7:51:26]S/N[ABC1234ab] - INFO: device os version is 7.0(3)I1(2)
[7:51:26]S/N[ABC1234ab] - INFO: device system image is n9000-dk9.7.0.3.I1.2.bin
[7:51:26]S/N[ABC1234ab] - INFO: check free space
[7:51:26]S/N[ABC1234ab] - INFO: free space is 3881536 kB
[7:51:26]S/N[ABC1234ab] - INFO: Ready to copy protocol scp, host 10.10.10.30, source
/server-list.cfg
[7:51:29]S/N[ABC1234ab] - INFO: Get Device Image Config File
[7:51:29]S/N[ABC1234ab] - INFO: removing tmp file /bootflash/server-list.cfg
[7:51:29]S/N[ABC1234ab] - INFO: Ready to copy protocol scp host
10.10.10.30/ABC1234ab/recipe.cfg recipe.cfg
[7:51:33]S/N[ABC1234ab] - INFO: Get Device Recipe
[7:51:33]S/N[ABC1234ab] - INFO: removing tmp file /bootflash/recipe.cfg
[7:51:35]S/N[ABC1234ab] - INFO: Download CA Certificate
[7:51:35]S/N[ABC1234ab] - INFO: Ready to copy protocol scp, host 10.10.10.30, source
/cacert.pem
[7:51:38]S/N[ABC1234ab] - INFO: create_image_conf
[7:51:51]S/N[ABC1234ab] - INFO: Ready to copy protocol scp, host 10.10.10.30, source
/n9000-dk9.7.0.3.I2.1.bin
[7:53:13]S/N[ABC1234ab] - INFO: Completed Copy of System Image
[7:53:35]S/N[ABC1234ab] - INFO: Ready to copy protocol scp, host 10.10.10.30, source
/ABC1234ab/device-config
[7:53:38]S/N[ABC1234ab] - INFO: Completed Copy of Config File
[7:53:38]S/N[ABC1234ab] - INFO: Split config invoked....

```

Fabric Wide (x1):

- **Validate node-to-node and end-to-end configuration and connectivity**
 - (Future) Trigger a validation in Ignite to verify configuration and connectivity

Conclusion

Using POAP and Ignite, users can dramatically reduce the time required to prepare the network for applications while reducing the possibility of configuration errors. Further, any effort applied to design one Pod/Fabric can be re-leveraged to accelerate future deployments.

Automating Access Configuration with Ansible

Problem Statement

Users perform manual Day-1 operations to configure network elements repeatedly for the purpose of on-boarding new workloads. One common category of Day-1 configuration activity is performing routine VLAN related operations:

- Check if VLAN exists
- Change names and descriptions of VLANs
- Configure a VLAN

It would be beneficial to automate this repetitive configuration with a goal of reducing time and effort needed to complete the configurations, while reducing the probability of errors.

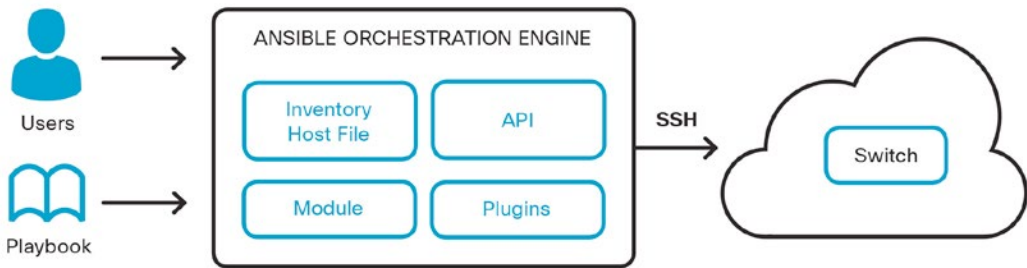
Today networks are in high demand and change constantly due to proliferation of customer demands. Network engineers need the ability to automate their network in a simple and manageable way. One of the most frequent network changes is creation and removal of VLANs where network engineers are accommodating dynamic customer demands.

Solution

Ansible provides a clean way to accomplish the creation and removal of VLANs because Ansible doesn't require an agent be installed on the devices. The main requirements for this solution are:

- ◦ SSH
- ◦ Python – this is not required but it is the most commonly used language

In this particular example we are going to demonstrate how to leverage Ansible to create VLANs. Below is a graphical representation of how Ansible relates to the network.

Figure: Ansible Workflow with a Cisco Open NX-OS Switch

To understand the solution approach, it might be beneficial to understand some of the key Ansible terminology:

- **Hosts:** Remote machines Ansible manages.
- **Groups:** Group of hosts assigned to a pool that can be conveniently targeted and managed together.
- **Inventory:** File describing the Hosts and Groups in Ansible.
- **Modules:** Modules (also referred to as “task plugins” or “library plugins”) are the components that do the actual work in Ansible. They are what gets executed in each playbook task.
- **Playbooks:** A collection of plays which the Ansible Engine orchestrates, configures, administers, or deploys. These playbooks describe the policy to be executed to the host(s). People refer to these playbooks as "design plans" which are designed to be human- readable and are developed in a basic text language called YAML.

Solution Approach

1) Host File

The hosts file leveraged for this specific use-case:

```
cisco@linux-dev:~/nxos-ansible$ more hosts
[all:vars]
ansible_connection = local

[spine]
dean
cisco@linux-dev:~/nxos-ansible$
```

2) Playbook

The playbook used for this particular use-case

```
cisco@linux-dev:~/nxos-ansible/ansible_playbooks$ more vlans.yml
# vlans.yml
---

- name: VLANs
  hosts: all
  connection: local
  gather_facts: no

  tasks:
    - name: ensure VLANs 2-20 and 99 exist on all switches
      nxos_vlan: vlan_id="2-20,99" state=present host={{ inventory_hostname }}

    - name: config VLANs names for a few VLANs
      nxos_vlan: vlan_id={{ item.vid }} name={{ item.name }} host={{ inventory_hostname }}
state=present
  with_items:
    - { vid: 2, name: web }
    - { vid: 3, name: app }
    - { vid: 4, name: db }
    - { vid: 20, name: server }
    - { vid: 99, name: native }
```


3) Checking VLAN

In this step we are verifying the VLANs are not configured in the switch

```
n9k-sw-1# sh vlan brief

VLAN Name                Status    Ports
-----
1    default                active    Eth1/1, Eth1/2, Eth1/3, Eth1/4
                                           Eth1/5, Eth1/6, Eth1/7, Eth1/8
                                           Eth1/9, Eth1/10, Eth1/11
                                           Eth1/12, Eth1/13, Eth1/14
                                           Eth1/15, Eth1/16, Eth1/17
                                           Eth1/18, Eth1/19, Eth1/20
                                           Eth1/21, Eth1/22, Eth1/23
                                           Eth1/24, Eth1/25, Eth1/26
                                           Eth1/27, Eth1/28, Eth1/29
                                           Eth1/30, Eth1/31, Eth1/32
                                           Eth1/33, Eth1/34, Eth1/35
                                           Eth1/36, Eth1/37, Eth1/38
                                           Eth1/39, Eth1/40, Eth1/41
                                           Eth1/42, Eth1/43, Eth1/44
                                           Eth1/45, Eth1/46, Eth1/47
                                           Eth1/48, Eth1/49, Eth1/50
                                           Eth1/51, Eth1/52, Eth1/53
                                           Eth1/54

n9k-sw-1#
```

4) Execute Playbook

Now run the playbook by executing `ansible-playbook vlans.yml`

```
cisco@linux-dev:~/nxos-ansible/ansible_playbooks$ ansible-playbook vlans.yml

PLAY [VLANs] *****

TASK: [ensure VLANs 2-20 and 99 exist on all switches] *****
changed: [dean]
```

```
TASK: [config VLANs names for a few VLANs] *****
changed: [dean] => (item={'name': 'web', 'vid': 2})
changed: [dean] => (item={'name': 'app', 'vid': 3})
changed: [dean] => (item={'name': 'db', 'vid': 4})
changed: [dean] => (item={'name': 'server', 'vid': 20})
changed: [dean] => (item={'name': 'native', 'vid': 99})

PLAY RECAP *****
dean                : ok=2    changed=2    unreachable=0    failed=0

cisco@linux-dev:~/nxos-ansible/ansible_playbooks$
```

5) Verify VLAN Creation

Check the switch to verify the VLANs have been created

```
n9k-sw-1# sh vlan brief

VLAN Name                Status      Ports
-----
1    default                active     Eth1/1, Eth1/2, Eth1/3, Eth1/4
                                           Eth1/5, Eth1/6, Eth1/7, Eth1/8
                                           Eth1/9, Eth1/10, Eth1/11
                                           Eth1/12, Eth1/13, Eth1/14
                                           Eth1/15, Eth1/16, Eth1/17
                                           Eth1/18, Eth1/19, Eth1/20
                                           Eth1/21, Eth1/22, Eth1/23
                                           Eth1/24, Eth1/25, Eth1/26
                                           Eth1/27, Eth1/28, Eth1/29
                                           Eth1/30, Eth1/31, Eth1/32
                                           Eth1/33, Eth1/34, Eth1/35
                                           Eth1/36, Eth1/37, Eth1/38
                                           Eth1/39, Eth1/40, Eth1/41
                                           Eth1/42, Eth1/43, Eth1/44
                                           Eth1/45, Eth1/46, Eth1/47
                                           Eth1/48, Eth1/49, Eth1/50
                                           Eth1/51, Eth1/52, Eth1/53
                                           Eth1/54
```

```
2    web          active
3    app          active
4    db           active
5    VLAN0005    active
6    VLAN0006    active
7    VLAN0007    active
8    VLAN0008    active
9    VLAN0009    active
10   VLAN0010    active
11   VLAN0011    active
12   VLAN0012    active
13   VLAN0013    active
14   VLAN0014    active
15   VLAN0015    active
16   VLAN0016    active
17   VLAN0017    active
18   VLAN0018    active
19   VLAN0019    active
20   server       active
99   native       active
```

Conclusion

Ansible provides an easy way to manage and automate the network with an agentless model. This example illustrates one way to leverage Ansible with Cisco Open NX-OS.

Workload On-Boarding

Problem Statement

As multi-tenant data centers - both public and private - attract an ever increasing number of customers, cloud operators are struggling to on-board tenants, applications, workloads, and users in a rapid and efficient fashion. Customers - many of whom have already benefited first-hand from automation tools - will no longer accept the delays that accompany traditional tenant on-boarding processes.

Solution

Network operators must begin to leverage automation tools to assist with workload on-boarding. One such automation tool is Puppet, which can be leveraged with Open NX-OS to enable:

- Intent-based configuration, dramatically decreasing configuration steps and deployment times
- The application of common tooling and skillsets across both server and network teams, decreasing operation costs
- Visibility into configuration change management history that aids in compliance monitoring

Although Puppet-based automation can be applied to many workload on-boarding problems, the example below focuses on a specific and particularly troublesome problem of access-port provisioning.

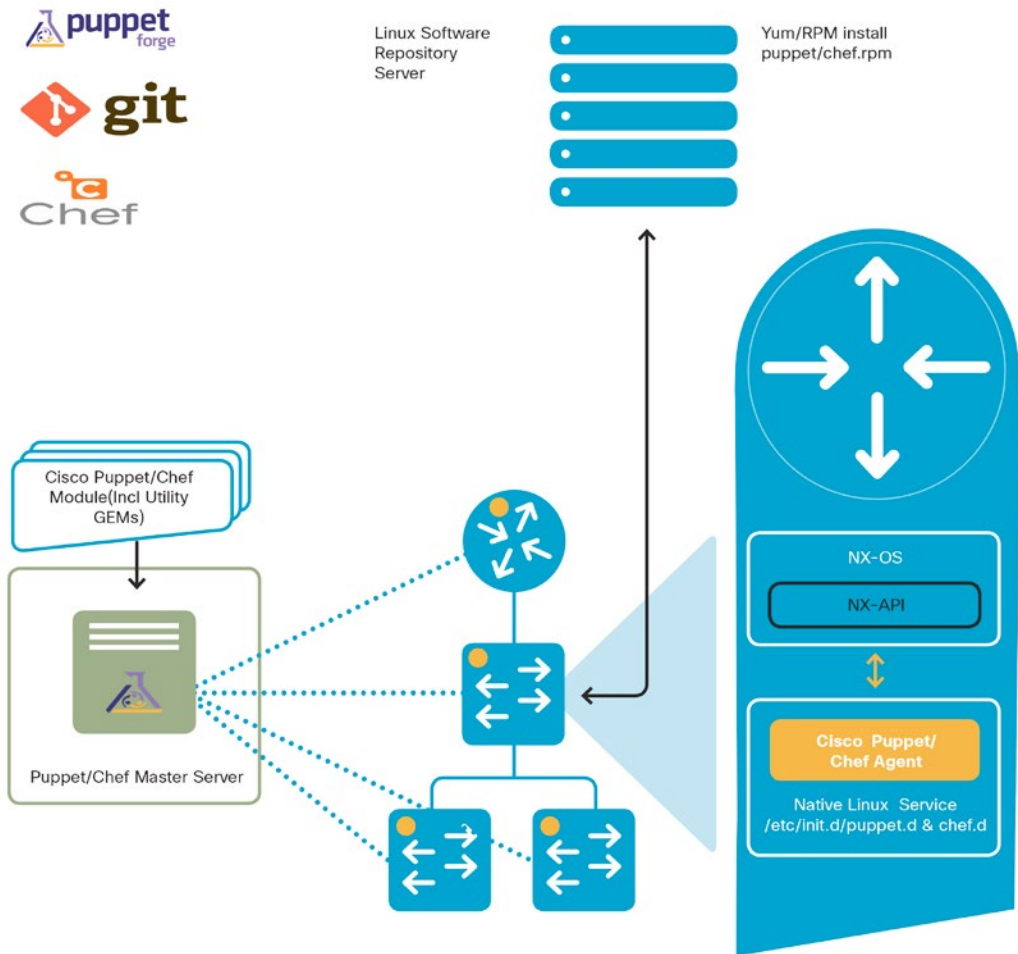
Historically, access-port provisioning has been a manual, labor-intensive, and error-prone process. With Puppet intent-based automation and Open NX-OS, processes that previously took hours or days can be completed in minutes, and with little potential for errors.

At the core of this use-case is the extensibility of open NX-OS Linux which allows for the integration of third-party software agents using RPMs. This use-case involves using a Puppet Agent to enable intent-based automation of the switching infrastructure.

The solution architecture (illustrated in Figure 1) consists of the following component pieces:

- A Puppet Master server
- A Cisco Puppet module
- A Puppet Agent / service which utilizes the NX-API CLI utility to configure the switch

Figure: Open NX-OS Agent-based Configuration Management (e.g., Puppet) Architecture



Puppet Domain-Specific Language

Puppet uses its own domain-specific language (DSL) to describe machine configurations and model resources. The declaration of resources is key to the Puppet DSL. When one resource depends on another, you should explicitly state the relationship to ensure that they are applied in the proper order.

The Puppet Resource Model Consists of two layers – Types (or Resources) and Providers. Types/Resources specify the interfaces used to describe resources in Puppet. Providers encapsulate the procedures used to manage resources on a specific platform.

Figure: Types and Providers supported out-of-the-box by the Cisco Puppet Module

Puppet Agent Types/Providers

cisco_vtp

cisco_tacacs_server

cisco_tacacs_server_host

cisco_snmp_server

cisco_snmp_community

cisco_snmp_group

cisco_ospf

cisco_ospf_vrf

cisco_vlan

cisco_bgp*

cisco_bgp_vrf*

cisco_interface

cisco_interface_ospf

cisco_interface_vlan

Puppet Manifests

The configurations for each node under management are written using the DSL language mentioned above. Code in this language is saved in files called *manifests*.

For complete information about the Puppet language see http://docs.puppetlabs.com/puppet/4.2/reference/lang_summary.html.

Manifests are files containing Puppet code on the puppet master server. They are standard text files saved with the `.pp` extension. Manifests should be arranged into modules that specify the configurations that should be applied to the node.

See http://docs.puppetlabs.com/pe/latest/puppet_modules_manifests.html#puppet-modules for more information on Puppet manifest modules.

Solution Approach

Puppet Master Setup

The Puppet Master must be setup in order to manage the configuration of the network switches. This is done once:

- 1 Download, install and configure the Puppet master software
- 2 Install the Cisco Puppet module directly from GitHub
- 3 Optionally, set up certificate auto-signing for Puppet agent nodes in your data center

Switch Native Puppet Agent Setup

Puppet Agents must be installed on each switch to enable communications with the Puppet Master. These steps are done once per switch:

- Ensure basic networking is configured, and communication is established between switch and Puppet master in the management namespace.

1) Enter Bash Shell

In NX-OS, Puppet Agent gets installed in the Bash Shell of the Cisco Nexus Switch

```
n9k-sw-1# run bash
bash-4.2$ whoami
admin
```

2) Change to the "management" namespace in the linux shell

This will result in changing the vrf to the management vrf.

```
bash-4.2# sudo ip netns exec management bash
bash-4.2# whoami
root
bash-4.2#
```

3) Add the DNS server to resolv.conf

```
bash-4.2# vi /etc/resolv.conf

nameserver <<DNS Server>>
```

4) Download puppet release agent rpm from yum.puppetlabs.com

Note: For the latest Puppet Agent info please refer to <https://github.com/cisco/cisco-network-puppet-module/blob/master/docs/README-agent-install.md#agent-config>

This step configures the yum repository on the switch for the agent, imports linux GPG encryption keys and copies the agent RPM to the switch.

```
bash-4.2# yum install http://yum.puppetlabs.com/puppetlabs-release-pcl-cisco-wrlinux-5.noarch.rpm
Loaded plugins: downloadonly, importpubkey, localrpmDB, patchaction, patching, protect-packages
groups-repo
| 1.1 kB    00:00 ...
localdb
| 951 B    00:00 ...
patching
| 951 B    00:00 ...
```



```

thirdparty
          | 951 B      00:00 ...
Setting up Install Process
puppetlabs-release-pcl-cisco-wrlinux-5.noarch.rpm
          | 5.8 kB      00:00
    Examining /var/tmp/yum-root-g33Fyq/puppetlabs-release-pcl-cisco-wrlinux-5.noarch.rpm:
puppetlabs-release-pcl-0.9.4-1.cisco_wrlinux5.noarch
    Marking /var/tmp/yum-root-g33Fyq/puppetlabs-release-pcl-cisco-wrlinux-5.noarch.rpm to be
installed
    Resolving Dependencies
--> Running transaction check
---> Package puppetlabs-release-pcl.noarch 0:0.9.4-1.cisco_wrlinux5 will be installed
--> Finished Dependency Resolution

Dependencies Resolved

=====
=====
Package                Arch          Version          Repository
Size
=====
=====
Installing:
  puppetlabs-release-pcl noarch          0.9.4-1.cisco_wrlinux5 /puppetlabs-pcl-cisco-lnx-
5.noarch 2.2 k

Transaction Summary

=====
=====
Install      1 Package

Total size: 2.2 k
Installed size: 2.2 k
Is this ok [y/N]: y
Downloading Packages:

```

```

Running Transaction Check
Running Transaction Test
Transaction Test Succeeded
Running Transaction
  Installing : puppetlabs-release-pc1-0.9.4-1.cisco_wrlinux5.noarch
                1/1

Installed:
  puppetlabs-release-pc1.noarch 0:0.9.4-1.cisco_wrlinux5

Complete!

```

5) Installing the Puppet Agent from the yum repository

```

bash-4.2# yum install puppet
Loaded plugins: downloadonly, importpubkey, localrpmDB, patchaction, patching, protect-
packages
groups-repo
                | 1.1 kB    00:00 ...
localdb
                |  951 B    00:00 ...
patching
                |  951 B    00:00 ...
thirdparty
                |  951 B    00:00 ...
puppetlabs-pc1
                | 2.5 kB    00:00
  puppetlabs-pc1/primary_db
                | 6.6 kB    00:00

  Setting up Install Process
Resolving Dependencies
--> Running transaction check
---> Package puppet-agent.x86_64 0:1.2.5-1.cisco_wrlinux5 will be installed
--> Finished Dependency Resolution

Dependencies Resolved

```

```

=====
=====
Package           Arch           Version           Repository           Size
=====
=====
Installing:
puppet-agent      x86_64         1.2.5-1.cisco_wrlinux5  puppetlabs-pc1      39 M

Transaction Summary

=====
=====
Install           1 Package

Total download size: 39 M
Installed size: 139 M
Is this ok [y/N]: y
Retrieving key from file:///etc/pki/rpm-gpg/RPM-GPG-KEY-puppetlabs
Downloading Packages:
puppet-agent-1.2.5-1.cisco_wrlinux5.x86_64.rpm
| 39 MB    00:11

Running Transaction Check
Running Transaction Test
Transaction Test Succeeded
Running Transaction

  Installing : puppet-agent-1.2.5-1.cisco_wrlinux5.x86_64
                                     1/1

Installed:
puppet-agent.x86_64 0:1.2.5-1.cisco_wrlinux5

Complete!
bash-4.2#

```

- Install `net_http_unix`, `cisco_nxapi`, and `cisco_nodeutil` gem modules either individually on the switch, or as part of the Puppet manifest for the switch.

- Download these packages as part of the Cisco agent software from <https://forge.puppetlabs.com/puppetlabs/ciscopuppet>

Additional information regarding installation instructions is available at <https://puppetlabs.com/solutions/cisco>

6) Edit the switch manifest on the Puppet Master to enable tenant on-boarding

For tenant on-boarding in data center environments, typical operations are creating VLAN, SVI, assigning ports to VLANs. An example manifest that on-boards a new Tenant A in VLAN 220 would appear as:

Provision Tenant A:

```
cisco_vlan { "220":
  ensure => present,
  vlan_name => 'TenantA',
  shutdown => 'true',
  state => 'active',
}
```

Provision VLAN Interface for Tenant A:

```
cisco_interface { "Vlan220" :
  svi_autostate => false,
}
```

Provision Tenant A Switch Port:

```
cisco_interface { "Ethernet1/2" :
  description => 'default',
  shutdown    => 'default',
  access_vlan => '220',
}
```

7) When applied to a switch the resulting configuration would be:

```
interface Ethernet1/2
  switchport access vlan 220

vlan 220
  name TenantA

interface Vlan220
  no shutdown
```

Conclusion

Enabling automation through the combination of NX-OS extensibility and Puppet results in a fundamental decrease in time-to-deployment for on-boarding tenants, improved SLA compliance, and decreased operational costs.

Infrastructure as Code

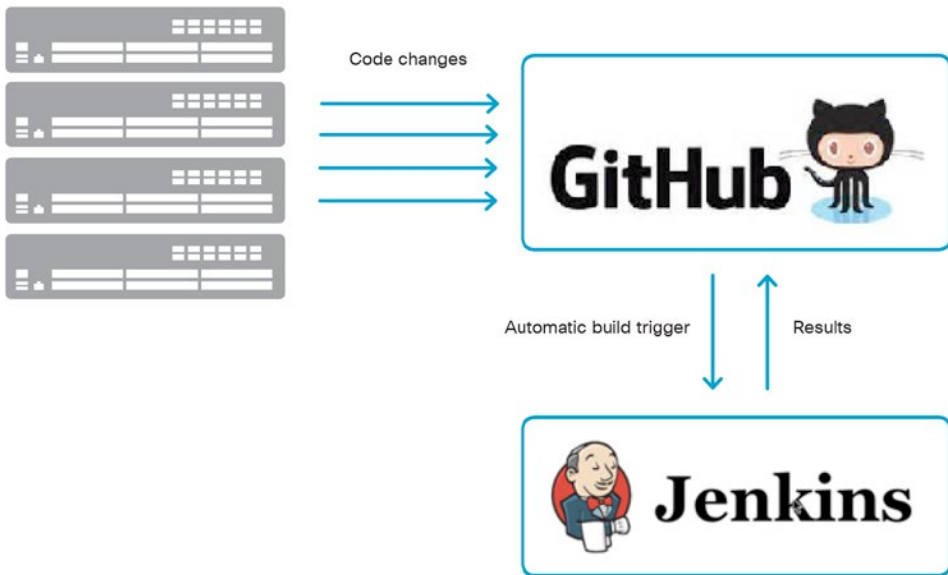
Problem Statement

Network operators have a strong interest in maintaining control over the configurations that have been applied to switches. Additionally, users need to store and track configuration changes within the networking infrastructure. Open source tools can be used to automatically push switch configuration changes to a central repository database system.

Solution

The DevOps movement has resulted in changes to many traditional processes and tools. One artifact of the movement is treatment of infrastructure data as code, which is different from previous models. This includes storing items such as device configurations in source control management systems. Some of the benefits of using a source control management system are audit tracking and the ability to roll back to any version that was previously checked-in. The source control system check-in could also trigger other parts of a change/test/release cycle, expanding events to tools such as Jenkins to automatically test and report on the effect the change itself had on the infrastructure.

The solution set for this problem can be applied to multiple network topologies. The devices need to have connectivity to a source control management system.

Figure: Open NX-OS Switches Integrate into Configuration Management Workflow

Solution Approach

This solution involves multiple elements.

1) Trigger Event

The trigger event involves the switch leveraging Embedded Event Manager (EEM) to monitor the CLI for changes. In this particular scenario, EEM is looking for match criteria "*copy running-config startup-config*." After EEM detects the command, it will invoke a Python script from guest shell called *autoconfig.py* which is stored in the local switch storage (bootflash)

Here is a sample EEM configuration:

```
event manager applet gitpush
  event cli match "copy running-config startup-config"
  action 2 cli copy running bootflash:/autoconfig/running.latest
```

```

action 3 cli guestshell run /home/guestshell/autoconfig.py
action 4 event-default

```

2) LXC Container / Guest Shell

The guest shell is used to host the Python Script and is where the *git* client is going to be running

```

n9k-sw-1# guestshell
[guestshell@guestshell ~]$

```

3) Installing Git client

The *git* client is available as open source package, and can be downloaded directly onto the device through Yum within the guest shell. The *git* client will allow changes to be pushed first into a local repository, and then pushed to a central configuration store. Y install can be used to grab the *git* client software:

```

guestshell::~$ sudo chvrf management yum install git

```

NOTE: In the above sample the management context (VRF - Virtual Routing and Forwarding) is used for connectivity.

4) Python script

A python script which is triggered directly from the Embedded Event Manager (EEM) will be used to save the configuration change, and use the *git* APIs to push the changes to the Central Repository.

The script is saved as *autoconfig.py* which matches the name that was triggered from the Embedded Event Manager (EMM) above. It will take the saved file and schedule it to be committed through the calls to *git*. Finally a *git* push is called to push the changes to an external repository. Notice above that vrf management is also used within the script for external connectivity.

The python script for this use case looks like the following:


```
#!/usr/bin/python

import os
import subprocess
from subprocess import call

f = open("autooutput.txt","w") #opens file with name of "test.txt"
os.chdir("/home/guestshell/autoconfig")

call(["mv", "/bootflash/autoconfig/running.latest", "/home/guestshell/autoconfig/n9k-sw-1/running"])
call(["git", "add", "n9k-sw-1/running"])
call(["git", "commit", "-m", "Configuration change"])
p = subprocess.Popen(['chvrf', 'management', 'git', 'push'], stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
out, err = p.communicate()
f.write(out)
f.write(err)
f.close
```

5) Git repository

A central *git* repository is needed to store the checked-in data. This could be an in-house *git* repository, or a cloud-based community repository such as GitHub.

6) Verification

When the end user enters a configuration save from the CLI, the central repository store will be updated.

Figure: Incremental Configuration on Open NX-OS Switch Archived on Git

The screenshot shows a Git Code Review interface for a repository named 'autoconfig.git'. The page title is 'Code Review / autoconfig.git / summary'. Below the title, there are navigation links: 'summary | shortlog | log | commit | commitdiff | review | tree'. A search bar is present with the text 'commit' and a search icon. The main content area displays the following information:

- description:** Push network element configs to git Repository
- owner:** codehub
- last change:** Mon, 1 Jun 2015 13:08:34 -0700 (20:08 +0000)
- URL:** http://codehub-one-samples-review.cisco.com:8090/p/autoconfig.git
ssh://jpleifer@codehub-one-samples-review.cisco.com:29418/autoconfig.git

Below this information is a 'shortlog' section showing a list of commits. Each entry includes the time since the commit, the author's name, the commit message, and links for 'commit', 'commitdiff', 'tree', and 'snapshot'. The first commit is highlighted with a green background and labeled 'master'. A large purple arrow points from the right towards the list of commits, with the text 'Source code repository (codehub)' inside it.

At the bottom of the page, there is a 'heads' section showing the current branch 'master' and a footer with the text 'Push network element configs to git Repository' and buttons for 'Atom' and 'RSS'.

Conclusion

As shown above, with this type of methodology leveraging open source tools and basic programming skills, a complete configuration store and revision mechanism can be created.

Troubleshooting with Linux Capabilities

Problem Statement

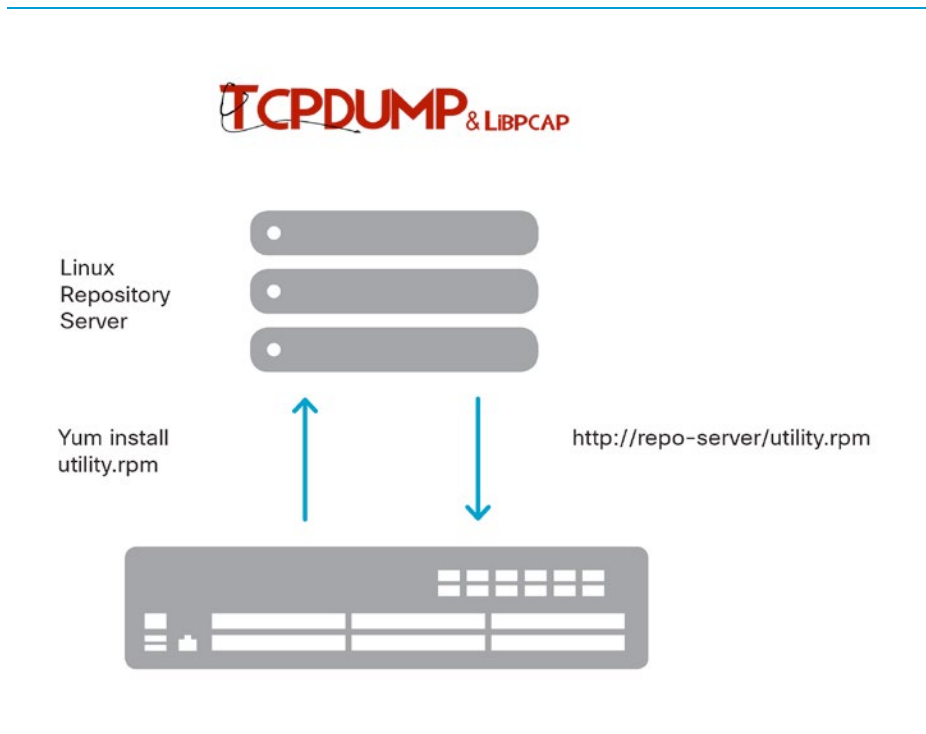
Network devices increasingly are running Linux as the base operating system. How do we leverage tools natively available in Linux to manage the switch and troubleshoot network problems?

Solution

Cisco Open NX-OS allows Linux shell access for users to access the underlying Linux file system on Cisco Nexus Series Switches, to collect information and troubleshoot issues using familiar Linux commands.

Solution Approach

In Open NX-OS, network interfaces are exposed as netdevices within Linux (EthX-X). Linux commands a network operator can use are **ifconfig**, **tcpdump**, **vsh** etc. to make it easier to manage the switch interfaces in the same manner as network ports on a Linux server.

Figure: Using Standard Package Management Infrastructure (e.g., yum) with Open NX-OS

For troubleshooting, use **tcpdump** to capture all packets on a given port, and dump output to a file:

```
bash-4.2$ sudo tcpdump -w file.pcap -i Eth1-1
tcpdump: WARNING: Eth1-1: no IPv4 address assigned
tcpdump: listening on Eth1-1, link-type EN10MB (Ethernet), capture size 65535 bytes

3 packets captured
3 packets received by filter
0 packets dropped by kernel
```

Use **ethtool** to display detailed interface statistics:

```
#ethtool -S eth1-1
NIC statistics:
    speed: 10000
    port_delay: 10
    port_bandwidth: 10000000
    admin_status: 1
    oper_status: 1
    port_mode: 0
    reset_counter: 20
    load-interval-1: 30
    rx_bit_rate1: 0
    rx_pkt_rate1: 0
    tx_bit_rate1: 272
    tx_pkt_rate1: 0
    load-interval-2: 300
    rx_bit_rate2: 0
    rx_pkt_rate2: 0
    tx_bit_rate2: 256
    tx_pkt_rate2: 0
    rx_unicast_pkts: 1340
    rx_multicast_pkts: 0
    rx_broadcast_pkts: 0
    rx_input_pkts: 1340
    rx_bytes_input_pkts: 1886720
```

Verify the MTU of an interface, and then use **ifconfig** to change mtu for an interface to jumbo MTU:

```
n9k-sw-1# run bash
bash-4.2#
bash-4.2# ifconfig Eth1-1
Eth1-1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet6 fe80::fac2:88ff:fe90:2cb2 prefixlen 64 scopeid 0x20<link>
    ether f8:c2:88:90:2c:b2 txqueuelen 100 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 2204374 bytes 170123906 (162.2 MiB)
```

```
TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0
```

The following example shows how to set the MTU for interface 1/1 to 9000

```
bash-4.2# sudo ifconfig Eth1-1 mtu 9000
bash-4.2#
bash-4.2# ifconfig Eth1-1
Eth1-1    Link encap:Ethernet  HWaddr f8:c2:88:90:2c:b2
          inet6 addr: fe80::fac2:88ff:fe90:2cb2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:9000  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2204856 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 B)  TX bytes:170161160 (162.2 MiB)
```

The last example depicts how customers can execute NX-OS commands from bash. This is beneficial because the user will be able to execute commands and leverage their existing Linux tools to manage the switch. In this particular example, we are querying the interfaces from the switch leveraging the `vsh` command from bash shell.

```
bash-4.2$ vsh -c "show interface brief" | grep up | awk '{print $1}'
mgmt0
Eth1/49
bash-4.2$
```

Conclusion

Cisco enables the capability to configure switch ports as Linux devices with standard Linux tools on Cisco Nexus Series switches. This will reduce and simplify the IT toolchain.

Network Monitoring with Splunk

Problem Statement

Fine granular visibility into network structure, performance and failures can help network operators manage the network more efficiently. Customers have had success using tools like Splunk to monitor compute infrastructures. Splunk can also be used in coordination with the network for capturing and graphing inventory, performance, congestion, latency and failure data.

Solution

Network performance monitoring capabilities can help an operator to proactively identify arising problems and resolve them. This use-case illustrates how a network operator can gather performance data, including response time, one-way latency, jitter (interpacket delay variance), packet loss, network resource availability, application performance, and server response time for various paths in the network. This capability can be extremely helpful in managing and troubleshooting a network.

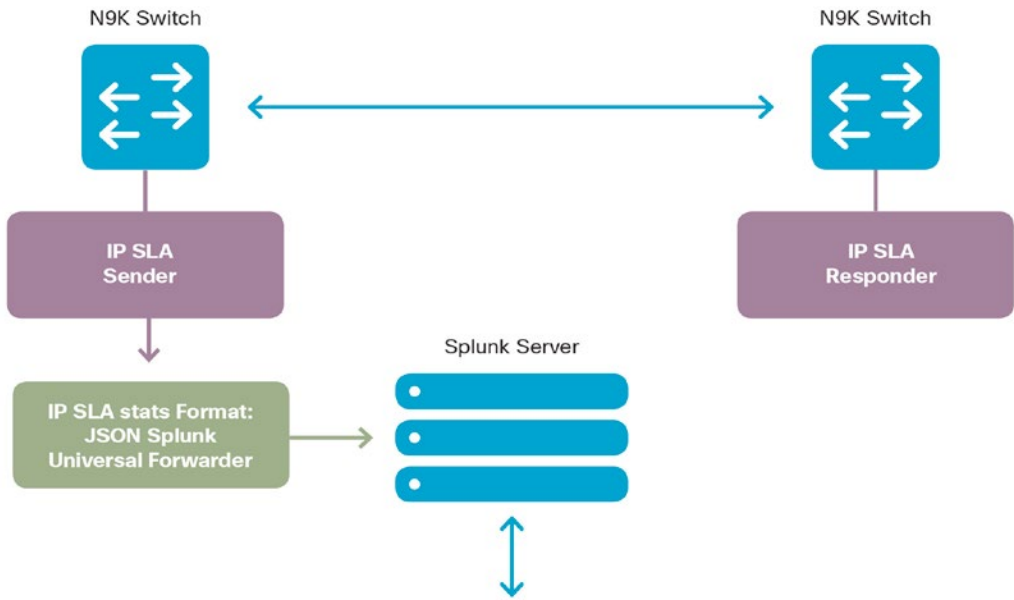
Splunk is a tool that can be used to gain operational visibility across the IT infrastructure. With Splunk you can search, report, monitor and analyze real-time streaming and historical IT data generated by all your IT systems from one place. Additionally, Splunk can troubleshoot application problems and investigate security incidents, reducing time to repair them.

IP SLA on Nexus device gathers rich network performance data by monitoring and capturing latency and jitter between designated network elements. An on-box collector is required to capture the performance metrics between the network components, and this performance data is aggregated and sent to a central location for further analysis.

Splunk forwarders on NX-OS switch (lightweight Splunk servers with indexing turned off) can be deployed in situations where the data needed isn't available over the network nor visible to the server where Splunk is installed. Splunk forwarders can monitor local application log files, capture the output of status commands on a schedule, grab performance metrics from virtual or non-virtual sources, or watch the file system for configuration, permissions and attribute

changes. Forwarders send data securely to the central Splunk server in real time. They are lightweight, can be deployed quickly and no additional cost is incurred.

Figure: Enhance Infrastructure Visibility with Splunk Forwarder on Open NX-OS



Splunk Web: <http://<splunkserver>:5000>

This use case utilizes numerous Open NX-OS capabilities:

- **Guest shell:** Guest shell provides a secure, isolated environment for users to install software that extends the switch functionality. We'll install the Splunk forwarder in the guest shell.
- **RPM capabilities:** Splunk forwarder running in a container or an RPM to send data to a centralized Splunk forwarder.
- **Python:** A python script running on the switch, utilizing native python capabilities, gathers information to be presented to Splunk.
- **NX-API CLI:** The python script interacts with NX-OS via NX-API CLI interface and gathers performance data.

Solution Approach

For the purpose of illustrating this particular use-case, we'll use two Nexus devices connected back to back.

1) Install Splunk forwarder as RPM on NX-OS

```
n9k-sw-1# guestshell
bash-4.2# yum install splunkforwarder-6.2.3-264376-linux-2.6-x86_64.rpm
```

2) Start Splunk

```
bash-4.2# splunk start --accept-license
```

3) Add forward server to push data to Splunk Enterprise

```
bash-4.2# splunk add forward-server <splunk enterprise>:9997
```

4) Enable forwarder

```
bash-4.2# splunk restart
```

5) Configure Splunk forwarder

Lastly, configure the Splunk forwarder to watch a monitor file for incoming data. When any data is written to the monitor file, the Splunk forwarder will pick that data up and send it to the Splunk server.

```
bash-4.2#splunk add monitor /bootflash/home/admin/monitor_file
```

The Splunk forwarder can be checked to see what files it is monitoring:

```
bash-4.2#splunk list monitor
```

Sample scripts to forward data to the Splunk enterprise. All scripts that will be run by Splunk Forwarder should be placed in **\$SPLUNK_HOME/bin/scripts** folder.

6) Create the interface-counter.py script

In addition to capturing IP SLA data, other data sources can be captured as well and sent to the Splunk Collector. As an example, the script below will gather interface counter data which is picked up by the Splunk Universal forwarder. Using the following script interfaces with NX-API CLI functionality to grab interface data through a JSON-RPC call, the resulting data is stored in the response dictionary.

The script used to push any statistics can be easily developed using NX-API CLI.

(nxapi URI)

Use the nxapi sandbox to cut-and-paste the python script with the desired 'show command' (In the sandbox select output format as JSON, and in the request pane select python)

(sample generated script from the sandbox for 'show interface counter' CLI)

```
#!/usr/bin/python

import os, json, sys, requests

Modify these please

url='http://<SWITCH_MGMT_IP>/ins'
```

```

switchuser='<USERNAME>'
switchpassword='<PASSWORD>'

myheaders={'content-type':'application/json'}
payload={
  "ins_api": {
    "version": "1.0",
    "type": "cli_show",
    "chunk": "0",
    "sid": "1",
    "input": "show interface ethernet1/13",
    "output_format": "json"
  }
}

response = requests.post(url,data=json.dumps(payload), headers=myheaders,auth=
(switchuser,switchpassword)).json()

#Add just this line.to push/forward json output to splunk server.
print json.dumps(response)

```

7) Create data-forwarder.sh script

This is a wrapper shell script that will execute specific python scripts.

```

#!/bin/bash
unset LD_LIBRARY_PATH
/opt/splunkforwarder/bin/scripts/interface-counter.py

```

Configure the inputs.conf file according to the sample shown below

```

script://$SPLUNK_HOME/bin/scripts/data-forwarder.sh
interval = 60
sourcetype = json

```

The script listed above can be modified to write the response dictionary data directly to the monitor_file which the Splunk universal forwarder will then pick up and send to the collector.

8) Executing the Splunk forwarder

For the IP SLA example, data specific to IP SLA can also be sent to the Forwarder. Here is an example where jitter data is gathered and sent to the forwarder.

Splunk search string and graph

```
Search string to plot the average max jitter time from IP SLA sender to responder every
minute:

host=lp1 sourcetype=json "udp-jitter" -> Filter the udp-jitter json formatted event from the
host
ins_api.outputs.output.body.TABLE_common.ROW_common.latest-return-code="1" -> Filter
successful event
earliest=-1m -> Filter events over the last one minute
| stats avg(ins_api.outputs.output.body.TABLE_jitter{}.ROW_jitter.sd-jitter-max) as
avg_max_jitter_time -> Calculate the average of max-jitter-time from source to destination
| gauge avg_max_jitter_time 20010 20025 20040 20055 -> Plot the value on a gauge
```

Conclusion

For this third-party monitoring application integration use-case, a Splunk forwarder was installed as an RPM. The forwarder can also be supported in a secure container. Splunk can be extended to include specific local queries of collected IP SLA data to be indexed for further visualization and analysis by the central server.

Network Monitoring with Open Source Tools

Problem Statement

Network Monitoring is a critical function in today's environment in order to:

- Optimize the network for performance and availability
- Keep track of network utilization
- Provide visibility for the current state of the network
- Provide certain types of IP SLA

There are many network monitoring software tools in the market today. These software network monitoring tools can be proprietary or created in an open source community.

Solution

In this particular use-case, we showcase the ways users can leverage an open source tool called *TCollector* to monitor the network. *TCollector* is an application running in the Cisco Nexus switch that will process data from local collectors and send the data to OpenTSB.

Solution Approach

1) Enter Bash Shell

In NX-OS, *TCollector* gets installed in the Bash Shell of the Cisco Nexus Switch.

```
n9k-sw-1# run bash
bash-4.2$ whoami
admin
```

2) Change the namespace interface in the shell to use "management"

This will move us into the management vrf and create a new shell.

```
bash-4.2# sudo ip netns exec management bash
bash-4.2# whoami
root
bash-4.2#
```

3) Add the DNS server to shell

```
bash-4.2# vi /etc/resolv.conf

nameserver <<DNS Server>>
```

4) Create a new yum repository configuration file

```
bash-4.2# pwd
/etc/yum/repos.d
bash-4.2#vi open-nxos.repo
[open-nxos]
name=open-nxos
baseurl=https://devhub.cisco.com/artifactory/open-nxos/7.0-3-I2-1/x86_64/
enabled=1
gpgcheck=0
metadata_expire=0
cost=500
sslverify=0
```

5) Install TCollector

The *TCollector* client is available as an open source package, and can be downloaded directly onto the device through yum.

```
root@n9k-sw-1#yum install tcollector
Loaded plugins: downloadonly, importpubkey, localrpmDB, patchaction, patching, protect-
packages
 groups-repo | 1.1 kB
00:00 ...
 localdb | 951 B
00:00 ...
```

```

patching | 951 B
00:00 ...
  thirdparty | 951 B
00:00 ...
  Setting up Install Process
  Resolving Dependencies
  --> Running transaction check
  --> Package tcollector.noarch 0:1.0.1-10 will be installed
  --> Finished Dependency Resolution

  Dependencies Resolved

=====
=====
  Package                Arch            Version          Repository
  Size
=====
=====
  Installing:
  tcollector             noarch         1.0.1-10        thirdparty
59 k
  Transaction Summary

=====
=====
  Install      1 Package

  Total download size: 59 k
  Installed size: 190 k
  Is this ok [y/N]: y
  Downloading Packages:
  Running Transaction Check
  Running Transaction Test
  Transaction Test Succeeded
  Running Transaction
  Warning: RPMDB altered outside of yum.

```



```

Installing : tcollector-1.0.1-10.noarch
1/1
Starting tcollector: [ OK ]
Installed:
    tcollector.noarch 0:1.0.1-10

Complete!

```

5) Verify TCollector is running

```

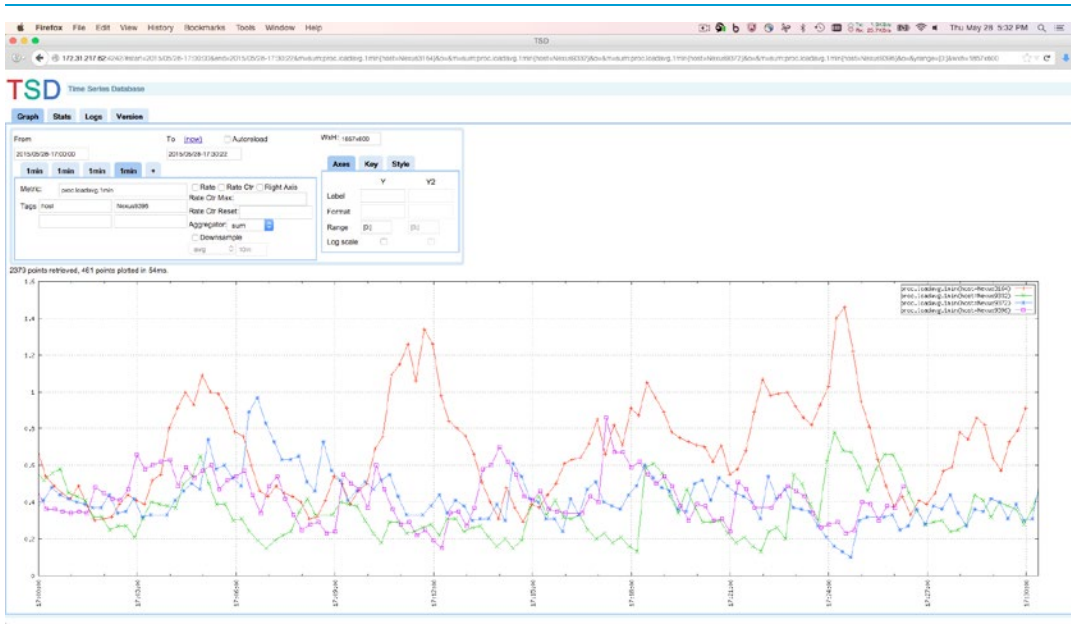
root@n9k-sw-1#ps -ef | grep tcoll
root      22788      1  0 22:35 ?          00:00:00 /usr/bin/python
/usr/local/tcollector/tcollector.py -D -H 10.6.54.60 -t host=n93k-2 -P /var/run/tcollector.pid
--reconnect-interval 0 --max-bytes 67108864 --backup-count 0 --logfile /var/log/tcollector.log
nobody    22830 22788   0 22:35 ?          00:00:00 /usr/bin/python
/usr/local/tcollector/collectors/0/netstat.py
nobody    22872 22788   0 22:35 ?          00:00:00 /usr/bin/python
/usr/local/tcollector/collectors/0/dfstat.py
root      22908 22788   0 22:35 ?          00:00:00 /usr/bin/python
/usr/local/tcollector/collectors/0/smart-stats.py
nobody    22942 22788   0 22:35 ?          00:00:00 /usr/bin/python
/usr/local/tcollector/collectors/0/iostat.py
nobody    22985 22788   0 22:35 ?          00:00:00 /usr/bin/python
/usr/local/tcollector/collectors/0/procstats.py
nobody    23036 22788   0 22:35 ?          00:00:00 /usr/bin/python
/usr/local/tcollector/collectors/0/ifstat.py
nobody    23040 22788   0 22:35 ?          00:00:00 /usr/bin/python
/usr/local/tcollector/collectors/0/procnettcp.py
root      23098 16343   0 22:35 pts/0    00:00:00 grep tcoll

```

6) TSDB Interface

Log into the TSDB interface to view the data.

Figure: Using TCollector Agent on Open NX-OS to Monitor Switch Performance with OpenTSDB



Conclusion

Open source software tools built for network operations, data gathering, statistical analysis and trending can provide simpler and more robust toolsets than the prevailing proprietary software product equivalents. Installing lightweight agents directly into a switch's shell provides an open interface and enables a straightforward, simple, and robust set of tools to monitor DC infrastructure.

Automating Network Auditing and Compliance

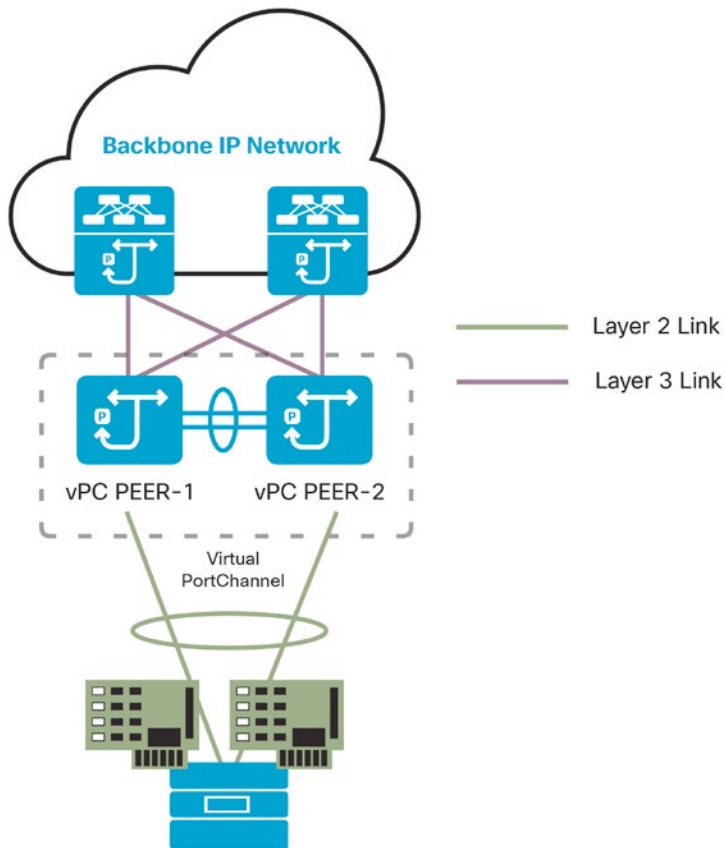
Problem Statement

There are numerous instances where periodic or on-demand audit checks on configurations can ensure consistency and security in the infrastructure. The following use case illustrates how a audit check for Virtual Port-Channel (vPC) configuration can be performed - the same methodology can be extended to check ACLs, QoS, and many other use cases.

vPC or Virtual Port Channel is a technology created by Cisco which allows physical links connected to two separate sets of switches to appear as a single port channel to the end devices. The end devices could be any network device such as servers, routers, firewalls, etc. The advantage of vPC is it provides redundancy to the devices while increasing bandwidth.

vPC configuration in some cases can be tedious and error-prone as:

- vPC requires several steps to configure
- vPC configuration steps must follow a specific order
- Certain vPC configuration elements must be identical on both switches

Figure: Sample Topology used by the vPC Consistency Handling Script

Operators can benefit from an automated mechanism that helps validate the consistency of vPC configurations.

Solution

This solution is going to illustrate a Python script that checks both vPC pair switches for any inconsistencies related to vPC. In this particular example we are only covering MTU mismatch, but it could be extended to any other value.

If the the script detects any configuration issues, it will identify the mis-match and attempt a recovery by applying the correct configuration value. We are able to accomplish this solution by using the Python script and NX-API CLI.

Solution Approach

vPC Consistency Check Script

The latest version of the script can be downloaded from <https://github.com/tecdct2941>

The following code snippet captures the various vPC configurations on which it is desirable to run consistency checks:

```
type_tbl = {
    'Interface type' : '1',
    'LACP Mode'      : '1',
    'STP Port Guard' : '1',
    'STP Port Type' : '1',
    'Speed'          : '1',
    'Duplex'         : '1',
    'MTU'            : '1',
    'Port Mode'      : '1',
    'STP MST Simulate PVST' : '1',
    'Native Vlan'    : '1',
    'Pvlan list'     : '2',
    'Admin port mode' : '1',
    'lag-id' : '1',
    'mode' : '1',
    'vPC card type' : '1',
    'Allowed VLANs' : '-',
    'Local error VLANs' : '-'
}
```

The following section uses NX-API CLI to gather the vPC operational state from a switch:

```

def check_vpc_status(switch_x):

    switch_ip = switch_x["mgmt_ip"]
        switchuser = switch_x["username"]
    switchpassword = switch_x["user_pw"]
    return_data = {}
    url = "http://" + switch_ip + "/ins"

    myheaders={'content-type':'application/json-rpc'}
    payload=[
        {
            "jsonrpc": "2.0",
            "method": "cli",
            "params": {
                "cmd": "show vpc brief",
                "version": 1.2
            },
            "id": 1
        }
    ]

    response = requests.post(url,data=json.dumps(payload), headers=myheaders,auth=
(switchuser,switchpassword)).json()

    resp_body = response['result']['body']
    return_data['vpc-peer-status'] = resp_body['vpc-peer-status']
    resp_vpc_table = resp_body['TABLE_vpc']['ROW_vpc']

    for iter in resp_vpc_table:

        one_vpc_id = iter['vpc-id']
        vpc_id.append(one_vpc_id)

        return_data[str(one_vpc_id)] = {}
        return_data[str(one_vpc_id)]['consistency-status'] = iter['vpc-consistency-
status']

        return_data[str(one_vpc_id)]['port-id'] = iter['vpc-ifindex']

    return return_data

```

The following section gathers vPC consistency state across the two switches:

```

def check_vpc_consistency(switch_x,switch_y, vpc_id):

    cmd = "show vpc consistency-parameters vpc " + str(vpc_id) + " errors"
    switch_ip = switch_x["mgmt_ip"]
        switchuser = switch_x["username"]
    switchpassword = switch_x["user_pw"]
    return_data = {}
    url = "http://" + switch_ip + "/ins"

    myheaders={'content-type':'application/json-rpc'}

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

    response = requests.post(url,data=json.dumps(payload), headers=myheaders,auth=
(switchuser,switchpassword)).json()

    resp_body = response['result']['body']['TABLE_vpc_consistency']['ROW_vpc_consistency']
fail_list = []

    for iter in resp_body:
        i_name = iter['vpc-param-name']
        i_type = iter['vpc-param-type']
        i_local = iter['vpc-param-local-val']
        i_peer = iter['vpc-param-peer-val']

        if i_local != i_peer:
            print "Found Inconsistency in " + i_name + ": local val: " + i_local +
" peer val : " + i_peer

```



```

        fail_list.append(iter)

return fail_list

```

The following section corrects any inconsistencies identified:

```

def correct_vpc_consistency(switch_x, switch_y, fail_list, vpc_id, port_id):

    print "Correcting VPC " + str(vpc_id) + "\n"
    switch_ip = ""
    switchuser = ""
    switchpassword = ""

    for iter in fail_list:
        if iter['vpc-param-name'] == "MTU":
            print "Correcting MTU"
            higher_mtu = ""
            if iter['vpc-param-local-val'] < iter['vpc-param-peer-val']:
                higher_mtu = iter["vpc-param-peer-val"]
                switch_ip = switch_x["mgmt_ip"]
                switchuser = switch_x["username"]
                switchpassword = switch_x["user_pw"]
            else:
                higher_mtu = iter["vpc-param-local-val"]
                switch_ip = switch_y["mgmt_ip"]
                switchuser = switch_y["username"]
                switchpassword = switch_y["user_pw"]

    payload={
        {
            "jsonrpc": "2.0",
            "method": "cli",
            "params": {
                "cmd": "conf t",
                "version": 1.2
            },
        },

```

```

        "id": 1
    },
    {
        "jsonrpc": "2.0",
        "method": "cli",
        "params": {
            "cmd": "interface " + port_id,
            "version": 1.2
        }
    },
    "id": 2
},
{
    "jsonrpc": "2.0",
    "method": "cli",
    "params": {
        "cmd": "mtu " + str(higher_mtu),
        "version": 1.2
    }
},
"id": 3
}
]

url = "http://" + switch_ip + "/ins"
myheaders={'content-type':'application/json-rpc'}
response = requests.post(url,data=json.dumps(payload), headers=myheaders,auth=
(switchuser,switchpassword)).json()

```

Finally, the control logic for the entire script:

```

def main():

    print "**** Calling vlan consistency checker ****"
    consistent1 = check_vpc_status(switch_a)

    print "Checking VPC status. Results:\n "
    is_consistent = 1

```

```

        for one_vpc_id in vpc_id:
            if consistent1[str(one_vpc_id)]['consistency-status'] == "INVALID":
                is_consistent = 0
                print "VPC " + str(one_vpc_id) + " is inconsistent - checking
reason...\n"

                cons_check = check_vpc_consistency(switch_a, switch_b, one_vpc_id)
                correct_it = correct_vpc_consistency(switch_a, switch_b, cons_check,
one_vpc_id, consistent1[str(one_vpc_id)]['port-id'] )
                print "VPC" + str(one_vpc_id) + " corrected"

            time.sleep(5)

        if is_consistent == 0:
            print "Rechecking VPC Status after correction. Results: \n"
            consistent_recheck = check_vpc_status(switch_a)

            for one_vpc_id in vpc_id:
                if consistent_recheck[str(one_vpc_id)]['consistency-status'] ==
"INVALID":
                    print "VPC " + str(one_vpc_id) + " is still inconsistent"

            else:
                print "No Inconsistency found !! \n"

```

Script Output

1) Here is a sample output where inconsistency was not found

```

admin@linux:python vcp_check_py
**** Calling vlan consistency checker ****
Checking VPC status. Results:

No Inconsistency found !!

*** Vlan consistency checker complete ***

```

2) Here is a sample output where inconsistency was found

```
admin@linux:python vcp_check_py
**** Calling vlan consistency checker ****
Checking VPC status. Results:

VPC 200 is inconsistent - checking reason...

Found Inconsistency in MTU: local val: 1500 peer val : 9216
Correcting VPC 200

Correcting MTU
VPC200 corrected
Rechecking VPC Status after correction. Results:

**** Vlan consistency checker complete ****
```

Conclusion

Using the power of Python scripting combined with the NX-API CLI, it is possible to automate many time-consuming and error-prone tasks. In this use-case, we showed how automation can help operators minimize configuration errors that could potentially lead to service interruptions, and eliminate time it might take to troubleshoot vPC problems manually.

Automated Network Topology Verification

Problem Statement

Network administrators are constantly striving to capture a real time view of their network topology or cable-plan so it can be compared against the intended topology. They can benefit from the use of programmatic tools capable of dynamically generating maps of a live topology and comparing it to archived versions of the designated cable-plan.

Solution

Leveraging an API-driven approach to dynamically generate a live cable-plan for a network allows operators to work with the resulting data programmatically. Having captured the live cable-plan they can utilize it as follows:

- Compare it to previous versions to identify changes
- Analyze the topology for troubleshooting and failure analysis
- Track the evolution of the topology
- Archive the cable-plan for future comparisons

nxtoolkit

The NX Toolkit is a set of Python libraries that allow for basic configuration of the Cisco Nexus Switch. It is intended to allow users to quickly begin using the NX-API REST interface and decrease the learning curve necessary to begin using the switch.

`nxtoolkit` is available as an open source project (Apache License, Version2.0) on Github <http://github.com/datacenter/nxtoolkit>

Cable-Plan Application for `nxtoolkit`

The cable-plan module allows the programmer to easily import existing cable-plans from XML files, import the currently running cable-plan from a Cisco Nexus Switch, export previously imported cable-plans to a file, and compare cable-plans.

More advanced users can use the Cable-Plan application to easily build a cable-plan XML file, query a cable-plan, and modify a cable-plan.

The Cable-Plan application is available as an application under the nxtoolkit open source project ([nxtoolkit/applications/cableplan](https://github.com/nxttoolkit/applications/cableplan)).

Cable-Plan XML Syntax

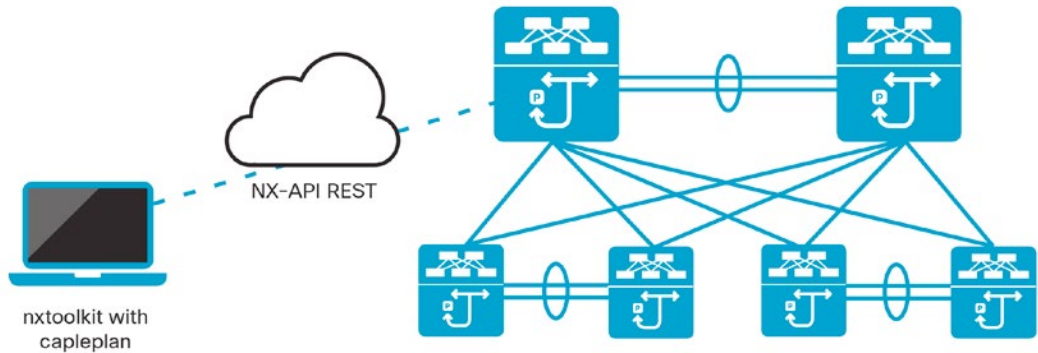
The cable-plan XML appears as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<?created by cable.py?>
<CISCO_NETWORK_TYPES version="None" xmlns="http://www.cisco.com/cableplan/Schema2"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="nxos-cable-plan-
schema.xsd">
  <DATA_CENTER networkLocation="None" idFormat="hostname">
    <CHASSIS_INFO sourceChassis="spine1" type="n9k">
      <LINK_INFO sourcePort="eth2/35" destChassis="leaf1" destPort="eth1/50"/>
      <LINK_INFO sourcePort="eth2/3" destChassis="leaf3" destPort="eth1/50"/>
      <LINK_INFO sourcePort="eth2/2" destChassis="leaf2" destPort="eth1/50"/>
    </CHASSIS_INFO>
    <CHASSIS_INFO sourceChassis="spine2" type="n9k">
      <LINK_INFO sourcePort="eth2/1" destChassis="leaf1" destPort="eth1/49"/>
      <LINK_INFO sourcePort="eth2/3" destChassis="leaf3" destPort="eth1/49"/>
      <LINK_INFO sourcePort="eth2/2" destChassis="leaf2" destPort="eth1/49"/>
    </CHASSIS_INFO>
  </DATA_CENTER>
</CISCO_NETWORK_TYPES>
```

The CHASSIS_INFO tag normally identifies the spine switches, then the leaf switches are contained in the LINK_INFO. When the XML is read, both leaf and spine switch objects will be created and the `get_switch()` and `get_link()` methods can be used to access them.

Solution Approach

Figure 1. Using `nxtoolkit` to Generate a Cable Plan from a Running Topology



Getting Started with Cable-Plan

The Cable-Plan application module is imported from `:file:'cableplan.py'` which can be found in the `nxtoolkit/applications/cableplan` directory.

It can be incorporated directly into a Python script, or it can be used from the command-line.

```
from cableplan import CABLEPLAN
```

To create a cable-plan from the current running Nexus switch, simply do the following:

```
cp = CABLEPLAN.get(session)
```

where `session` is a Nexus switch session object generated using the `nxtoolkit`, `cp` will be the cable-plan object.

Export that cable-plan by opening a file and calling the `export()` method as follows:

```
cpFile = open('cableplan1.xml','w')
cp.export(cpFile)
cpFile.close()
```

The cable-plan will be written to the `:file:`cableplan1.xml`` file.

Working with Saved Cable-Plans

Reading an existing cable-plan xml file is equally easy:

```
fileName = 'cableplan2.xml'
cp2 = CABLEPLAN.get(fileName)
```

Note that you don't have to explicitly open or close the file. The `get(fileName)` method will handle this.

Comparing Cable-Plans

Comparing cable-plans is one of the more interesting capabilities of the cable-plan module and is very easy to do using "difference" methods. When generating the difference between two cable-plans, the module will return those items that exist in the first cable-plan, but not in the second.

Missing Switches

For example, assume that in the above example, the second cable-plan read from the `:file:`cableplan2.xml`` file does not have switch "Spine3" and the first cable-plan does have it. The following example will print all of the switches in the first cable-plan and not in the second:

```
missing_switches = cp1.difference_switch(cp2)
for switch in missing_switches :
    print switch.get_name()
```

This should print the following output:

```
Spine3
```

Missing Links

Similarly, the following example will print all of the missing links:

```
missing_links = cp1.difference_link(cp2)
for link in missing_links :
    print link.get_name()
```

New and Missing Links

To understand all of the differences between two cable-plans it is necessary to compare them in both directions:

```
missing_links = cp1.difference_link(cp2)
extra_links = cp2.difference_link(cp1)
print 'The following links are missing from the second cable-plan'
for link in missing_links :
    print link.get_name()
print 'The following links are extra links in the second cable-plan'
for link in extra_links:
    print link.get_name()
```

Cable-Plan from the Command Line

Invoking the Cable-Plan application from the command line is very simple.

From the command prompt do the following:

```
python cableplan.py -h
```

This will return usage instructions that explain each of the command line options.

There are two primary functions that can be invoked from the command-line: 'export' and 'compare'.

The 'export' function, selected with the '-e' option, will create a cable-plan by reading the state of the Nexus switch. This cable-plan will be either displayed on the monitor or, if a filename is specified, will be placed in a file. It will be formatted in XML.

The 'compare' function will compare two cable-plans. One of those must be in a file that is specified at the command line and the second one can come either directly from the switch or from a second file. If only the '-c1 file_name' option is used, then the content of file_name is compared to the actual running configuration of the switch:

```
python cableplan.py -c1 netwrk1.xml
```

To compare two files, then both the '-c1 file_name1' and '-c2 file_name2' options must be used:

```
python cableplan.py -c1 netwrk1.xml -c2 netwrk2.xml
```

This comparison will list all of the links in the first cable-plan that are not in the second, and vice-versa.

Conclusion

Using the NX-API REST interface that is part of Open NX-OS and open source tools like *nx-toolkit*, operators can very easily build tools and scripts to help understand the current state of their infrastructure.

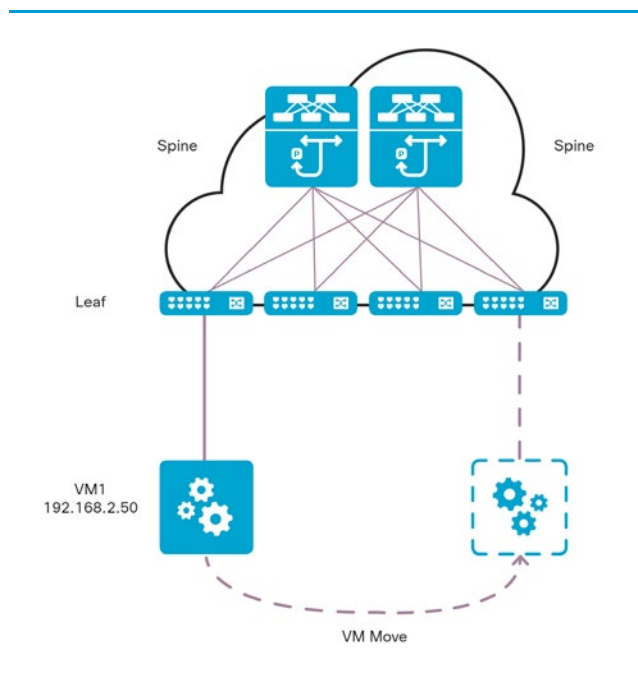
Workload Mobility and Correlation

Problem Statement

Virtualization has become a cornerstone technology in data centers today and is used extensively to enable faster provisioning and improve resource utilization rates. These virtualized servers are frequently moved between compute resources to satisfy demands for performance and service-scale.

When operators have a need to track the location of VMs or groups of VMs within a network, this workload mobility can present significant challenges. For example, in the diagram below, a workload has been moved but the network operations team is unaware of it. An automated means of tracking these changes is needed.

Figure: Tracking VMs in the Infrastructure with Open NX-OS Tools



Solution

The solution incorporates a number of Cisco and third-party / Open Source capabilities, including:

- Extensible Messaging and Presence Protocol (XMPP) and Python XMPP modules to facilitate group communications with NX-OS switches
- Cisco VM Tracker to access virtual machine name information on each switch
- A Python script that manages the communication with all of the target switches
- An XMPP/Jabber room where VM locations are reported

All communications are secured using SSL/TLS, as is typical with XMPP. Rule-Base Access Control (RBAC) is used to map the authenticated user executing the command via Python/XMPP to the switch command-line interface. RBAC is enforced in a delegated manner on the Switch itself.

Cisco NX-OS provides a extensive set of tools and components in order to achieve virtual machine visibility in accordance with VM Tracker for VMM - Virtual Machine Manager (vCenter) integration or VDP (part of lldpad) for the Open Virtual Switch (OVS). With this, we are able to associate the name of a virtual machine (VM) with the Cisco Nexus switch to which it is connected.

Solution Approach

1) Enabling VM Tracker feature

```
n9k-sw-1# conf t
Enter configuration commands, one per line. End with CNTL/Z.
n9k-sw-1(config)# feature vmtracker
n9k-sw-1(config)#
```

2) Establishing a Connection to vCenter

```
n9k-sw-1# configure terminal
n9k-sw-1(config)# vmtracker connection conn1
n9k-sw-1(config-vmt-conn)# remote ip address 10.1.1.1 port 80 vrf management
n9k-sw-1(config-vmt-conn)# username user1 password cisco123
```

```
n9k-sw-1(config-vmt-conn)# connect
```

3) Verification

Once the connection and the information between switch and vCenter has been exchanged you should be able to see the Virtual Machines.

```
n9k-sw-1# show vmtracker info detail
```

```
-----
Interface      Host           VMNIC   VM    State PortGroup  VLAN-Range
-----
Ethernet1/3    192.168.2.50  vmic4   VM1   on   PGroup100  100
-----
```

4) Using XMPP for VM visibility (VM Tracker)

With XMPP we can create groups of switches and then use group-chat execute commands on all of the switches simultaneously. This allows us to send - from a single location - a VM Tracking command to the group, and have each switch return the list of VMs that are connected.

5) Install an XMPP Server

```
admin@linux# yum install jabberd
```

6) Join a Group

Here is an example on how to attach a switch to the group chat "xmpp":

```
guestshell# jabberd attach group xmpp
guestshell>xmpp#
```

7) Executing a CLI Command

Here is a example of sending a single CLI command to multiple switches leveraging XMPP group chat:

```

guestshell>xmpp# show clock

<n9k-sw-1@xmpp.cisco.com/ (fmgr-device) (ABC1234abcd)>
18:46:37.481 UTC Wed Sep 30 2015
</n9k-sw-1@xmpp.cisco.com/ (fmgr-device) (ABC1234abcd)>

<n9k-sw-2@xmpp.cisco.com/ (fmgr-device) (DEF5678abcd)>
18:46:56.420 UTC Wed Sep 30 2015
</n9k-sw-2@xmpp.cisco.com/ (fmgr-device) (DEF5678abcd)>
Responses expected:2 received:2, time spend:117 msec

```

Below is another sample to showcase the power of VM Tracker with XMPP to gather information across multiple switches based on VM Name, IP address, connected interface, and so on:

```

guestshell>xmpp#
guestshell>xmpp# show vmtracker info detail

<n9k-sw-1@xmpp.cisco.com/ (fmgr-device) (ABC1234abcd)>
-----
Interface      Host           VMNIC  VM    State PortGroup  VLAN-Range
-----
Ethernet1/3    192.168.2.50  vmnic4 VM1   on   PGroup100  100
-----

<n9k-sw-2@xmpp.cisco.com/ (fmgr-device) (DEF5678abcd)>
-----
Interface      Host           VMNIC  VM    State PortGroup  VLAN-Range
-----
-----

```

8) Leveraging Python

Programmatic access using XMPP will allow for the execution of scheduled commands, the capture of structured responses (XML), and the storage of the data to a repository.

In the example below a Python script connects to XMPP and executes a show command against "VM tracker" with a respective VM name. The result is the location of the switch where the virtual machine (VM) is present.

```
import xmpp, re

vm="VM1"
cmd="show vmtracker info detail | include " + vm
jid='python@xmpp.cisco.com'
pwd="P$$$w0rd"
room="server@xmpp.cisco.com"

def messageHandler(conn, msg):
    if msg.getType() == "groupchat":
        result=re.findall("Interface [1-9]", str(msg.getBody()))
        if result:
            sw=re.findall("/(.*)/",str(msg.getFrom()))
            print vm + " was found on:\n"
            print sw[0] + "\n"
            print str(msg.getBody()) + "\n"

def StepOn(conn):
    try:
        conn.Process(1)
    except KeyboardInterrupt:
        return 0
    return 1

def GoOn(conn):
    while StepOn(conn):
        pass

jid=xmpp.protocol.JID(jid)
cl=xmpp.Client(jid.getDomain(), debug=[])

if cl.connect() == "":
    print "connection failed"
    sys.exit(0)
```



```

if cl.auth(jid.getNode(),pwd) == None:
    print "authentication failed"
    sys.exit(0)

cl.RegisterHandler('message',messageHandler)
cl.sendInitPresence()
cl.send(xmpp.Presence(to='{0}/{1}'.format(room, jid)))
message = xmpp.Message(room, cmd)
message.setAttr('type', 'groupchat')
cl.send(message)

GoOn(cl)

```

Below is the output of the Python script, which can be stored for correlation:

```

VM1 was found on: n9k-sw-1@xmpp.cisco.com

-----
Interface      Host           VMNIC   VM    State PortGroup  VLAN-Range
-----
Ethernet1/3    192.168.2.50  vmnic4  VM1   on   PGroup100  100
-----

Results returned :: 0

```

Conclusion

Virtual Machine visibility - real time or historical - can be achieved using Python scripts with a structured message bus (XMPP) and capturing the results. This information is automatically time stamped as per the XMPP message bus. Extracting this information and storing it in a repository can simplify the tracking of VMs and network troubleshooting in a virtualized environment.

Network Resiliency

Problem Statement

In some cases, users may need to manipulate the way network elements come online. This could be necessary to ensure servers are able to successfully come online when they are getting built.

Solution

In this particular use-case, requirements dictate host ports are kept in a shutdown state until uplink ports are active. This is necessary to ensure the ESX hosts are built properly.

Solution Approach

The following tools are leveraged to achieve the stated requirements:

- Embedded Event Manager (EEM)
 - To track the port up/down events
- Python
 - To execute a script based on an EEM argument to take the host port up or down.

The first stage is to create the EEM script to track the uplink port states and pass the argument based on state UP or DOWN:

```
event manager applet link_reboot
  event syslog pattern "Module 1 current-status is"
  action 1 syslog priority critical msg Running linkchange6 down
  action 2 cli source linkchange6.py down
event manager applet link_up
  event track 30 state up
  action 1 syslog priority critical msg Running linkchange6 up
  action 2 cli source linkchange6.py up
!
track 1 interface Ethernet2/1 line-protocol
```

```
track 2 interface Ethernet2/2 line-protocol
track 3 interface Ethernet2/3 line-protocol
track 4 interface Ethernet2/4 line-protocol
track 5 interface Ethernet2/1/1 line-protocol
track 6 interface Ethernet2/1/2 line-protocol
track 7 interface Ethernet2/1/3 line-protocol
track 8 interface Ethernet2/1/4 line-protocol
track 9 interface Ethernet2/2/1 line-protocol
track 10 interface Ethernet2/2/2 line-protocol
track 11 interface Ethernet2/2/3 line-protocol
track 12 interface Ethernet2/2/4 line-protocol
track 13 interface Ethernet2/3/1 line-protocol
track 14 interface Ethernet2/3/2 line-protocol
track 15 interface Ethernet2/3/3 line-protocol
track 16 interface Ethernet2/3/4 line-protocol
track 17 interface Ethernet2/4/1 line-protocol
track 18 interface Ethernet2/4/2 line-protocol
track 19 interface Ethernet2/4/3 line-protocol
track 20 interface Ethernet2/4/4 line-protocol
track 30 list boolean or
    object 1
    object 2
    object 3
    object 4
    object 5
    object 6
    object 7
    object 8
    object 9
    object 10
    object 11
    object 12
    object 13
    object 14
    object 15
    object 16
    object 17
    object 18
```

```
object 19
```

```
object 20
```

The second phase is to create the Python script that takes the argument from the EEM script and does the following:

- ARGV = Down
 - Check the host ports and shut them down and place on a flat file
- ARGV = UP
 - Read flat file and bring the host ports back online

```
#!/usr/bin/python

import sys
import os
import re
from cisco import cli

hostlistfile = "hostlist"
print cli('pwd')

try:
    sys.argv[1]
except IndexError:
    print "Error: Missing argument, need either 'up' or 'down'"
    exit()

if sys.argv[1] == "up":
    # At this point we should have a file containing ports to bring up.
    dirresult = cli("dir hostlist")
    if dirresult == "No such file or directory" :
        print "No hostlist found, exiting"
        exit()
    file = cli("show file hostlist")
    for match in file[0:-1].split('\n'):
        if match[0:8] == "Ethernet":
            cli("config terminal ; interface %s ; no shutdown" % match)
    cli ("delete hostlist no-prompt")
```

```

exit()

if sys.argv[1] == "down":
    # Uplinks are not yet up, lets see which hosts are active, bring those
    # down and save to a file
    print "Generating host list dynamically."
    result = cli("show interface")
    for rline in result[0:-1].split('\n'):
        match = re.match(r'^Ethernet[\S]+.*', rline)
        if match:
            #Check name against Ethenert2/ we will skip these uplinks.
            match2 = re.match(r'Ethernet2/.*', match.group(1))
            if match2:
                continue

            # If we made it this far, check state then adjust
            match3 = re.match(r'Ethernet1/.*', match.group(1))
            if match3:
                match4 = re.match(r'.*(Administratively down|SFP not inserted).*', rline)
                if match4 == None:
                    print match.group(1)
                    cli("echo '%s '>> hostlist" %match.group(1))
                    #Change state
                    cli("config t ; interface %s ; shutdown" % match.group(1))

```

Conclusion

Embedded Event Manager (EEM) can be used to trigger Python scripts to solve network connectivity issues that are affected by timing.

Programmability Tools for Network Engineers

Introduction

This chapter explores essential tools for network programming such as languages and environments, development and testing tools, source code and version control. An introduction to Cisco DevNet for Open NX-OS is provided for readers to explore the capabilities of Open NX-OS. Resources for learning network programming languages and Open NX-OS concepts are also provided.

For network programming novices, some good practices to follow are also outlined, which pertain to use of programming tools and development environments, storing and sharing your code, and integrating both of these processes.

Languages and Environments

Choosing a language in which to develop code is the first decision when building an application or program, followed by the platform for which the code will be written.

Programming languages are governed by rules and regulations (syntax) for constructing programs that pass instructions to a processor. Open NX-OS provides the ability to build native Linux applications using the Open NX-OS SDK, as well as utilizing NX-API REST for extensible and flexible programming options.

Python

Python is a heavily used language for network programming; it is a modular, extensible and flexible language with a focus on code readability. Code written in Python is executed through an interpreter, allowing portability of code between different platforms. Portability is a key factor, particularly for administrators and developers who may work in Windows, Linux or OS X. Python is growing in popularity and industry adoption. Other Python features include:

- High level and readable syntax
- Support for object-oriented constructs
- Ability to express concepts in fewer lines of code

Many of Cisco's network programmability projects available at <https://github.com/datacenter> are written using Python.

Other Languages

Python is certainly the most widely adopted language, although it is not the only language utilized for network programming. Metaphorically speaking, Python applications may be a single spoke in the larger wheel of network management and associated applications. Network programmers developing in Python may also use Python frameworks such as Django or Flask for rapid web development. Formatting and tagging with HTML and CSS, or integration with existing web-based applications using PHP may be desired. Programming language selection and usage should always tie directly back to core requirements and constraints.

Integrated Development Environments (IDE)

While all program code begins as text in a window, thankfully there are software packages available for download and use that can help simplify the development process. These packages are called *Integrated Development Environments* (IDEs) and integrate many of the common tasks in software development to enable rapid application development. While most beginners and novice programmers will utilize a text editor for their first projects, soon these users will leverage an Integrated Development Environment. The IDE is not code- or domain-specific, and in many cases the IDE is not language-specific, either.

An IDE can incorporate extensions, such as those discussed later with regard to version control, to allow programmers to integrate other aspects of application development into their workflow. For example, *git* extensions available within an IDE would allow check-in/check-out of code, as well as editing of changelog information during commits.

PyCharm is an IDE with full support for Python, as well as for HTML, CSS, JavaScript, Node.js and more.

<https://www.jetbrains.com/pycharm/>

Komodo is another IDE with full support for Python, as well as Go, Perl, Ruby and more.

<http://komodoide.com/>

Software Development Kits

A software development kit (SDK) differs fundamentally from an IDE. An SDK is a set of functions, packages, documentation and programs required for development on a specific platform or technology, whereas an IDE can be thought of as providing tools for application development using a specific language or technology. Java is one of the most widely-used languages, and it requires an SDK in order to develop Java applications. It is also important to outline the fact that *running* these applications does not require a full SDK, only a runtime environment. Software development kits are typically quite dependent on the target platform. For example, developing iOS applications for iPhone requires the Swift language, contained within the iOS SDK. Developers can then use the Xcode IDE for developing iOS applications.

Python does not have an SDK *per se*; its modularity supports using commands like *pip* to download and install packages and modules for use in Python programs.

Programming languages, IDEs and SDKs are essential components for software development. The user has multiple options related to language and IDE of choice. What is critical, is that APIs and data models are made available for network elements to be programmed.

Development and Testing Tools

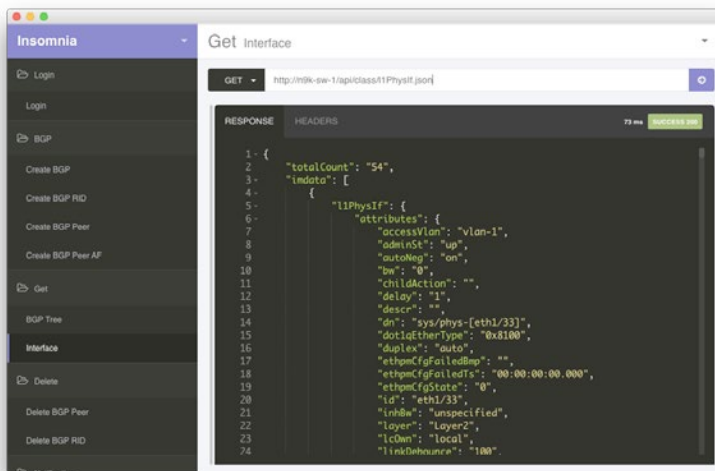
Just as network administrators have tools at their disposal for troubleshooting, debugging, and analysis, a programmer has their own set of tools. The tools listed in this section can aid in development, testing and debugging of code.

Postman

Postman can be utilized to build and test APIs. In the context of network programmability where an API is already constructed and ready to be followed, administrators and developers can utilize Postman to debug code and test REST calls that need to be made. Postman is available from <https://www.getpostman.com/> and also offers a browser extension for Google Chrome, available within the Chrome Web Store.

There are similar easy-to-use tools like Insomnia and Paw that are popular as well.

Figure: REST Clients like Insomnia can be helpful in getting started with NX-API REST



NX-API CLI Developer Sandbox

The NX-API CLI Developer Sandbox is an excellent transition tool which can be enabled from the NX-OS command line. After enabling *feature nxapi*, an nginx web server is started on the switch which allows developers or administrators the ability to transcode traditional Nexus command line inputs to REST-based operations via HTTP/HTTPS. It also includes the ability to generate Python code, and supports different message formats like XML or JSON.

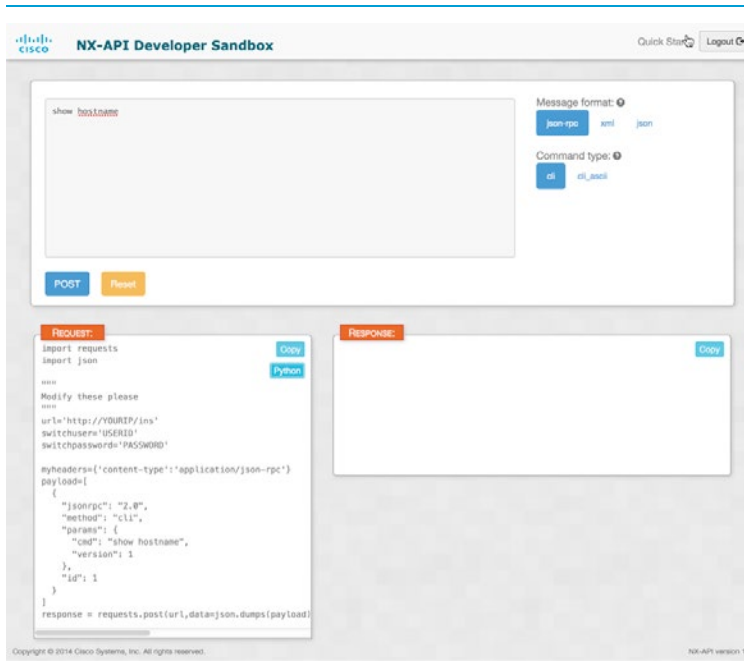
Enabling the NX-API CLI Developer Sandbox:

```
n9k-sw-1(config)# exit
n9k-sw-1# conf t
Enter configuration commands, one per line. End with CNTL/Z.
n9k-sw-1(config)# feature nxapi
n9k-sw-1(config)# show feature | inc nxapi
nxapi                1                enabled
```

Accessing the NX-API CLI Developer Sandbox is accomplished by pointing a web browser to the hostname or IP address of the switch. This functionality is supported on any in-band and out-of-band management IP address.

```
http://<ip or hostname>
```

Enter access credentials to the switch and the NX-API CLI Developer Sandbox will appear.

Figure: NX-API CLI Developer Sandbox

Visore

Visore is an Italian word, which translates to Viewer. It allows users to a mechanism to browse and navigate the management objects (MOs) in the system's management information tree (MIT). This enables users to view the structure of the Open NX-OS DME data-model that backs the NX-API REST interface. Visore enables users to gather information about the data associated with a specific object or a group of objects, typically for the purpose of troubleshooting and event analysis. Visore is also a very useful educational tool to understand how Open NX-OS stores data within with the data model. Visore supports querying by Distinguished Name (DN) or Class.

Note: Visore is not capable of performing configuration operations.

To access Visore, follow these steps:

- Enable NX-API in the configuration of the switch with `feature nxapi`
- Open a web browser and connect to the URL `http://<switch ip address>/visore.html`

The authentication challenge will display where the user will need to enter credentials.

Figure: Cisco NX-API CLI Sandbox Login Screen

The image shows a web-based login form for the Cisco NX-API CLI Sandbox. The form is enclosed in a light gray border. At the top, there is a header bar with the word "Login" in bold black text and a small "x" icon in a square on the right. Below the header, there are two input fields. The first is labeled "Username:" and is currently empty, with a blue border around it. The second is labeled "Password:" and is also empty. At the bottom right of the form, there is a gray button with the word "Login" in black text.

After authenticating, the Visore object store browser is displayed.

Figure: Browsing Cisco NX-API REST Object Store with Visore

Object Store Browser (c) 2012-2013 Cisco Systems, Inc.

Filter

Class or DN:

Property: Op: Val1: Val2:

[Run Query](#)

[Display URI of last query](#)

[Display last response](#)

topSystem	
address	0.0.0.0
childAction	
configIssues	
currentTime	2015-09-29T07:13:52.343+00:00
dn	sys < >
fabricId	1
fabricMAC	00:22:BD:F8:19:FF
id	0
inbMgmtAddr	0.0.0.0
lcOwn	local
modTs	2015-09-21T05:53:25.775+00:00
mode	unspecified
monPolDn	uni/fabric/monfab-default < >
name	N9Kv-1
oobMgmtAddr	0.0.0.0
podId	1
role	unsupported
serial	
state	out-of-service
status	
...	...

Annotations:

- Class (Object) Name: points to the 'topSystem' header.
- DN This is the parent view: points to the 'dn' field.
- Device Name: points to the 'name' field.

Here is a sample BGP configuration, which we will then show in the MIT using Visore:







```

router bgp 65501
  router-id 10.10.10.12
  timers bgp 60 180
  timers prefix-peer-timeout 30
  timers prefix-peer-wait 90
  graceful-restart
  graceful-restart restart-time 120
  graceful-restart stalepath-time 300
  reconnect-interval 60
  fast-external-fallover
  enforce-first-as
    
```

```
event-history periodic
event-history events
event-history cli
address-family ipv4 unicast
  network 10.10.0.0/16
  network 168.10.10.0/24
  network 192.0.0.0/8
  maximum-paths 1
  maximum-paths ibgp 1
  nexthop trigger-delay critical 3000 non-critical 10000
  client-to-client reflection
  distance 20 200 220
  dampen-igp-metric 600
neighbor 10.10.10.11
  remote-as 65000
  dynamic-capability
  timers 60 180
  address-family ipv4 unicast
    next-hop-third-party
```

Visore can be used to examine how the configuration is represented in the data model. For example, we browse to the top-level of the BGP object at **sys/bgp**:

Figure: Browsing Cisco NX-API REST BGP Object Store with Visore

bgpEntity	
adminSt	enabled
childAction	
dn	sys/bgp < >   
lcOwn	local
modTs	2015-09-15T17:39:48.628+00:00
monPolDn	uni/fabric/monfab-default < >   
name	
operErr	
operSt	enabled
status	
uid	0

By navigating deeper into the BGP data model (select the right-arrow next to **sys/bgp**), we can see how BGP data is modeled within the MIT.

bgpInst	
activateTs	2015-09-15T17:39:50.819+00:00
adminSt	enabled
asPathDbSz	0
asn	65501
attribDbSz	100
childAction	
createTs	2015-09-15T17:39:50.129+00:00
ctrl	fastExtFallover
dn	sys/bgp/inst < > ! H
lcOwn	local
memAlert	normal
modTs	2015-09-15T17:39:48.635+00:00
monPolDn	uni/fabric/monfab-default < > ! H
name	
numAsPath	0
numRtAttrib	1
operErr	
snmpTrapSt	disable
status	
syslogLvl	err
uid	0
ver	v4
waitDoneTs	2015-09-15T17:39:50.819+00:00

As we continue to navigate the browser, we can observe object hierarchy and dependency.

Figure: Navigating Deeper into Cisco NX-API REST Object Store with Visore

bgpDom	
always	disabled
bestPathIntvl	300
bgpCfgFailedBmp	
bgpCfgFailedTs	00:00:00:00.000
bgpCfgState	0
childAction	
clusterId	unspecified
dn	sys/bgp/inst/dom-default < >   
firstPeerUpTs	2015-08-19T20:30:47.037+00:00
holdIntvl	180
kaIntvl	60
lcOwn	local
maxAsLimit	0
modTs	2015-09-15T17:39:51.648+00:00
mode	fabric
monPolDn	uni/fabric/monfab-default < >   
name	default
numEstPeers	0
numPeers	1
operRtrId	10.10.10.12
operSt	up
rd	unknown:unknown:0:0
rtrId	10.10.10.12
status	
uid	0

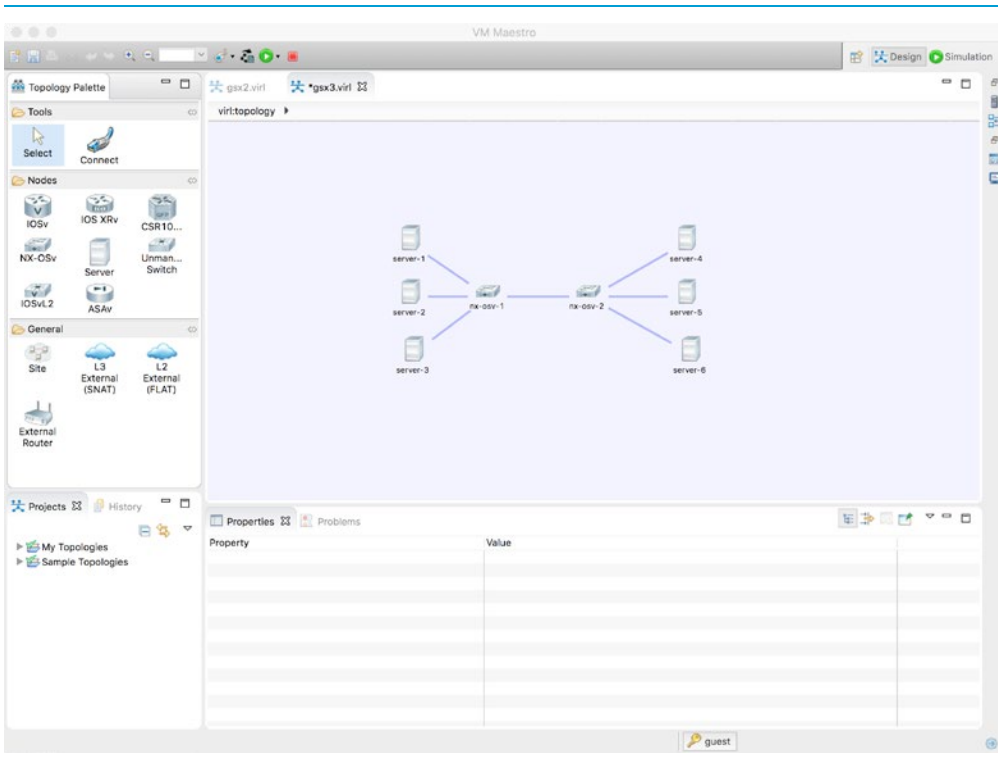
Firebug

For web applications, Firebug is available for debugging and inspecting HTML, CSS and JavaScript. It is available at <http://getfirebug.com/>. Like Postman, it has a Google Chrome extension available within the Google Chrome Web Store. The extension is lightweight and not as feature-dense as the native Firebug application, so be sure to analyze your requirements for debug and inspection.

Virtual Internet Routing Lab (VIRL) and NX-OSv

Cisco VIRL is a network orchestration and virtualization platform that allows users to:

- Design and configure simulated networks with Cisco and third-party elements using a graphical tool
- Set properties on network topologies or individual network elements.
- Generate router and switch configurations, and visualizations of network designs that illustrate a wide range of characteristics such as Layer-2 connectivity, IP addressing, IGP configurations, BGP configurations, and more.
- Start and stop simulations of network designs.
- Interact with active simulations, including the ability to access simulated nodes via console connections, modify running configurations, and extract configurations for use in subsequent simulations for transfer to 'real' routers.
- Link network simulations with external networks for management-, control-, and data-plane connectivity.

Figure: Using VIRL to Simulate an Open NX-OS Environment

VIRL is based in part on OpenStack, and the elements that are used to build network topologies are executed as Nova / KVM virtual machines. These are linked with Neutron-managed networks and subnets.

Many virtualized elements are available, including:

- Cisco NX-OSv
- Cisco IOSv
- Cisco IOS XRv
- Cisco IOS XE (as CSR1000v)
- Cisco ASAv
- Ubuntu 14.04 LTS server

VIRL provides a simple and cost-effective mechanism for developers (or anyone else) to learn the NX-API REST interface and create network applications.

Note: users can leverage the DevNet Open NX-OS Sandbox available at <https://developer.cisco.com/sandbox> for NX-API REST interface testing.

Beyond providing a simulation capability that can be used with NX-API REST, VIRL also provides several other facilities that expose RESTful interfaces that can be used for learning about APIs, including the full set of OpenStack APIs and those exposed by VIRL directly.

For more information on VIRL, including configuration guides, tutorials, and how to purchase VIRL for personal or development use, please visit the VIRL microsite at <http://virl.cisco.com>.

Source Code and Version Control

Why Version Control?

Being organized and tracking changes are two very critical parts to developing, curating, and maintaining code. The use of a source code version control system is extremely important for developers, whether they are working independently or as part of a larger team. There are three primary approaches to version control: local, centralized and distributed. Important questions to ask when selecting a version control system can include the following:

- Who will use the version control system at my company?
- Is any information contained within the source of a proprietary or secure nature?
- Will anyone outside of my company use this code? (see Open Source and Licensing)
- Will access to the repository be required from different machines or locations?
- What is the rate of change of my code?
- How is the version control system being backed up or replicated?

Local version control is easily relatable: create a directory on a machine and copy data or files into it. These local systems evolve into systems that incorporate databases to track changes and commits. While simple to instantiate, local version control systems suffer from several drawbacks including backup and replication, and the inability to share code with a larger community.

Centralized version control expands on local version control by moving the repository to a centralized location, and granting access to the code repository for users within an organization. Subversion, Visual Source Safe and CVS are good examples of centralized version control systems. While this approach solves the issue of granting access to larger teams for code collaboration, it still fails to address failure scenarios of the version control system repository, whether it is housed on a file server, storage array, or other resource. Losing the version control server constitutes a code stop for all developers working on projects within the system.

Distributed version control takes centralized version control a step further. Repositories and code are distributed amongst different servers, which can be distributed between multiple servers or data centers, placed in the cloud, or modeled as a software service (SaaS) offering. Git is a good example of a distributed version control system, and is discussed in the next sec-

tion. Considerations when using a distributed version control system typically focus around access, and whether or not code should be stored locally within an organization or be put onto a cloud-based repository such as GitHub.

Git

For the purposes of this book, *Git* will be discussed as the primary version control tool. In a pure client-server context, the *Git* client is responsible for checking-out/ checking-in code, as well as cloning and building repositories.

The *Git* server can be used to build a distributed version control system for an organization or department. The use of a *Git* server that will be maintained organizationally or departmentally will be based on requirements and constraints outlined during the requirements-gathering phase. It is critical to review this criteria with business stakeholders, particularly security and audit departments.

GitHub

GitHub is a cloud-based repository with a web-based interface for anyone to use to collaborate on projects through building, downloading and providing feedback on code. A social layer is added to the distributed version control system, allowing users to create branches or forks of code, submit bugs (and fixes), watch or tag favorite projects, as well as create documentation for projects and code.

GitHub allows users in different organizations, cities or countries to create and share code. It is geared toward those who embrace the Open Source movement, so it is imperative to review organizational requirements and constraints around code sharing and the licensing that will be applied to any code that is distributed on GitHub.

Cisco DevNet for Open NX-OS

As you begin to explore network programmability and how to leverage APIs with Cisco Nexus switches, a good place to learn and get help is DevNet – Cisco’s primary resource for developers, engineers, and customers who want build applications or services with or around Cisco solutions. DevNet can be found at <http://developer.cisco.com>, and the Open NX-OS microsite can be found at <http://developer.cisco.com/opennxos>.

You can visit the microsite and explore on your own, or take a brief tour here. The landing page for the Open NX-OS site includes links to currently featured content and discussions:

The screenshot shows the landing page for the Cisco Open NX-OS community. The page has a dark blue header with the Cisco logo and navigation links. Below the header, there is a main content area with a welcome message and three call-to-action buttons. The page is divided into two columns: 'FEATURED SCRIPTS' and 'RECENT DISCUSSIONS'.

FEATURED SCRIPTS			
Cisco Chef Recipes	★★★★★ (1) user ratings	3 downloads	20 views
NX-API getting started	☆☆☆☆☆ (0) user ratings	3 downloads	17 views
Ignite - Network bootstrapper	★★★★★ (1) user ratings	3 downloads	15 views
NXAPI Examples	☆☆☆☆☆ (0) user ratings	0 downloads	14 views
Python Samples	☆☆☆☆☆ (0) user ratings	0 downloads	12 views
NexusDash - Monitoring Dashboard	★★★★★ (1) user ratings	0 downloads	10 views

[View all scripts »](#)

RECENT DISCUSSIONS
Can I add a new type and provider for Puppet? Posted by Asha Hegde - Sept. 01, 2015
Where can I find Puppet manifest and Chef recipes? Posted by Asha Hegde - Sept. 01, 2015
What is Ignite? Posted by Asha Hegde - Sept. 01, 2015
What is nxtoolkit? Posted by Asha Hegde - Sept. 01, 2015
What are the important highlights of OpenNXOS? Posted by Asha Hegde - Sept. 01, 2015

[View all discussions »](#)

In the Getting Started section you'll find some quick examples of how to configure common switch features, including interfaces, VLANs, OSPF, and AAA. For each of these, examples are provided for Puppet, NX-API, Python, Chef, and Ansible.

The screenshot shows the Cisco Open NX-OS Getting Started page. The page has a dark blue header with the Cisco logo and navigation links: Open NX-OS, Getting Started, Code Shop, Discussions, Article Center, and API. A search bar is also present. Below the header, there are two columns of navigation links. The left column is titled 'CONFIGURATION' and includes links for Interface, VLAN, OSPF, TACACS, AAA, and MONITORING. The right column is also titled 'CONFIGURATION' and includes a link for Interface. The main content area is titled 'Interface' and contains the text: 'Program and manage interfaces. You can configure interfaces using different configuration management tools. Following example shows how to display Nexus 9000 interfaces, bring the interfaces up or down, configure IP address.' Below this text are tabs for 'Puppet', 'NX-API', 'Python', 'Chef', and 'Ansible'. The 'NX-API' tab is selected. Below the tabs is a 'Script' tab and a 'Sandbox' icon. The script content is as follows:

```
#Config physical ethernet port
POST /api/mo/sys/phys-[eth1/1].json?query-target=self HTTP/1.1
Host: switch_ip
Content-Type: application/json
Cache-Control: no-cache
Postman-Token: 5c811b93-249d-b10c-653a-8ba12107416c

{
  "l1PhysIf": {
    "attributes": {
      "accessVlan": "vlan-321",
      "adminSt": "down",
      "bw": "0",
```

In the discussions section, you can start or participate in discussion about NX-OS and NX-API programming, search for solutions to issues that may have been encountered and resolved by others, or get help from community experts.

Home > Discussions

Discussions Recent Featured

All Automation Extensibility APIs and Scripting Programmability - General Misc.

	votes	replies	views	Discussion Title	Posted by	Time
1	1	22		Can I add a new type and provider for Puppet?	ashegde2	Sept. 1, 2015, 4:27 a.m.
0	1	14		What is nxtoolkit?	ashegde2	Sept. 1, 2015, 3:51 a.m.
0	1	6		What is Ignite?	ashegde2	Sept. 1, 2015, 3:44 a.m.
0	1	9		Where can I find Puppet manifest and Chef recipes?	ashegde2	Sept. 1, 2015, 3:37 a.m.
0	1	10		What are the important highlights of OpenNXOS?	ashegde2	Sept. 1, 2015, 3:37 a.m.

Page 1

[Start a Discussion](#)

RECENT ACTIVITY

Devarshi Shah commented on [Can I add a new type and provider for Puppet?](#)
Can I add a new type and provider are not present for a particular feature, you can you cisco_command_config to configure the 4 weeks ago

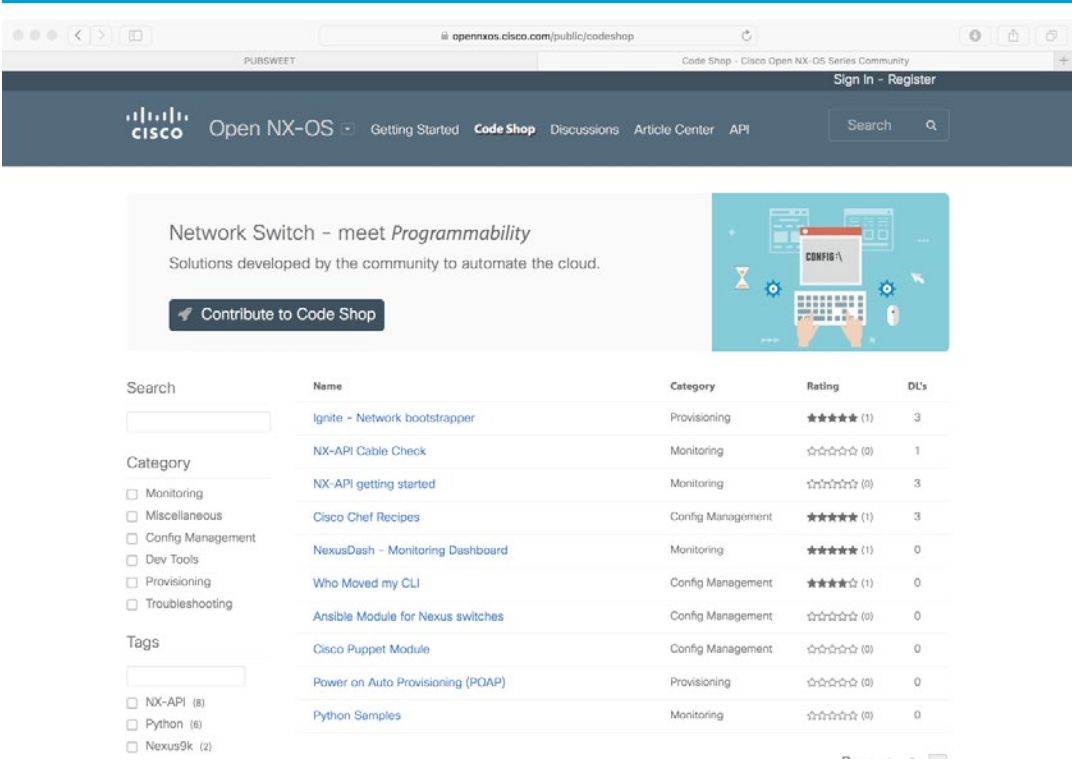
Ranganath Rao commented on [Where can I find Puppet manifest and Chef recipes?](#)
Visit <http://github.com/cisco/Puppet-modules> <https://github.com/cisco/cisco-network-puppet-module> Chef 4 weeks, 1 day ago

Ranganath Rao commented on [What is Ignite?](#)
Ignite is a server-side tool that supports Nexus POAP – it enables users to design their network topology and configuration and 4 weeks, 1 day ago

Ranganath Rao commented on [What is nxtoolkit?](#)
nxtoolkit is a set of python libraries that allow basic configuration of the Cisco Nexus 9000/3000 Series Switch. It is intended to 4 weeks, 1 day ago

Asha Hegde commented on [What are the important highlights of](#)

The Code Shop section provides a means through which you can share scripts, programs, recipes, and other samples for the community to review and use. You can also search, sort and browse code samples submitted by others.



Network Switch - meet *Programmability*
Solutions developed by the community to automate the cloud.

[Contribute to Code Shop](#)

Search	Name	Category	Rating	DL's
	ignite - Network bootstrapper	Provisioning	★★★★★ (1)	3
	NX-API Cable Check	Monitoring	☆☆☆☆☆ (0)	1
	NX-API getting started	Monitoring	☆☆☆☆☆ (0)	3
	Cisco Chef Recipes	Config Management	★★★★★ (1)	3
	NexusDash - Monitoring Dashboard	Monitoring	★★★★★ (1)	0
	Who Moved my CLI	Config Management	★★★★☆ (1)	0
	Ansible Module for Nexus switches	Config Management	☆☆☆☆☆ (0)	0
	Cisco Puppet Module	Config Management	☆☆☆☆☆ (0)	0
	Power on Auto Provisioning (POAP)	Provisioning	☆☆☆☆☆ (0)	0
	Python Samples	Monitoring	☆☆☆☆☆ (0)	0

Category

- Monitoring
- Miscellaneous
- Config Management
- Dev Tools
- Provisioning
- Troubleshooting

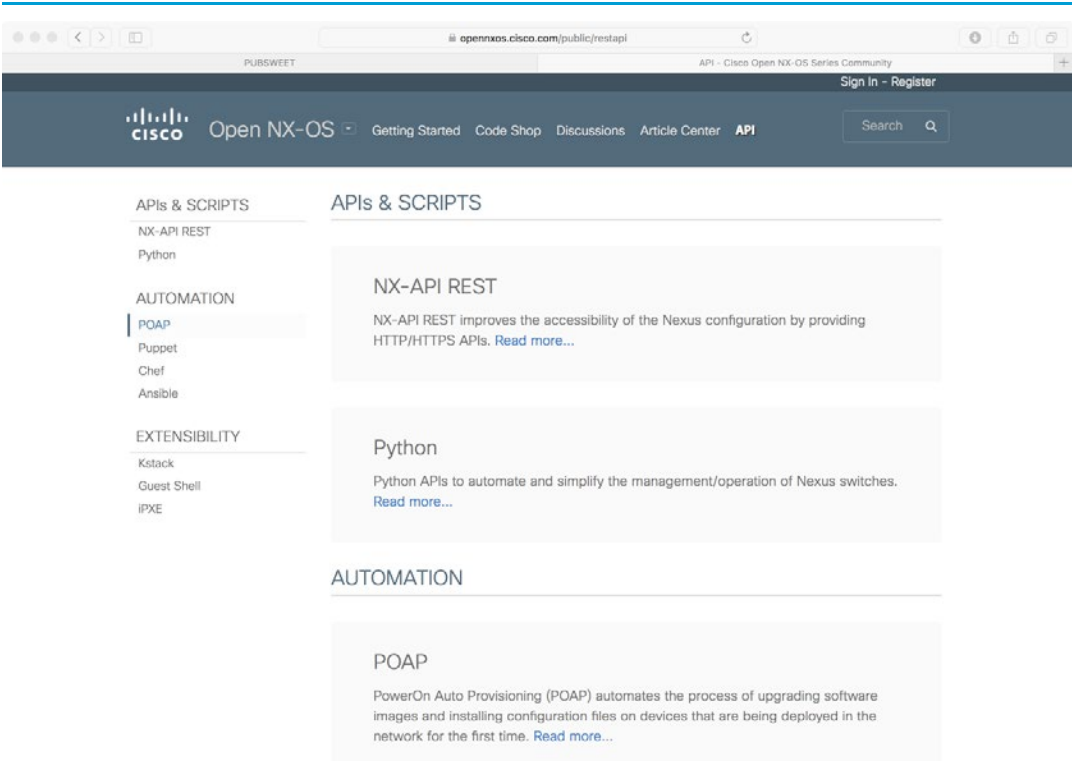
Tags

- NX-API (8)
- Python (6)
- Nexus9k (2)

In the Article Center you can browse, search, and review announcements, configuration guides, and other content submitted to the community by Cisco Nexus / NX-API engineers.

The screenshot shows a web browser window displaying the Cisco Open NX-OS Article Center. The browser's address bar shows the URL `opennxos.cisco.com/public/articles`. The page header includes the Cisco logo, the text "Open NX-OS", and navigation links for "Getting Started", "Code Shop", "Discussions", "Article Center", and "API". A search bar is located in the top right corner. Below the header, the page content is organized into two main sections: "Article Center" and "RECENT ACTIVITY". The "Article Center" section features a "Recent" tab and a search box. Below this, a list of five articles is displayed, each with a profile picture, a title, and a date of August 31, 2015. The articles are: "Register for DemoFriday: Achieve Network Programmability & Seamless Scalability with Cisco's NX-API", "TechWise Live Webinar - Maximize Programmability with NX-API REST", "Program and Automate Your Switches with the New, More Open NX-OS", "Cisco DemoFriday: Maximizing Network Programmability and Automation with Cisco Open NX-OS", and "Open Networking with Cisco ACI and Open NX-OS". The "RECENT ACTIVITY" section is currently empty. At the bottom of the page, there is a footer with the text "Page 1".

The API section of the Open NX-OS microsite is where you'll find detailed information and configuration guides for the complete portfolio of NX-OS programmability solutions, including for the NX-API REST interface, Python scripting, and automation tools such as POAP, Puppet, Chef, and Ansible.



Within the NX-API REST subsection, you'll find a complete reference on how to use the NX-API REST interface to configure Open NX-OS features and capabilities - physical interfaces, logical interfaces, protocols, etc.

The screenshot shows a web browser displaying the Cisco Open NX-OS API REST SDK documentation. The page title is "Overview of the Cisco Nexus 9000 Series NX-API REST SDK". The navigation menu on the left includes items like "Getting Started with the Cisco Nexus 9000 ...", "Configuring RBAC Settings", "Configuring BGP", "Configuring Interfaces and Port Channels", "Configuring a Subinterface", "Configuring SVIs", "Configuring VRRP", "Running CLIs Through Route Policy Manager", "Configuring an IP Route", "Configuring STP Settings", "Configuring ACLs", "Configuring QoS", "Configuring UDLD", "Additional Configuration Areas", "Querying Interface and VLAN Counters an...", "Running the NX-API REST from the Guest ...", and "Using the Managed Object Browser". The main content area starts with an introduction: "On Cisco Nexus devices, configuration is performed using command-line interfaces (CLIs) that run only on the device. NX-API REST improves the accessibility of the Nexus configuration by providing HTTP/HTTPS APIs that:" followed by two bullet points: "• Make specific CLIs available outside of the switch." and "• Enable configurations that would require issuing many CLI commands by combining configuration actions in relatively few HTTP/HTTPS operations." Below this, it states "NX-API REST supports show commands, basic and advanced switch configurations, and Linux Bash." and "NX-API REST uses HTTP/HTTPS as its transport. CLIs are encoded into the HTTP/HTTPS POST body. The NX-API REST backend uses the Nginx HTTP server. The Nginx process, and all of its children processes, are under Linux cgroup protection where the CPU and memory usage is capped. If the Nginx resource usage exceeds the cgroup limitations, the Nginx process is restarted and restored." The right sidebar has tabs for "Example Request" and "JSON Response". At the bottom left, it says "V1.0 Copyright © 2015 by Cisco Systems, Inc. All Rights Reserved. Published August 27, 2015". At the bottom center, it says "Understanding the Model Driven Programming Framework".

The Open NX-OS DevNet community should be your primary resource for information and updates on Open NX-OS programmability. Start there whenever you need help, want to share code, or browse for new Open NX-OS or NX-API REST capabilities.

Learning and Helpful Resources

This section will outline some useful resources that developers and administrators can use for self-learning and help in developing projects, simple or complex. Most of these resources are accessed or delivered via the web, and some include platforms where a learning and feedback environment is completely integrated into the website. This level of accessibility makes it possible for anyone interested to sign up and begin learning, even if they have no previous experience coding or developing programs.

Learning a Language

Prior to the Internet, the two primary avenues for learning a programming language were textbooks or university courses. However, the Internet now enables students to teach themselves a language in an accessible and interactive way. The following resources are free unless noted, and are available for public use.

Code Academy - <http://www.codecademy.com/>

Interactive, self-paced website to learn programming languages such as Python, Ruby and C++.

Google's Python Class - <https://developers.google.com/edu/python/>

Instructional website for learning Python, with example code and community discussion forums.

Coursera - <https://www.coursera.org/>

eLearning site for online classes in all disciplines and fields. Rice University and University of Michigan offer multi-week Python courses for beginner and intermediate skill levels. Instructors are available for grading and feedback of assignments.

edX - <https://www.edx.org/>

eLearning website started by MIT and Harvard, offering online and self-paced classes in multiple disciplines and fields. Courses in Python are offered by MIT and the University of Texas at Arlington.

Learn Python the Hard Way - <http://learnpythonthehardway.org/>

Website for Python learners with videos, documentation and sample code for use. Fee of \$29.95 USD.

Please note: this is by no means meant to be an exhaustive list of programming courses or resources available on the web, but is simply meant to point the reader in a direction for finding a solution which best fits their learning style. Many universities and community colleges also offer online and instructor-led courses.

Getting Help

Every programmer and developer will eventually come to an error message they can't debug on their own, or a program that does not flow properly or produce the expected output or results. Thankfully there are resources available online for assistance.

Cisco DevNet - <http://developer.cisco.com/opennxos/>

Online source of information for learning concepts, APIs, code examples and more, with online communities for posting questions or having discussions.

Stack Overflow - <http://stackoverflow.com/>

Online discussion forum for questions or requesting help with coding approaches or problems in any programming language.