



NSO

Network Automation

ABSTRACT

Learn how works automation with NSO and YANG modeling language.

Andreetta,R,Riccardo,JP11 R

Who put the doc together, searching a lot of stuff on the web. Original authors have been referenced at the beginning of their paragraphs.

1 Contents

1	NSO	7
2	Learning YANG.....	8
2.1	Modules.....	8
2.2	YANG Base Types.....	10
2.3	Typedef.....	10
2.4	Type Restrictions	10
2.5	Common YANG types	11
2.6	YANG Data Definitions.....	12
2.6.1	Grouping Statement	12
2.6.2	Grouping Statement with Refine.....	12
2.6.3	Leaf Statement	13
2.6.4	Container Statement	13
2.6.5	Leaf-list Statement	13
2.6.6	List Statements	14
2.6.7	Attributes of list an leaf-lists.....	14
2.6.8	Keys.....	14
2.6.9	Multiple Keys	15
2.7	Leafref.....	15
2.7.1	Multiple Key Leafref	15
2.7.2	Deref() XPATH Operator	16
2.7.3	Other DEREf() Examples.....	17
2.7.4	Understanding Path Expansions.....	19
2.7.5	More on leafrefs	20
2.8	MUST Statement	22
2.9	YANG keywords	23
2.9.1	Unique	23
2.9.2	Range	23
2.9.3	Error-message.....	23
2.9.4	Template tag operations (merge, replace, create, ncreate, delete)	24
2.9.5	count.....	25
2.9.6	pattern.....	26
2.9.7	starts-with.....	26
2.9.8	min-elements	27
2.9.9	When.....	27
2.9.10	tailf:hidden	27

2.9.11	Leafrefs with conditional path references	27
3	YANG.....	28
3.1	It's all about describing the data structure	29
3.2	Let's learn the YANG syntax.....	29
3.3	Writing our first YANG statements.....	29
3.4	How to create a custom data type	30
3.5	Understanding configuration data and state data	31
3.6	Adding our configuration data	31
3.7	Adding our state data	32
3.8	How to validate a YANG module	32
3.9	How to view the schema tree.....	33
3.10	Introducing "yang2dsdl"	33
3.11	How to validate a data instance	33
3.12	Possible YANG keywords	35
3.13	YANG identities and affiliations.....	35
3.14	YANG modules extensions.....	38
3.15	Another extension for ip addresses	40
4	NSO.....	41
4.1	Fastmap and NED	43
4.2	Accessing the Network (NEDs)	44
4.3	L2 vpn service example.....	45
4.4	Loops and arrays inside the xml file	48
4.5	Programming statements inside xml.....	51
4.6	L3 vpn service	51
4.7	Nested Services	54
4.8	External data books	55
4.9	Makefile.....	57
4.10	Interfaces choice and references	58
4.11	Matching devices of a specified group	60
4.12	Useful NCS commands.....	61
4.13	Troubleshooting and debug	63
4.14	Operations and maintenance	66
4.15	Configuration file	67
4.16	Complex checks	67
4.16.1	Retrieving data from configs	69
4.16.2	Reconciling configs example.....	70

4.16.3	Sending notifications	75
4.16.4	Python background worker for NSO.....	76
4.17	Writing a Python background worker for Cisco NSO - finale	90
4.17.1	Thread whispering	91
4.17.2	CDB configuration changes.....	93
4.17.3	Monitoring HA events.....	93
4.17.4	Child process liveness monitor	95
4.17.5	Hiding things from the bg function.....	96
4.17.6	A selectable queue	96
4.17.7	A library with a simple user interface.....	100
4.17.8	Finale	101
4.18	Python checks on CLI output	102
4.19	Python action dry-run commit custom	103
4.20	Python action callback example	104
4.21	REST queries	105
4.21.1	Listing device interfaces	108
4.22	Stacked services (one father service calls multiple children).....	108
4.22.1	How to use Stacked Services to improve performance?.....	108
5	NSO installation, cfg examples	112
5.1	NCS Installation.....	112
5.2	Install NCS.....	112
5.3	Setup NCS	112
5.4	What are the packages available.....	114
5.5	Compile a new NED	114
5.6	To link the newly compiled NEDS.....	115
5.7	Add Emulated Devices to the Network	116
5.8	Create device with Net Sim	116
5.9	Create Device Group.....	117
5.10	Create Customer.....	119
5.11	Lets now create a Device template	119
5.12	Now Lets create a Service.....	122
5.13	Lets Start with creating an L2 VPN YANG template	122
5.14	Now Lets create a Device Template based on the Actual CLI configuration.....	125
5.15	Assign Service Instances to Customers.....	134
5.16	Creating a Custom Auth Group for "Real Cisco Devices"	134
5.17	You Can Install Custom Commands at <code>/ncs-run/scripts/command</code>	137

5.18	NCS Troubleshooting commands	138
5.19	Service Model Optimisation	138
5.20	<code>optimized.l2vpn.yang</code>	138
5.21	<code>optimized.l2vpn-template.xml</code>	140
5.22	<code>optimized.l2vpn.Makefile</code>	141
5.23	NSO 200 Base Yang	142
6	Cisco NSO and Ansible together	148
6.1	What is Ansible?	148
6.2	Cisco NSO Ansible modules	148
6.2.1	The <code>nso_config</code> module	148
6.2.2	The <code>nso_show</code> module	149
6.2.3	The <code>nso_action</code> module	150
6.3	Lab setup	150
6.3.1	Reserving the Sandbox	150
6.3.2	Setting up Cisco NSO and local environment	151
6.3.3	Examine Cisco NSO	151
6.3.4	Examine the local environment	152
6.4	Step 1: Manage Cisco NSO devices with Ansible	152
6.4.1	Explore the inventory	152
6.4.2	Explore the <code>add_devices.yml</code> playbook	153
6.4.3	Run the playbook	154
6.4.4	Connect to Cisco NSO and verify that devices are added	155
6.4.5	Explore the <code>remove_devices.yml</code> playbook	155
6.4.6	Run the playbook	156
6.4.7	Connect to Cisco NSO and verify that devices are actually removed	156
6.4.8	Put devices back to the configuration	156
6.5	Step 2: Manage device configurations	156
6.5.1	Explore the <code>manage_loopbacks.yml</code> playbook	157
6.5.2	Verify the configuration for interfaces	158
6.5.3	Execute the playbook	159
6.5.4	Modify the configuration	159
6.5.5	Execute the playbook	159
6.6	Step 4: Gather live status from devices	160
6.6.1	Explore the <code>get_version.yml</code> playbook	160
6.6.2	Execute the playbook	161
6.6.3	Verify the outputs	161

6.7	Step 5: Manage service instances with Ansible.....	162
6.7.1	Explore the <code>manage_svi_service.yml</code> playbook	162
6.7.2	Explore service instance configuration files	163
6.7.3	Configure service instances	163
6.7.4	Verify the configuration on Cisco NSO	164
6.7.5	Modify the service instance configuration	164
6.7.6	Remove the service instance	165
6.7.7	Verify the configuration on Cisco NSO	165
7	Making changes.....	165
7.1	Setting configuration with PUT	166
8	Invoking actions.....	167
9	Netsim for NSO	168
9.1	Purpose.....	168
9.2	Documentation.....	168
9.3	Dependencies	169
9.4	Build instructions	169
9.5	Usage examples	169
9.5.1	config	169
9.5.2	create-device	169
9.5.3	add-device	169
9.5.4	start.....	169
9.5.5	load	169
9.5.6	update-network.....	169
9.6	Creating and managing devices.....	169
10	Netsim wrapper for NSO	171
10.1	Introduction.....	171
10.2	Pre-requisites.....	171
10.3	Installation and Downloads	171
10.4	Features.....	171
10.4.1	Delete a device(s) from topology	171
10.4.2	Create Network/Device Template.....	172
10.4.3	Create Network/Device From Template	172
10.4.4	How to choose the Templates and How they look (Template options).....	172
10.5	Help.....	174
10.6	FAQ	175
11	Creating Network Configurations with Jinja.....	175

12	Using YAML and Jinja to Create Network Configurations	178
13	Automating Your Network Operations, Part 1 – Ansible Basics.....	182
13.1	Configuring settings on an IOS device	183
14	Automating Your Network Operations, Part 2 – Data Models.....	184
14.1	Keeping your IT infrastructure operational.....	184
14.2	Where do you get help for the modules that underpin the automation?	184
14.3	So, what is a data model?.....	184
15	Automating Your Network Operations, Part 3 – Data-Driven Ansible	186

Copyright disclaimer: I've found most of the content of this document on the internet, on blogs, Cisco communities. To be honest, Cisco's documentation of its products is, averagely speaking, the best in the world. Compare it to that of other vendors, before saying it's not true because once in your life you upgraded version 1.3b to 1.xc and you had a problem.

NSO is in my opinion is an exception, in a bad way. Finding good documentation is difficult, **dCloud** is an extremely good resource since the Labs are quite simple and well written, but if you have to do something useful related to real life examples (just to make an example, managing notifications to/from NSO ... read chapter 4.16 to get an idea of what I mean), it's gonna be much more complex and you will often find yourself to be alone, trying to solve all your issues.

The labs I've found on **dCloud** were written by:

Gregg Schudel, Software Solutions Architect, 'NSO Americas Sales'.

Here's another active community:

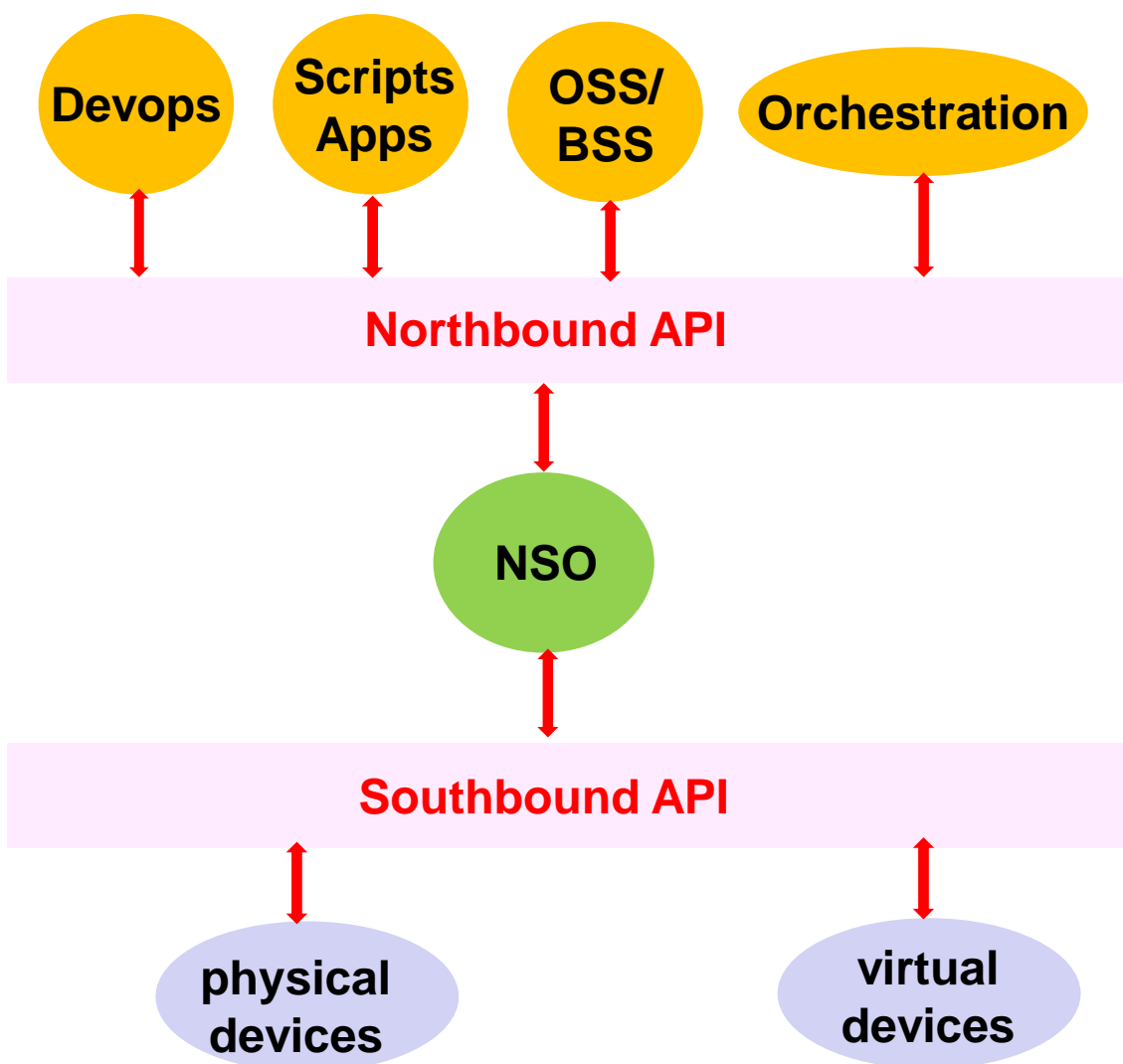
<https://community.cisco.com/t5/nso-developer-hub/ct-p/5672j-dev-nso>

... where Cisco employees often help and answer questions.

As usual, I gather data about topics I'm interested in, trying to give it an understandable structure, to be possibly used as a reference in the future. I'm **not selling any** of this data as a book, not making out ANY money of it, **nor directly nor indirectly with internet web traffic**. It's not my intellectual property, since I DO NOT write it from scratch on my own, I ask myself questions and I try to find the answers on the internet. It's a time consuming task, but there's not that much added value (at least in my opinion). And I don't want to put myself in trouble, asking to everyone if I could publish something or something else, to share a way to divide incomes: **if there is no income, there's nothing to divide**. Github itself, as far as I know, doesn't make money with advertising or user profiling, but having private companies paying for its services (and 'open sourcers' using them for free). Moreover, whenever possible I always put the references, being it internet links, videos, names and surnames of the people from which I have copied something. Should I have forgotten someone, just let me know and I'll add it. Probably people will also need to re-check things on the original site, thus I'm not even driving traffic away from them.

1 NSO

“Cisco **Network Services Orchestrator** (enabled by Tail-f)”, since this is the changed name (due to the “Sell As A Name” service) of the product developed by a Swedish company acquired in 2014 by Cisco Systems. This is why you still get ‘ncs’ on their CLI. Automation was already on fire in those years, surfing the wave of SDN, VNF, and many other super fun acronyms. It is worldwide known and accepted that companies are slow in developing new services because (we)men (for the sake of inclusion) are lazy and prone to errors. By the way, I started doing ‘automation’ since I started working in the networking world in 2005. For the poor people working in the IT world, it’s probably the only way to survive and maintain their mental brain sanity, for those who work as consultants it’s even more necessary to do the job that 3.5 normal employees don’t want to do.



Benefits of this product are ... could you have guessed some of them ? They are all magical marketing keywords, belonging to that very small marketing dictionary that everybody is nowadays addicted to.

“

- **Accelerate revenue-generating services** with automated, self-service, on-demand provisioning that reduces activation times from months to minutes.

- **Increase agility and scale** with the capability to create, reconfigure, and repurpose services in real time with virtually unlimited horizontal scale.
- **Simplify your network operations** by automating the end-to-end service lifecycle and reducing manual configuration steps by up to 90 percent.
- **Promote open architectures** with multi-vendor capabilities to automate advanced device features, easily bundle multiple network services, and help assure them in real time.
- **Ensure trusted operations** with the industry's most robust feature set for ensuring trust across the network, other management systems and NSO clients, apps and users.

...

Cisco® Network Services Orchestrator (NSO) enabled by Tail-f® is in production in all of the top ten service providers and a number of large enterprises today. It provides end-to-end lifecycle service automation to design and deliver high-quality services faster and more easily. It lets you create and change services using standardized models **without the need for time-consuming custom coding** or service disruption. In addition, you can help ensure that your services are delivered in real time and meet even the most stringent service-level agreements (SLAs). The orchestrator **automates the full range of multivendor devices** across both physical and virtual environments and continually refines and repackages network services at the speed of your business."

The above has been taken from the following link:

<https://www.cisco.com/c/en/us/solutions/service-provider/solutions-cloud-providers/network-services-orchestrator-solutions/at-a-glance.html#~true-agility-with-automation>

We'll go much more in detail later on, but now we start diving into 'YANG', which is a language through which are described services in NSO (and this is why it's important).

2 Learning YANG

2.1 Modules

This file is based on the Youtube video:

<https://www.youtube.com/watch?v=AdIcYrz3AjU&t=1854s>

... published by '**Tail-f Systems**'.

Below is a sample YANG file generated by the:

```
ncs-make-package --service-skeleton template ospf_deploy .
```

We will use this for our learning and understanding for the YANG file and so the same based on real life YANG file examples. Notes are added as comments in the file below:

```
module ospf_deploy {
    namespace "http://com/example/ospf_deploy";
    prefix ospf_deploy;

    import ietf-inet-types {
        prefix inet;
    }
    import tailf-ncs {
        prefix ncs;
    }
    organisation "ACME Inc";
```

```

revision 2007-01-02 {
    description "Second version v2";
}
}

```

Note above that each module begins with the module declaration, the filename **should equal the module name**. Each module is uniquely identified in the system with the **namespace prefix** below is how the namespace will be referenced in the file going forward. Something like `ospf_deploy:` followed by something. Note that YANG is XML Definition language and maps one to one with XML. The `import` and `include` add other modules to the YANG model. Using the revision information in , when working with NETCONF, it will advertise the version which will make the management device know what s the device capable of.

import

A yang imports is similar to including a Header file in a C Code.

Include

Include statement is used to pull submodules into a main. A module does not have to be contained within one file. You can decompose it for ease of maintenance desing.

Submodules

A submodule is written in a separate `acme-system.yang` file and does not have a namespace of its own. (Notice the same in the picture below). The submodule is included in the parent module, So parent module can refer → to the submodule but submodule cannot refer to items in the parent module .

Submodules

```

module acme-module {
    namespace "...";
    prefix acme;

    import "ietf-yang-types" {
        prefix yang;
    }
    include "acme-system";

    organization "ACME Inc.";
    contact joe@acme.example.com;
    description "Module describing the
        ACME products";
    revision 2007-06-09 {
        description "Initial revision.";
    }
}

```

```

submodule acme-system {
    belongs-to acme-module {
        prefix acme;
    }

    import "ietf-yang-types" {
        prefix yang;
    }

    container system {
        ...
    }
}

```

Each submodule belongs to one specific main module

Attention: The submodule cannot reference definitions in main module

2.2 YANG Base Types

Type Name	Meaning
int8/16/32/64	Integer
uint8/16/32/64	Unsigned integer
decimal64	Non-integer
string	Unicode string
enumeration	Set of alternatives
boolean	True or false
bits	Boolean array
binary	Binary BLOB
leafref	Reference
identityref	Unique identity
empty	No value, void
	...and more

2.3 Typedef

Below is the a Typedef defined for `percent` . The leaf `completed` inherits the typedef `percent`.

```
typedef percent {  
    type unit16 {  
        range "0 .. 100"  
    }  
    description "Percentage";  
}  
  
leaf completed {  
    type percent;  
}
```

2.4 Type Restrictions

Notice below how Restrictions are applied on `derived-int32` .

```
typedef my-base-int32-type {  
    type int32 {
```

```

    range "1..4 | 10..20" # 1 to 4 and 10 to 20
  }
}

typedef derived-int32 {
  type my-base-int32-type {
    range "11..max"; # Derived from the typedef above but is only limited to
11 to 20 .
  }
}

```

2.5 Common YANG types

Common Networking Data types are stored in **ietf-yang-types** (RFC 6021) like the following. These can be added to your YANG file using the following.

```

import `ietf-yang-types` {
  prefix yang
}

leaf remote-ip {
  type yang:ipv4-address { # Here we refer the ietf yang type for IPv4 address.
    pattern "10\\.0\\.0\\.0\\.[0-9]+";
  }
}

```

counter32/64	ipv4-address
gauge32/64	ipv6-address
object-identifier	ip-prefix
date-and-time	ipv4-prefix
timeticks	ipv6-prefix
timestamp	domain-name
phys-address	uri
ip-version	mac-address
flow-label	bridgeid
port-number	vlanid
ip-address	... and more

2.6 YANG Data Definitions

2.6.1 Grouping Statement

Grouping can contain any YANG structure (leafs or containers etc) . In this example we group

```
module ospf_deploy {
  namespace "http://com/example/ospf_deploy";
  prefix ospf_deploy;

  import "ietf-inet-types" {
    prefix inet;
  }

  grouping target {
    leaf address {
      type inet:ip-address;
      description "Target IP";
    }
    leaf port {
      type inet:port-number;
      description "Target port";
    }
  }

  container peer {
    container destination {
      uses target;
    }
  }
}
```

Notice that when you view the actual output of the code above , it lists the container structure starting with `peer` , `destination` and then adds the `target` group to it .

```
#pyang -f tree ospf_deploy.yang module: ospf_deploy
+--rw peer
  +--rw destination
    +--rw address?      inet:ip-address
    +--rw port?         inet:port-number
```

2.6.2 Grouping Statement with Refine

So with `refine` we further take the above example and refine the grouping with some constraints or defaults. In the example below we set the default value to 80.

```
grouping target {
  leaf address {
    type inet:ip-address;
    description "Target IP";
  }
  leaf port {
    type inet:port-number;
    description "Target port";
  }
}

container peer {
  container destination {
    uses target {

```

```

        refine port {
            default 80;
        }
    }
}

```

2.6.3 Leaf Statement

A leaf is a single item and can have multiple attributes.

```

leaf host-name {
    type string;
    mandatory true;
    config true;
}

```

Attributes for leaf

config	Whether this leaf is a configurable value ("true") or operational value ("false"). Inherited from parent container if not specified
default	Specifies default value for this leaf. Implies that leaf is optional
mandatory	Whether the leaf is mandatory ("true") or optional ("false")
must	XPath constraint that will be enforced for this leaf
type	The data type (and range etc) of this leaf
when	Conditional leaf, only present if XPath expression is true
description	Human readable definition and help text for this leaf
reference	Human readable reference to some other element or spec
units	Human readable unit specification (e.g. Hz, MB/s, °F)
status	Whether this leaf is "current", "deprecated" or "obsolete"

2.6.4 Container Statement

A container is used to organize the leaves in a structure. It does not have type of its own.

```

container system {
    containers services {
        container ssh {
            presence "Enables SSH"
            description "SSH Service Specific configuration"
        }
    }
}

```

2.6.5 Leaf-list Statement

It's is a list of items. Do not see this as an array .

```

leaf-list domains-search {
    type string;
    ordered-by user; # How the list is ordered.
    description "List of domain names to search";
}

```

2.6.6 List Statements



```
list user {
  key name;
  leaf name {
    type string;
  }
  leaf uid {
    type uint32;
  }
}

leaf full-name {
  type string;
}
leaf class {
  type string;
  default viewer;
}
```

Think of Lists as a Table of Items , `key` is the key of the data table.

2.6.7 Attributes of list and leaf-lists

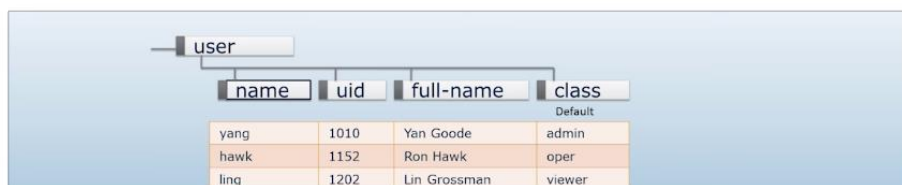
Attributes for list and leaf-list

max-elements	Max number of elements in list. If max-elements is not specified, there is no upper limit, i.e. "unbounded"
min-elements	Min number of elements in list. If min-elements is not specified, there is no lower limit, i.e. 0
ordered-by	List entries are sorted by "system" or "user". System means elements are sorted in a natural order (numerically, alphabetically, etc). User means the order the operator entered them in is preserved. "ordered-by user" is meaningful when the order among the elements have significance, e.g. DNS server search order or firewall rules.

2.6.8 Keys

The key field is used to specify which row are we referring to.

Keys



The key field is used to specify which row we're talking about.

```
/user{yang}/name = yang
/user{yang}/uid = 1010
/user{yang}/class = admin
```

No two rows can have same key value

```
/user{ling}/class = viewer
```


2.6.9 Multiple Keys

Notice in the example below we have the key "ip prefix" allowing us to select based on two keys , IP and Prefix.

Multiple keys

ip	prefix	next-hop	metric
16.40.0.0	16	220.40.0.1	20
16.42.0.0	16	82.193.16.1	40
16.42.0.0	24	82.193.16.6	50

Multiple key fields are needed when a single key field isn't unique.

Key fields must be a unique combination

```
list route {  
  key "ip prefix";  
  ...  
}
```

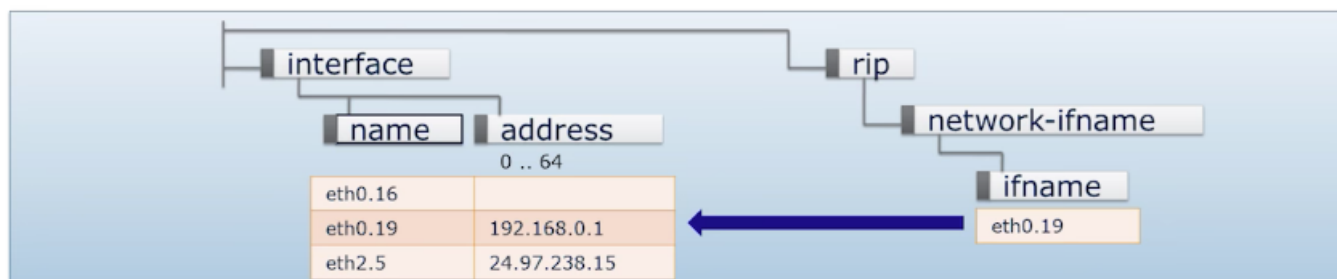
```
/route{16.40.0.0 16}/next-hop  
= 220.40.0.1
```

Key order significant

2.7 Leafref

A Leafref can refer to another leaf . So basically what it means is , the only values that can be selected are the values the Leafref is pointing to.

Leafref



Here, the RIP routing subsystem has a list of leafrefs pointing out existing interfaces

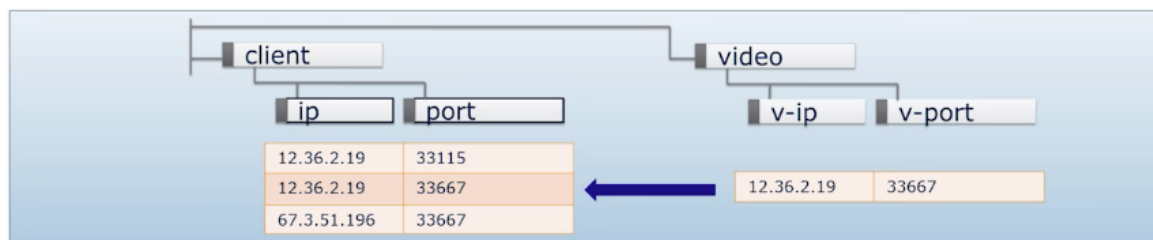
```
container rip {  
  list network-iframe {  
    key iframe;  
  
    leaf iframe {  
      type leafref {  
        path "/interface/name";  
      }  
    }  
  }  
}
```

2.7.1 Multiple Key Leafref

In the example below , a give set of IP and Port is to be selected from the client table. Now selecting the Ip Address is easy , but there are duplicate IP Addresses .

Having the Xpath of `ip=current()` helps us go back in the tree and ensure integrity by limiting the scope to the current v-ip in question .

Multiple Key Leafref



```

container video {
  leaf v-ip {
    type leafref {
      path "/client/ip";
    }
  }
  leaf v-port {
    type leafref {
      path "/client[ip=current()/../v-ip]/port";
    }
  }
}

```

©13 TAIL-F all rights reserved

MAY 27, 2013 27

2.7.2 Deref() XPATH Operator

Now looking at the example above of Leafref, if the number of keys increases (v-ip, v-port and v-stream) it will get convoluted in the nesting of the `current()` pointer. This is made easy by the `deref()` operator.

NOTE: I wonder how this could be defined easy ... you will try to compile your YANG model adding and removing randomly double dot points hoping to see what you expect to.

Deref() XPATH Operator

<pre> container video { leaf v-ip { type leafref { path "/client/ip"; } } leaf v-port { type leafref { path "/client [ip=current()/../v-ip]/port"; } } leaf v-stream { type leafref { path "/client [ip=current()/../v-ip] [port=current()/../v-port] /stream"; } } } </pre>	<pre> container video-deref { leaf v-ip { type leafref { path "/client/ip"; } } leaf v-port { type leafref { path "deref(..v-ip) ../port"; } } leaf v-stream { type leafref { path "deref(..v-port) ../stream"; } } } </pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

©13 TAIL-F all rights reserved

MAY 27, 2013

<http://www.yang-central.org/twiki/pub/Main/YangTools/pyang.1.html>

The `deref` function follows the reference defined by the first node in document order in the argument node-set, and returns the nodes it refers to. If the first argument node is an instance-identifier, the function

returns a node-set that contains the single node that the instance identifier refers to, if it exists. If no such node exists, an empty node-set is returned.

If the first argument node is a leafref, the function returns a node-set that contains the nodes that the leafref refers to. If the first argument node is of any other type, an empty node-set is returned. The following example shows how a leafref can be written with and without the deref function:

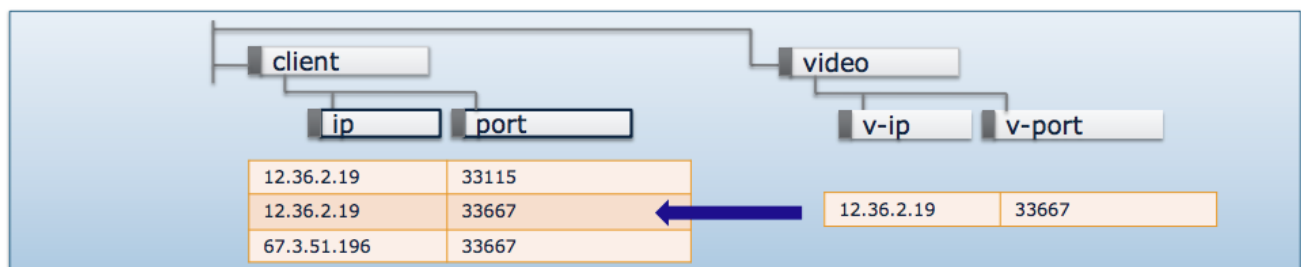
Without Deref

```
leaf my-ip {
  type leafref {
    path "/server/ip";
  }
}
leaf my-port {
  type leafref {
    path "/server[ip = current()]/../my-ip/port";
  }
}
```

After Deref

```
leaf my-ip {
  type leafref {
    path "/server/ip";
  }
}
leaf my-port {
  type leafref {
    path "deref(..my-ip)/../port";
  }
}
```

2.7.3 Other Deref() Examples



```
container video {
  leaf v-ip {
    type leafref {
      path "/client/ip";
    }
  }
  leaf v-port {
    type leafref {
      path "/client[ip=current()]/../v-ip/port";
    }
  }
}
```

Without Deref

```
container video {
```

```

leaf v-ip {
    type leafref {
        path "/client/ip";
    }
}
leaf v-port {
    type leafref {
        path
            "/client[ip=current()/../v-ip]/port";
    }
}
leaf v-stream {
    type leafref {
        path
            "/client[ip=current()/../v-ip][port=current()/../v-port]/stream";
    }
}
}
}

```

With Deref

```

container video {
    leaf v-ip {
        type leafref {
            path "/client/ip";
        }
    }
    leaf v-port {
        type leafref {
            path "deref(..v-ip)/../port";
        }
    }
    leaf v-stream {
        type leafref {
            path "deref(..v-port)/../stream";
        }
    }
}
}

```

I noticed in **xpath.trace** that the following leafref iterates through every GigabitEthernet interface on every device in the CDB to constrain the "must" down to just GigabitEthernet interfaces on the current device. This seems very inefficient and introduces quite a bit of delay (several seconds) while leafref values are being constrained to the current device. Is there a better way to do this so that the device is selected prior to the interface search?

```

list switch-interfaces-GigabitEthernet {
    key switch-interface-id;
    when "../switch-interface-type = 'GigabitEthernet'";
    leaf switch-interface-id {
        tailf:info "The switch interface identifier";
        type leafref {
            path
                "/ncs:devices/ncs:device/ncs:config/ios:interface/ios:GigabitEthernet/ios:name";
        }
        must "current()=deref(deref(current()/../../switch-
name))/../ncs:config/ios:interface/ios:GigabitEthernet/ios:name;
// ../../switch-name is a leafref to "/net-topo:net-topo/net-
topo:switch/net-topo:switch-name"
// net-topo:switch-name is a leafref to "/ncs:devices/ncs:device/ncs:name"
    }
}

```

```
}
```

Solution:

```
/ncs:devices/ncs:device[ncs:name=../../../../switch-  
name]/ncs:config/ios:interface/ios:GigabitEthernet/ios:name
```

2.7.4 Understanding Path Expansions

It is always better to name your leafs and variables different to the system bases names and variables ensuring that their is no confusion when they are used in long XPATHS. For example the use of device or devices in naming your leafs can cause a lot of confusion with the native systems `ncs:device` and `ncs:devices`

Notice in the below example how the `../` and `../../../../` expressions expand.

- For example in the container `ios` , the line `../var1_router_name` goes up one level to the leaf `var1_router_name` .
- Also , in the leaf `intf-number` notice how the `../../../../` goes two levels up just like a unix directory to `var1_router_name`
- Finally notice the expansion of `current()` , notice how it starts from the top level hierarchy of `/ncs:service/learning_deref/.....`

```
module learning_deref {  
  namespace "http://com/example/learning_deref";  
  prefix learning_deref;  
  
  import ietf-inet-types { prefix inet; }  
  import tailf-ncs { prefix ncs; }  
  import tailf-common { prefix tailf; }  
  import tailf-ned-cisco-ios { prefix ios;}  
  
  augment "/ncs:services" // AUGMENT is used to add to another data model , OR  
  augment it. So in the example we are adding/augmenting the learning_deref to  
  it.  
  {  
    list learning_deref  
    {  
      key "name";  
      uses ncs:service-data;  
      ncs:servicepoint "learning_deref";  
  
      leaf name  
      {  
        mandatory true;  
        type string;  
      }  
  
      list link  
      {  
        min-elements 2;  
        max-elements 2;  
        key "var1_router_name";  
        leaf var1_router_name  
        {  
          mandatory true;  
          type leafref  
          {  
            path "/ncs:devices/ncs:device/ncs:name"; // This path points to  
a list of routers , not a sepcific router  
          }  
        }  
      }  
    }  
  }  
}
```

```

    } //routername

    container ios
    {
        // name=current()/../var1_router_name = PE11
        // ncs:name=current()/../var1_router_name EXPANDS to
        // ncs:name=/ncs:services/learning_deref/link/var1_router_name
        when
        "/ncs:devices/ncs:device[ncs:name=current()/../var1_router_name]/ncs:device-
type/ncs:cli/ncs:ned-id='ios-id:cisco-ios'"
        {

            //tailf:dependency "../device";
            //tailf:dependency "/ncs:devices/ncs:device/ncs:device-type";
        }

        leaf intf-number
        {
            mandatory true;
            type leafref
            {
                path
                "deref(..../var1_router_name)/../ncs:config/ios:interface/ios:GigabitEthernet/i
os:name";
            }
        } //intf-number
    } //ios
}
}
}
}

```

2.7.5 More on leafrefs

A leafref is used to model relationships in the data model, as described in [Section 3.10.1, “Modeling Relationships”](#). In the simplest case, the leafref is a single leaf that references a single key in a list:

```

list host {
    key "name";
    leaf name {
        type string;
    }
    ...
}

leaf host-ref {
    type leafref {
        path "../host/name";
    }
}

```

But sometimes a list has more than one key, or we need to refer to a list entry within another list. Consider this example:

```

list host {
    key "name";
    leaf name {
        type string;
    }
}

```

← the name MUST be unique for every element in the list

```

list server {
    key "ip port"; ← the couple ip+port must be unique inside the list
    leaf ip {
        type inet:ip-address;
    }
    leaf port {
        type inet:port-number;
    }
    ...
}

```

This 'nested' list is something like the following:

```

host[0][0][ip1, port1]
host[0][1][ip1, port1]
host[0][2][ip1, port1]
host[0][3][ip1, port1]

host[1][0][ip1, port1]
host[1][1][ip1, port1]
host[1][2][ip1, port1]

```

If we want to refer to a specific server on a host, we must provide **three values**; the host name, the server ip and the server port. Using **leafrefs**, we can accomplish this by using three connected leaves:

```

# this is just a leaf, that must be chosen among the list of hosts, thus the
# reference is /host/name/. Beware that names are unique.

```

```

leaf server-host {
    type leafref {
        path "/host/name";
    }
}

```

```

# this is a list of ip addresses, but they can't be chosen randomly: they MUST
# BE
# the ip addresses of the above server-hosts. Since server-ip is a brother
# hierarchically speaking of the server-host, the meaning is:
#
# consider the current leaf, go back to the father and down to server-host
# the name must be equal to that of the server-host

```

```

leaf server-ip {
    type leafref {
        path "/host[name=current()../server-host]/server/ip";
    }
}

leaf server-port {
    type leafref {
        path "/host[name=current()../server-host]"
        + "/server[ip=current()../server-ip]../port";
    }
}

```

The path specification for `server-ip` means the ip address of the server under the host with same name as specified in `server-host`.

The path specification for `server-port` means the port number of the server with the same ip as specified in `server-ip`, under the host with same name as specified in `server-host`.

This syntax quickly gets awkward and error prone. ConfD supports a shorthand syntax, by introducing an XPath function `deref()` (see [the section called “XPATH FUNCTIONS”](#)). Technically, this function follows a leafref value, and returns all nodes that the leafref refer to (typically just one). The example above can be written like this:

```
leaf server-host {
    type leafref {
        path "/host/name";
    }
}

leaf server-ip {
    type leafref {
        path "deref(..server-host)/../server/ip";
    }
}

leaf server-port {
    type leafref {
        path "deref(..server-ip)/../port";
    }
}
```

Note that using the `deref` function is syntactic sugar for the basic syntax. The translation between the two formats is trivial. Also note that `deref()` is an extension to YANG, and third party tools might not understand this syntax. In order to make sure that only plain YANG constructs are used in a module, the parameter `--strict-yang` can be given to **confdc -c**.

2.8 MUST Statement

An Example of a `must` statement below . There is a `access-timeout` and a `retry-timer` .

In the below example we are setting by `must` that the value of `retry-time` should be **less** that the `access-timeout`. In the past we had to simply put this in the code or programming level , but here in this example we can have the same defined at the data model level. **The `current()` function (from XPATH) in the code below refers to the value of the current node** (which is `retry-timer`).

Next The path `../` means that we go up one level to `timeout` and then reach `access-path` in the tree.

Restricts valid values by Xpath 1.0 expression

```
container timeout {  
  leaf access-timeout {  
    description "Maximum time without server response";  
    units seconds;  
    mandatory true;  
    type uint32;  
  }  
  leaf retry-timer {  
    description "Period to retry operation";  
    units seconds;  
    type uint32;  
    must "current() < ../access-timeout" {  
      error-app-tag retry-timer-invalid;  
      error-message "The retry timer must be "  
        + "less than the access timeout";  
    }  
  }  
}
```

The above constraint will be validated and enforced. As a best practice you should add comment near `must` to describe what it is doing. Another example for Cisco NSO to restrict the devices to be chosen among a couple of groups:

```
leaf-list device {  
  type leafref {  
    path "/ncs:devices/ncs:device/ncs:name";  
  }  
  must "/ncs:devices/ncs:device-group[ncs:name='g1']/ncs:device-name[. =  
current()]"  
    + "or /ncs:devices/ncs:device-group[ncs:name='g2']/ncs:device-name[. =  
current()]" ;  
}
```

2.9 YANG keywords

2.9.1 Unique

Ensure uniqueness of values.

2.9.2 Range

Restricts the range

2.9.3 Error-message

Define the error for the model

In the example below we ensure the the value of `vpn-id` is unique . The `range` restricts the value and `error-message` is for the error for incorrect input.

```
list l3mplsvpn-ce-config  
{  
  tailf:info "Used to Configure the CE Side of the MPLSL3VPN";  
  key "vpn-name";  
  unique vpn-id;  
  
  leaf vpn-id  
  {  
    tailf:info "Name of the VPN ID";  
    type uint32  
    {  
      range 1..10;  
    }  
  }  
}
```



```

    error-message "Invalid VPN ID"
  }
}
}

```

2.9.4 Template tag operations (merge, replace, create, ncreate, delete)

Templates allow for defining different behaviour when applying the template. This is accomplished by setting tags, as an attribute. Existing tags are: `merge`, `replace`, `delete`, `create` or `ncreate` on relevant nodes in the template. A tag is inherited to its sub-nodes until a new tag is introduced.

- *merge*: Merge with a node if it exists, otherwise create the node. This is the default operation if no operation is explicitly set.
- ...
- `<config tags="merge">`
- `<interface xmlns="urn:ios">`
- ...
- *replace*: Replace a node if it exists, otherwise create the node.
- ...
- `<GigabitEthernet tags="replace">`
- `<name>{link/interface-number}</name>`
- `<description tags="merge">Link to PE</description>`
- ...
- *create*: Creates a node. The node can not already exist. An error is raised if the node exists.
- ...
- `<GigabitEthernet tags="create">`
- `<name>{link/interface-number}</name>`
- `<description tags="merge">Link to PE</description>`
- ...
- *ncreate*: Merge with a node if it exists. If it does not exist, it will *not* be created.
- ...
- `<GigabitEthernet tags="ncreate">`
- `<name>{link/interface-number}</name>`
- `<description tags="merge">Link to PE</description>`
- ...
- *delete*: Delete the node.
- ...
- `<GigabitEthernet tags="delete">`
- `<name>{link/interface-number}</name>`
- `<description tags="merge">Link to PE</description>`
- ...

2.9.5 count

To count all occurrences of a xpath:

```
augment "/ncs:services"
{
  list l3mplsvpn
  {
    tailf:info "Layer-3 MPLS VPN Service";
    key "vpn-name";
    unique interface;

    leaf vpn-name
    {..}

    leaf device
    {..}

    leaf interface
    {
      tailf:info "Customer Facing Interface";
      type string;

      # What the below expression expands to :
      # "In NO other vpn configuration (vpn-name !=) the current device AND its
      # current interface should be configured" which results in the expression to be
      # zero.
      must "count(..../l3mplsvpn[vpn-name != current()]/../vpn-name][device =
current()]/../device][interface=current()]) = 0"

      # another way to do the same would be the following:
      #// must "count(..../l3mplsvpn[vpn-name != current()]/../vpn-name]" + "[device =
# current()]/../device][interface=current()]) = 0"
      # Notice the + in the above command , it is nothing but to ensure
      # continuation of the entire path . It is not doing any arithmetic SUMMATION

      {
        error-message "Interface is already used for another link.";
      }
    }
  }
}
```

Example Code execution

```
# services l3mplsvpn vpn1 device SP1 interface 0/1
# services l3mplsvpn vpn2 device SP1 interface 0/2

admin@ncs (config-l3mplsvpn-vpn2) # commit dry-run | debug xpath

# Statement to be validated
Evaluating XPath for: /services/l3mplsvpn:l3mplsvpn[vpn-name='vpn2']/interface:
count(..../l3mplsvpn[vpn-name != current()]/../vpn-name][device =
current()]/../device][interface=current()]) = 0

get_next(/ncs:services/l3mplsvpn) = {vpn1}
get_elem("/ncs:services/l3mplsvpn{vpn1}/device") = SP1
get_elem("/ncs:services/l3mplsvpn{vpn2}/device") = SP1
get_elem("/ncs:services/l3mplsvpn{vpn1}/interface") = 0/1
get_elem("/ncs:services/l3mplsvpn{vpn2}/interface") = 0/2
get_next(/ncs:services/l3mplsvpn{vpn1}) = {vpn2}
```

```

get_next(/ncs:services/l3mplsvpn{vpn2}) = false
2018-05-25T18:40:13.758 XPath for: /services/l3mplsvpn:l3mplsvpn[vpn-
name='vpn2']/interface returns true
2018-05-25T18:40:13.759
Evaluating XPath for: /services/l3mplsvpn:l3mplsvpn[vpn-name='vpn2']/device:
  /ncs:devices/ncs:device/ncs:name
get_elem("/ncs:services/l3mplsvpn{vpn2}/device") = SP1
exists("/ncs:devices/device{SP1}") = true
get_elem("/ncs:services/l3mplsvpn{vpn2}/device") = SP1
2018-05-25T18:40:13.763 XPath for: /services/l3mplsvpn:l3mplsvpn[vpn-
name='vpn2']/device returns true
2018-05-25T18:40:13.766
Evaluating XPath for: /services/l3mplsvpn:l3mplsvpn[vpn-name='vpn1']/interface:
  count(..../l3mplsvpn[vpn-name != current()]/../vpn-name][device =
current()]/../device][interface=current()]) = 0
get_next(/ncs:services/l3mplsvpn) = {vpn1}
get_next(/ncs:services/l3mplsvpn{vpn1}) = {vpn2}
get_elem("/ncs:services/l3mplsvpn{vpn2}/device") = SP1
get_elem("/ncs:services/l3mplsvpn{vpn1}/device") = SP1
get_elem("/ncs:services/l3mplsvpn{vpn2}/interface") = 0/2
get_elem("/ncs:services/l3mplsvpn{vpn1}/interface") = 0/1
get_next(/ncs:services/l3mplsvpn{vpn2}) = false
2018-05-25T18:40:13.768 XPath for: /services/l3mplsvpn:l3mplsvpn[vpn-
name='vpn1']/interface returns true
cli {
  local-node {
    data services {
      +   l3mplsvpn vpn2 {
      +   }
    }
  }
}

```

2.9.6 pattern

To define a pattern. The below construct define an IP Address patterns and also the error which should be returned if its now followed.

```

leaf pe-ip {
  tailf:info "PE Interface IP Address";
  mandatory false;
  type inet:ipv4-address {
    pattern "172\\.( [1][6-9] | [2][0-9] | 3[0-1] )\\.\\. *"
    {
      error-message
        "Invalid IP address. IP address should be in the 172.16.0.0/12 range.";
    }
  }
}

```

2.9.7 starts-with

To define a pattern. In the below example on the devices whose name starts with PE will be displayed in the options or be validated for.

```

leaf device {
  tailf:info "PE Router";
  mandatory true;
  type leafref {
    path "/ncs:devices/ncs:device/ncs:name";
  }
  must "starts-with(current(),'PE') {

```

```

    error-message "Only PE devices can be selected.";
  }
}

```

2.9.8 min-elements

Statement to define the minimum number of entries in a list.

```

list link
{
  tailf:info "PE-CE Attachment Point";
  key "link-name";
  unique "link-id";
  unique "device interface";
  min-elements 1;
}

```

2.9.9 When

To control a leaf visibility. In the example below the leaf `ce-ip` will only be visible when the `routing-protocol` is set to `bgp`:

```

leaf ce-ip
{
  tailf:info "CE Interface IP Address";
  when "../routing-protocol='bgp'";           ← previous leaf config
  mandatory false;
  type inet:ipv4-address
  {
    pattern "172\\.([1][6-9]|[2][0-9]|3[0-1])\\.\\.*"
    {
      error-message
        "Invalid IP address. IP address should be in the 172.16.0.0/12 range.";
    }
  }
}

```

2.9.10 tailf:hidden

... to hide the leaf from the northbound interfaces. This will remove the leaf from appearing in CLI / Web UI . This is a precautionary step since we want to apply it programmatically.

```

augment "/ncs:services"
{
  leaf l3mplsvpn-id-cnt
  {
    description "Provides a unique 32-bit number used as VPN instance
identifier";
    tailf:hidden "Counter";
    type uint32;
    default "1";
  }
}

```

2.9.11 Leafrefs with conditional path references

```

grouping interface-leafrefs-grouping {
  choice interface-type {
    leaf ios-GigabitEthernet{
      when "derived-from(/ncs:devices/ncs:device" +
        "[ncs:name=current()/../device]/ncs:device-type" +
        "/ncs:cli/ncs:ned-id, 'ios-ned-id:cisco-ios-cli')";
    }
  }
}

```

```

    tailf:info "Cisco IOS Gigabit Ethernet interface";
    type leafref {
        path "deref(..../device)/../ncs:config" +
            "/ios:interface/ios:GigabitEthernet/ios:name";
    }
}
leaf ios-xr-GigabitEthernet {
    when "derived-from(/ncs:devices/ncs:device" +
        "[ncs:name=current()/../device]/ncs:device-type" +
        "/ncs:cli/ncs:ned-id, 'iosxr-ned-id:cisco-iosxr-cli')";
    tailf:info "Cisco IOS-XR Gigabit Ethernet interface";
    type leafref {
        path "deref(..../device)/../ncs:config" +
            "/cisco-ios-xr:interface" +
            "/cisco-ios-xr:GigabitEthernet/cisco-ios-xr:id";
    }
}
leaf ios-xr-TenGigE {
    when "derived-from(/ncs:devices/ncs:device" +
        "[ncs:name=current()/../device]/ncs:device-type" +
        "/ncs:cli/ncs:ned-id, 'iosxr-ned-id:cisco-iosxr-cli')";
    tailf:info "Cisco IOS-XR 10 Gigabit Ethernet interface";
    type leafref {
        path "deref(..../device)/../ncs:config" +
            "/cisco-ios-xr:interface" +
            "/cisco-ios-xr:TenGigE/cisco-ios-xr:id";
    }
}
leaf junos-interface {
    when "derived-from(/ncs:devices/ncs:device" +
        "[ncs:name=current()/../device]/ncs:device-type" +
        "/ncs:netconf/ncs:ned-id, 'junos-ned-id:juniper-junos-nc')";
    tailf:info "Junos interface";
    type leafref {
        path "deref(..../device)/../ncs:config/junos:configuration" +
            "/junos:interfaces/junos:interface/junos:name";
    }
}
leaf alu-interface {
    when "derived-from(/ncs:devices/ncs:device" +
        "[ncs:name=current()/../device]/ncs:device-type" +
        "/ncs:cli/ncs:ned-id, 'alu-ned-id:alu-sr-cli')";
    tailf:info "ALU interface";
    type leafref {
        path "deref(..../device)/../ncs:config/alu:port/alu:port-id";
    }
}
}

```

3 YANG

NOTE: this chapter has been taken from a second reference, written by **Matt Albrecht**.

<https://ultraconfig.com.au/blog/learn-yang-full-tutorial-for-beginners/>

In short, YANG is a language used to describe data models of network devices. The language is maintained by [NETMOD](#) - an IETF working group. The latest version of YANG is 1.1, and the full specification of the language is documented in [RFC 7950](#). With that said, what is a data model of a network device? To answer that, let's imagine a hypothetical scenario where your friend asks you what IP interface attributes can be configured on a specific router. You might say:

"Well, to configure an interface on this router, you need to supply: an interface name, an IP address, and a subnet mask. You also need to enable the interface - the router will keep the interface disabled if you don't."

Now, as simple as this response sounds, what we just did was describe a data model for an IP interface. A YANG model will do the same but uses strict syntax rules to make the model standardized and easy to process with computers.

3.1 It's all about describing the data structure

Note that in our response to our friend, we didn't provide any specific values for an interface - we only described the data structure. This is an important distinction to note as it will make the bigger picture much easier to understand when we later talk about [NETCONF](#).

3.2 Let's learn the YANG syntax

Let's redo our IP interface example, but this time we'll describe the model using YANG rather than plain English.

I'll be working on Ubuntu 18.04 for this demo. Before we write our YANG model, let's install an excellent python utility called "[pyang](#)", which will allow us to interact with our model. We'll also need two dependencies for the utility to work.

```
# Install pyang dependencies
sudo apt-get install -y xsltproc libxml2-utils

# Clone the pyang repo
git clone https://github.com/mbj4668/pyang

# Install pyang
cd pyang
sudo python setup.py install
```

With that done, let's create a new file for defining our YANG model. The file extension must be "yang". I'll name my file:

```
ultraconfig-interfaces.yang
```

We'll now open the file in a text editor.

3.3 Writing our first YANG statements

At the top of the file, we'll add a "*module*" statement followed by the name of our module and a braces block. The module name must match the name of our file.

```
module ultraconfig-interfaces {

}
```

All of the content we now add to the file will go inside the module braces. Next, let's add header information, meta-information, and revision history to our module. You'll notice that strings are terminated

by semicolons rather than newline characters - this allows long strings to be written over multiple lines preserving readability.

```
yang-version 1.1;

namespace
  "http://ultraconfig.com.au/ns/yang/ultraconfig-interfaces";

prefix if;

organization
  "Ultra Config Pty Ltd";

contact
  "Support: <https://ultraconfig.com.au/contact/>";

description
  "This YANG module has been created for the purpose of a tutorial.
  It defines the model for a basic ethernet interface";

revision "2020-01-03" {
  description
    "Initial Revision";
  reference
    "Learn YANG - Full Tutorial for Beginners";
}
```

The labels yang-version, namespace, organization, etc are known as "statements" in YANG terminology. Let's go over the function of each statement below.

- **yang-version** - Identifies the YANG language specification that the module will conform to. We'll ensure our module conforms to YANG 1.1 which is defined in RFC 7950.
- **namespace** - This is an XML namespace that must be unique for the module. We used a URL but you can use a URN, URI or any other unique identifier here. The namespace specified here must match the namespace on any XML objects which conform to our YANG model.
- **prefix** - A short and unique string to identify our module. This prefix may be used in other YANG modules to import definitions contained in this one.
- **organization** - A string identifying the entity responsible for the module.
- **contact** - Contact details for the entity responsible for the module.
- **description** - A description of the module.
- **revision** - Used for version control. Each edit to a YANG module will add a new revision statement detailing the changes in sub-statements.

With the header & meta information now out of the way, we can move on to our module definition.

3.4 How to create a custom data type

The YANG language includes a set of built-in data types. The language also allows, however, the ability for developers to define custom data types.

We'll do that now for our IPV4 address and subnet mask. Both of these attributes should conform to the "dotted-quad" data type defined below. We'll add this definition to our YANG module.

```
typedef dotted-quad {
  type string {
    pattern
      '(([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\.){3}'
      + '([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])';
  }
}
```

```

description
  "Four octets written as decimal numbers and
   separated with the '.' (full stop) character.";
}

```

You may be thinking, why not just use a string instead of defining a custom data type? While this would work, it would certainly be a bad practice. In all programming, it is always best to add constraints on the lowest layers to avoid a later reliance on higher layers for error checking.

Our "dotted-quad" definition is quite simple once you get the idea. Our definition says:
"Define a new type called dotted-quad. A value will conform to this data type if it is a string and matches the regular expression defined in the pattern statement."

3.5 Understanding configuration data and state data

Moving on, let's now add a "container" for our interfaces.

```

container interfaces {
}

```

Our "interfaces" container will hold the child nodes for our configuration data and state data. Let's distinguish between these two data types below.

- **Configuration Data** - These are read/write configuration fields. For our interface example, this would be the interface name, IP address, subnet mask, and admin enabled/disabled.
- **State Data** - These are read-only operational data fields. For our interface example, this could include a packet counter and an operational state (physically up or down).

Again, remember that YANG modules will only define the structure of configuration and state data. It will not contain an instantiated value of the data.

3.6 Adding our configuration data

We'll now add a list to our container for defining our interface configuration data.

```

list interface {
  key "name";
  leaf name {
    type string;
    mandatory "true";
    description
      "Interface name. Example value: GigabitEthernet 0/0/0";
  }
  leaf address {
    type dotted-quad;
    mandatory "true";
    description
      "Interface IP address. Example value: 10.10.10.1";
  }
  leaf subnet-mask {
    type dotted-quad;
    mandatory "true";
    description
      "Interface subnet mask. Example value: 255.255.255.0";
  }
  leaf enabled {
    type boolean;
    default "false";
  }
}

```



```

        description
        "Enable or disable the interface. Example value: true";
    }
}

```

Our interface configuration data is quite readable. We have four leaf nodes which define the attributes of an interface. These are labelled with the identifiers "name", "address", "subnet-mask" and "enabled".

Three of the leaf nodes are marked as mandatory. The "enabled" node is optional and will have a default value of "false" if not specified.

You'll also notice that the data type of the "address" and "subnet-mask" is "dotted-quad". This matches the identifier of our earlier definition.

3.7 Adding our state data

Let's now add our state data to the interfaces container.

```

list interface-state {
    config false;
    key "name";
    leaf name {
        type string;
        description
        "Interface name. Example value: GigabitEthernet 0/0/0";
    }
    leaf oper-status {
        type enumeration {
            enum up;
            enum down;
        }
        mandatory "true";
        description
        "Describes whether the interface is physically up or down";
    }
}

```

Looking at the state data you'll notice that one key difference is the "config" statement which is set to false. This indicates that the child nodes belonging to the list are read-only.

You'll also notice the enumeration data type that we hadn't used before. This is another built-in type, which allows us to restrict the valid values for the "oper-status" node to a finite set; in our case, the operational status will only ever be up or down.

That concludes the construction of our YANG module for an interface. We may now proceed to interact with the module using pyang.

3.8 How to validate a YANG module

The first cool trick to learn with pyang is how to do basic validation. Run the command below to ensure the YANG module is syntactically correct.

```
pyang ultraconfig-interfaces.yang
```

If all is well the output should come out clean. If there are any syntax errors in the module the command will print a message explaining the issue.

To demonstrate this, let's introduce a typographical error by replacing the type "enumeration" with the word spelled incorrectly, "inumeration". Running the validation command again will highlight the error.

```
ultraconfig-interfaces.yang:69: error: type "inumeration" not found in module
"ultraconfig-interfaces"
```

Alright, now that we have confirmed that, let's fix our spelling mistake.

3.9 How to view the schema tree

The next great trick to learn is how to view the schema tree of your YANG module. The schema tree is a summarised visual form of our YANG data model. You can view the tree by adding the format specifier option to your command.

```
pyang -f tree ultraconfig-interfaces.yang
```

Here's how the output will look for our example.

```
module: ultraconfig-interfaces
  +--rw interfaces
    +--rw interface* [name]
      | +--rw name      string
      | +--rw address   dotted-quad
      | +--rw subnet-mask dotted-quad
      | +--rw enabled?   boolean
    +--ro interface-state* [name]
      +--ro name      string
      +--ro oper-status enumeration
```

The "rw" acronym is short for read-write. The "ro" acronym is short for read-only. The rest of the tree is quite self-explanatory. The question mark next to the "enabled" node indicates that this object is optional.

3.10 Introducing "yang2dsdl"

Let's now move on to another awesome utility called "yang2dsdl". The program is bundled into the pyang project and should have been added to your system path upon installation.

The utility is short for "YANG to DSDL (Document Schema Definition Languages)". The primary purpose of this tool is to convert YANG modules to DSDL schemas but we can also use it to validate instances of data to ensure they conform to a YANG module.

3.11 How to validate a data instance

This will be easier to understand with an example. Below is an XML instance of the YANG module which we defined today.

```
<data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <interfaces xmlns="http://ultraconfig.com.au/ns/yang/ultraconfig-interfaces">
    <interface>
      <name>GigabitEthernet 0/0/0</name>
      <address>10.10.10.1</address>
      <subnet-mask>255.255.255.0</subnet-mask>
    </interface>
    <interface>
      <name>GigabitEthernet 0/0/1</name>
      <address>192.168.1.1</address>
      <subnet-mask>255.255.255.0</subnet-mask>
    </interface>
  </interfaces>
</data>
```

If you are familiar with NETCONF this XML data object will look familiar to you. This is what a payload looks like in a NETCONF request to edit the configuration of a network device.

We'll now use yang2dsdl to ensure the XML object is valid. Save the XML object to a file. We called our file:

```
data.xml
```

We can now run the command below which will generate the DSDL schemas of our YANG module and validate our data instance.

```
yang2dsdl -v data.xml ultraconfig-interfaces.yang
```

The program output is shown below.

```
== Generating RELAX NG schema './ultraconfig-interfaces-data.rng'
Done.

== Generating Schematron schema './ultraconfig-interfaces-data.sch'
Done.

== Generating DSRL schema './ultraconfig-interfaces-data.dsrl'
Done.

== Validating grammar and datatypes ...
data.xml validates

== Adding default values... done.

== Validating semantic constraints ...
No errors found.
```

We can see that no errors were found.

We'll now put an invalid IP address into our XML file and rerun the validation. You'll get an error like the one below.

```
== Validating grammar and datatypes ...
data.xml:5: element address: Relax-NG validity error : Error validating datatype
string
data.xml:5: element address: Relax-NG validity error : Element address failed to
validate content
data.xml:3: element interface: Relax-NG validity error : Invalid sequence in
interleave
data.xml:3: element interface: Relax-NG validity error : Element interface
failed to validate content
Relax-NG validity error : Extra element interface in interleave
data.xml:3: element interface: Relax-NG validity error : Element interfaces
failed to validate content
data.xml fails to validate
```

There we go, that's everything essential you need to learn about YANG!

With a solid grasp of the concepts of the YANG language, you'll find that automation solutions built on top of the NETCONF protocol become demystified.

NETCONF is the protocol for sending and receiving configuration data and state data of network devices.

And YANG is the language that describes the structure of this data.

I hope you enjoyed this tutorial as much as I enjoyed making it. You can download the full YANG module we developed today using the link below.

```
[nso/src/ncs/yang/tailf-common.yang]
type tailf:ip-address-and-prefix-length;
. . .
typedef ip-address-and-prefix-length {
    type union {
        type tailf:ipv4-address-and-prefix-length;
        type tailf:ipv6-address-and-prefix-length;
    }
}
. . .
typedef ipv4-address-and-prefix-length {
```

```

type string {
  pattern
    '([([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\.]{3})'
  + '([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])'
  + '/([([0-9])|([1-2][0-9])|(3[0-2]))';
}
description
  "The ipv4-address-and-prefix-length type represents a combination
  of an IPv4 address and a prefix length. The prefix length is given
  by the number following the slash character and must be less than
  or equal to 32.";
}

```

3.12 Possible YANG keywords

Follows hereafter an explanation of the different possible declarations in a YANG model:

- **augment:** Adds new schema nodes to a previously defined schema node.
- **container:** An interior data node that exists in at most one instance in the data tree. A container has no value, but rather a set of child nodes.
- **data model:** A data model describes how data is represented and accessed.
- **data node:** A node in the schema tree that can be instantiated in a data tree. One of container, leaf, leaf-list, list, and anyxml.
- **data tree:** The instantiated tree of configuration and state data on a device.
- **derived type:** A type that is derived from a built-in type (such as uint32), or another derived type.
- **device deviation:** A failure of the device to implement the module faithfully.
- **grouping:** A reusable set of schema nodes, which may be used locally in the module, in modules that include it, and by other modules that import from it. The grouping statement is not a data definition statement and, as such, does not define any nodes in the schema tree.
- **identifier:** Used to identify different kinds of YANG items by name.
- **leaf:** A data node that exists in at most one instance in the data tree. A leaf has a value but no child nodes.
- **leaf-list:** Like the leaf node but defines a set of uniquely identifiable nodes rather than a single node. Each node has a value but no child nodes.
- **list:** An interior data node that may exist in multiple instances in the data tree. A list has no value, but rather a set of child nodes.
- **module:** A YANG module defines a hierarchy of nodes that can be used for NETCONF-based operations. With its definitions and the definitions it imports or includes from elsewhere, a module is self-contained and “compilable”.
- **state data:** The additional data on a system that is not configuration data such as read-only status information and collected statistics [RFC4741].

3.13 YANG identities and affiliations

Let's try to understand YANG a bit better by using an example. Imagine for a second you want to write the ultimate Star Wars framework; one framework to rule them all (wrong movie). To begin with, we want to start by being able to add individuals from the Star Wars universe into a list. Those individuals will have the following information:

- **name**, everybody has a name, even if it's a model number. Nothing we really have to do here. A name is just a string.
- **age**, which we will be limit to 2000, because who wants to live forever (wrong movie again). To support the age we are going to create a new type that we will use to enforce the correctness of the data:

```
typedef age {
    type uint16 {
        range 1..2000;
    }
}
```

- **affiliation**, you are either with the empire or against it. For the affiliation we are going to create an identity that will unequivocally identify each possible affiliation:

```
identity AFFILIATION {
    description "To which group someone belongs to";
}
```

```
identity EMPIRE {
    base AFFILIATION;
    description "Affiliated to the empire";
}
```

```
identity REBEL_ALLIANCE {
    base AFFILIATION;
    description "Affiliated to the rebel alliance";
}
```

Now that we have set the foundation, let's create the model:

```
// module name
module napalm-star-wars {

    // boilerplate
    yang-version "1";
    namespace "https://napalm-yang.readthedocs.io/napalm-star-wars";

    prefix "napalm-star-wars";

    // identity to unequivocally identify the faction an individual belongs to
    identity AFFILIATION {
        description "To which group someone belongs to";
    }

    identity EMPIRE {
        base AFFILIATION;
        description "Affiliated to the empire";
    }

    identity REBEL_ALLIANCE {
        base AFFILIATION;
        description "Affiliated to the rebel alliance";
    }

    // new type to enforce correctness of the data
    typedef age {
        type uint16 {
            range 1..2000;
        }
    }

    // this grouping will all the personal data we will assing to individuals
    grouping personal-data {
        leaf name {
            type string;
        }
    }
}
```

```

    leaf age {
        type age;
    }
    leaf affiliation {
        type identityref {
            base napalm-star-wars:AFFILIATION;
        }
    }
}

// this is the root object defined by the model
container universe {
    list individual {
        // identify each individual by using the name as key
        key "name";

        // each individual will have the elements defined in the grouping
        // uses personal-data; ← refers to the previous
    }
}

```

```

$ pyang -f tree napalm-star-wars.yang
module: napalm-star-wars
  +--rw universe
    +--rw individual* [name]
      +--rw name          string
      +--rw age?          age
      +--rw affiliation?  Identityref

```

Make sense, it's what we were expecting. Now, let's make something useful with it and build python code from the model. We can use `pyangbind` for that (the lib `napalm-yang` uses under the hoods):

```

$ export PYBINDPLUGIN=`/usr/bin/env python -c \
    'import pyangbind; import os; print "%s/plugin" %
os.path.dirname(pyangbind.__file__)'`
$ pyang --plugindir $PYBINDPLUGIN -f pybind napalm-star-wars.yang >
napalm_star_wars.py

```

Now we have some python code we can put to test:

```

>>> import napalm_star_wars
>>>
>>> sw = napalm_star_wars.napalm_star_wars()
>>>
>>> obi = sw.universe.individual.add("Obi-Wan Kenobi")
>>> obi.affiliation = "REBEL_ALLIANCE"
>>> obi.age = 57
>>>
>>> luke = sw.universe.individual.add("Luke Skywalker")
>>> luke.affiliation = "REBEL_ALLIANCE"
>>> luke.age = 19
>>>
>>> darth = sw.universe.individual.add("Darth Vader")
>>> darth.affiliation = "EMPIRE"
>>> darth.age = 42
>>>
>>> yoda = sw.universe.individual.add("Yoda")
>>> yoda.affiliation = "REBEL_ALLIANCE"

```

```

>>> yoda.age = 896
>>>
>>> import json
>>> print(json.dumps(sw.get(), indent=4))
{
  "universe": {
    "individual": {
      "Obi-Wan Kenobi": {
        "affiliation": "REBEL_ALLIANCE",
        "age": 57,
        "name": "Obi-Wan Kenobi"
      },
      "Luke Skywalker": {
        "affiliation": "REBEL_ALLIANCE",
        "age": 19,
        "name": "Luke Skywalker"
      },
      "Darth Vader": {
        "affiliation": "EMPIRE",
        "age": 42,
        "name": "Darth Vader"
      },
      "Yoda": {
        "affiliation": "REBEL_ALLIANCE",
        "age": 896,
        "name": "Yoda"
      }
    }
  }
}

```

3.14 YANG modules extensions

So our framework has been a success, so much that people has started adding mods to it. One of those mods adds support for individuals working as mercenaries and it also adds an extra piece of information into the personal data of each individual to indicate if the individual is in active duty or retired. YANG is quite powerful when it comes to extending existing models; you don't really need to fork the project, change the schema or do anything crazy. You just import the old model and add new stuff. So let's see how the extension to our existing model would look like:

```

module napalm-star-wars-extended {

  yang-version "1";
  namespace "https://napalm-yang.readthedocs.io/napalm-star-wars-extended";

  prefix "napalm-star-wars-extended";

  // We import the old model
  import napalm-star-wars { prefix napalm-star-wars; }

  // New identity based off the old AFFILIATION
  identity MERCENARY {
    base napalm-star-wars:AFFILIATION;
    description "Friend for money";
  }

  // This grouping contains the new information we want to attach
  // to the personal data of the old model
  grouping extended-personal-data {
    leaf status {
      type enumeration {

```

```

        enum ACTIVE {
            description "In active duty";
        }
        enum RETIRED {
            description "Enjoying retirement, probably in a house by a
lake";
        }
    }
}

// This is where we tell what part of the old model we want to extend
augment "/napalm-star-wars:universe/napalm-star-wars:individual" {
    uses extended-personal-data;
}
}

```

Easy, right? Beauty is that you can load the extensions if you want and if someone do changes in the original model you will benefit from them as you didn't fork the model. Now let's do the same we did before and see how we can take advantage of the extensions.

The tree representation looks good:

```

$ pyang -f tree napalm-star-wars-extended.yang napalm-star-wars.yang
module: napalm-star-wars
  +--rw universe
    +--rw individual* [name]
      +--rw name                string
      +--rw age?                age
      +--rw affiliation?        identityref
      +--rw napalm-star-wars-extended:status? Enumeration

```

Now let's create some code with the extensions in place:

```

$ pyang --plugindir $PYBINDPLUGIN -f pybind napalm-star-wars-extended.yang napalm-star-
wars.yang > napalm_star_wars_extended.py

```

And use it:

```

>>> import napalm_star_wars_extended
>>>
>>> sw = napalm_star_wars_extended.napalm_star_wars()
>>>
>>> obi = sw.universe.individual.add("Obi-Wan Kenobi")
>>> obi.affiliation = "REBEL_ALLIANCE"
>>> obi.age = 57
>>> obi.status = "RETIRED"
>>>
>>> darth = sw.universe.individual.add("Darth Vader")
>>> darth.affiliation = "EMPIRE"
>>> darth.age = 42
>>> darth.status = "ACTIVE"
>>>
>>> yoda = sw.universe.individual.add("Yoda")
>>> yoda.affiliation = "REBEL_ALLIANCE"
>>> yoda.age = 896
>>> yoda.status = "RETIRED"
>>>
>>> boba = sw.universe.individual.add("Boba Fett")
>>> boba.affiliation = "MERCENARY"
>>> boba.age = 32
>>> boba.status = "ACTIVE"
>>>

```



```

>>> import json
>>> print(json.dumps(sw.get(), indent=4))
{
  "universe": {
    "individual": {
      "Obi-Wan Kenobi": {
        "status": "RETIRED",
        "affiliation": "REBEL_ALLIANCE",
        "age": 57,
        "name": "Obi-Wan Kenobi"
      },
      "Darth Vader": {
        "status": "ACTIVE",
        "affiliation": "EMPIRE",
        "age": 42,
        "name": "Darth Vader"
      },
      "Yoda": {
        "status": "RETIRED",
        "affiliation": "REBEL_ALLIANCE",
        "age": 896,
        "name": "Yoda"
      },
      "Boba Fett": {
        "status": "ACTIVE",
        "affiliation": "MERCENARY",
        "age": 32,
        "name": "Boba Fett"
      }
    }
  }
}

```

Great, we now know if the individual is enjoying a golden retirement or not and we also support an extra faction in our extended version of the framework.

3.15 Another extension for ip addresses

Now let's see how `napalm-yang` actually takes advantage of the extensibility of YANG. If you take a look to the [openconfig-if-ip](#) model you will realize it doesn't support the `secondary` option that some platforms require when configuring multiple IPv4 addresses on the same interface. Well, not a huge deal, we can fix it ourselves:

```

module napalm-if-ip {
  yang-version "1";
  namespace "https://github.com/napalm-automation/napalm-
yang/yang_napalm/interfaces";

  prefix "napalm-ip";
  import openconfig-interfaces { prefix oc-if; }
  import openconfig-vlan { prefix oc-vlan; }
  import openconfig-if-ip { prefix oc-ip; }
  organization "NAPALM Automation";

  contact "napalm-automation@googlegroups.com";
  description "This module defines some augmentations to the interface's IP
model of OC";

  revision "2017-03-17" {
    description
      "First release";
    reference "1.0.0";
  }
}

```

```

    }

    grouping secondary-top {
        description "Add secondary statement";

        leaf secondary {
            type boolean;
            default "false";

            description
                "Most platforms need a secondary statement on when configuring
multiple IPv4
                addresses on the same interfaces";

            reference
                "https://www.cisco.com/c/en/us/td/docs/ios/12_2/ip/configuration/guide/fipr_c/1c
fipadr.html#wp1001012";
        }
    }

    augment "/oc-if:interfaces/oc-if:interface/" +
        "oc-if:subinterfaces/oc-if:subinterface/" +
        "oc-ip:ipv4/oc-ip:addresses/oc-ip:address/" +
        "oc-ip:config" {
        description "Add secondary statement to subinterfaces' IPs";
        uses secondary-top;
    }

    augment "/oc-if:interfaces/oc-if:interface/" +
        "oc-vlan:routed-vlan/" +
        "oc-ip:ipv4/oc-ip:addresses/oc-ip:address/" +
        "oc-ip:config" {
        description "Add secondary statement to routed VLANs' IPs";
        uses secondary-top;
    }
}

```

4 NSO

Basically NSO is an orchestration and automation tool, that can be accessed through a CLI (probably more powerful) or a GUI. Let's say as usual that the CLI is for expert users, while the GUI is for everyone. It has a configurations and nodes database that is kept local, and should be kept in sync with the configuration of the on-field routers, switches and nodes in general. This is the '**sync**' concept: you can check if the configurations are in sync with that of the devices. If someone modifies configuration without using NSO, you can check this out and re-sync the configurations: there is the '**sync-to**' concept (from NSO to the devices ... can be dangerous !!!), and the '**sync-from**' concept (copy the devices configurations locally, for those devices that are out of sync).

```

admin@ncs% run show devices list
NAME ADDRESS DESCRIPTION NED ID ADMIN STATE
-----
alu0 127.0.0.1 - alu-sr unlocked
c0 127.0.0.1 - cisco-ios unlocked
c1 127.0.0.1 - cisco-ios unlocked
fip0 127.0.0.1 - netconf unlocked
j0 127.0.0.1 - netconf unlocked

admin@ncs% request devices sync-from
sync-result {
device alu0
result true

```

```

}
sync-result {
  device c0
  result true
}

```

The NED (Network Element Driver) is something invoked to interact with the final device.

Netsim is an embedded software that emulates a few devices, based on their NED, that allows for tests and simple retrieval of xml configurations.

The devices can be added by simply configuring their name and ip address, and can be organized and divided into different groups. A device can of course belong to more groups.

There are predefined commands to create 'services', that create all the necessary directories and some predefined 'skeleton' files, to be changed and filled as needed.

A **yang** file defines all the variables needed to build the service, you can also configure cross-references to node's configurations, in such a way that the GUI provides a multiple selection choice on the router's existing interfaces, which a useful feature. They can be single elements, lists, structured data composed by multiple 'childs', enumeration elements, some constraints can be embedded to check for data consistency (for example a vlan must be in the range 1-4096).

Input variables can be used to perform configurations through 'xml' files, that are used to perform the REST queries toward the target devices. Configurations are quite straightforward, even though of course you need to start from on field 'native' configurations to retrieve the xml one:

```
show full-configuration devices device device0 | display xml
```

For example a vrf xml file could contain something like the following:

```

<vrf xmlns="urn:ios">
  <definition>
    <name>MGMT-BBIP</name>                ← string variable inside yaml service
file
    <rd>64840:2</rd>
    <route-target>
<export>
  <asn-ip>64840:2</asn-ip>              ← rt value for full-mesh vrf
</export>
<import>
  <asn-ip>64840:2</asn-ip>              ← rt value for full-mesh vrf
</import>
    </route-target>
    <address-family>
<ipv4>
</ipv4>
    </address-family>
    </definition>
</vrf>

```

The above example should be reused replacing the on field values with parameters like the following:

"{/VRF_Name}"

... that have been defined in the yaml service file. Continuing with the example, the xml file becomes like the following:

```

<config-template xmlns="http://tail-f.com/ns/config/1.0"
  servicepoint="VRF-Configuration">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>

```

```

<name>{deref(/Device)/../member} </name>
<config>
  <vrf xmlns="urn:ios">                                ← scope and NED reference
    <definition>
      <name>{/VRF_Name}</name>                             ← parameter
      <rd>{/RD}</rd>                                       ← parameter
      <address-family>
        <ipv4/>
      </address-family>
      <route-target>
        <export>
          <asn-ip>{/RT-export}</asn-ip>                     ← parameter
        </export>
        <import>
          <asn-ip>{/RT-import}</asn-ip>                     ← parameter
        </import>
      </route-target>
    </definition>
  </vrf>
</config>
</device>
</devices>
</config-template>

```

Beware that the above template works ONLY for IOS devices. In case there are also XR devices, NX-OS devices, Juniper devices, OTHERS config context MUST be added for everything to work fine. So **this tool is absolutely not an abstraction one**, it is a multivendor tool, for all devices supporting REST queries, but for EVERY of them you will need to download the xml file and replace the values with the variables. The job NSO does for us, is applying the correct configuration depending on the device type (at least this one was expected right ?).

In my opinion, **this is NOT yet automation at all**: filling up a GUI to perform configurations, instead of ssh-ing the devices, doesn't save time nor money, and it doesn't make the whole process less dangerous or error free.

There should be a way to upload 'bulks' of data for example through a spreadsheet, ALL data should be checked for consistency and potential errors, also referring to existing node's configurations and external databases, and THEN the whole bunch of configurations should be applied. It is COMPLEX and requires A LOT of work.

Sometimes services can be potentially complex. Desinging the automation, means desinging how to split services into simple configurations, and put them together when needed through another 'father' service that uses them. It depends on how much things can be complicated or not. For example to build a service in ACI, you would need physical ports configuration, I2/I3 configurations, contracts configurations. Each of them could be a single service, but you can potentially think of a service that uses all of them as 'children'. Things can get potentially more and more complex, for this reason there is the possibility of adding a python or java script that reads the variables configured by the operator through a GUI or the NSO CLI, and perform complex checks on the passed parameters, calling different xml template files depending on the parameter's values.

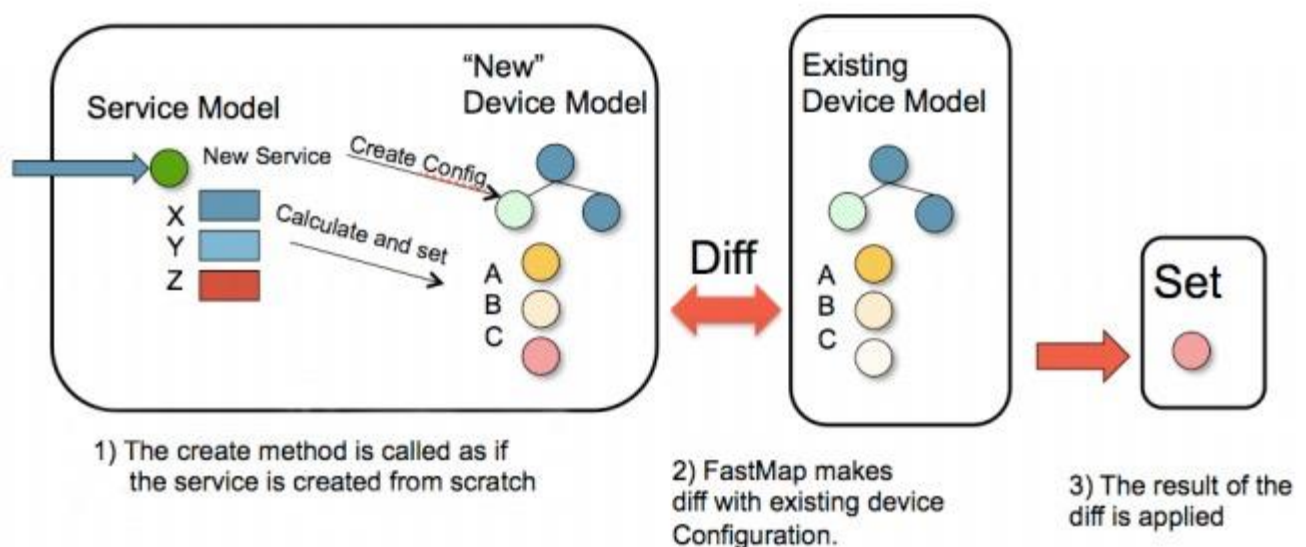
4.1 Fastmap and NED

As a Service Developer you need to express the mapping from a YANG service model to the corresponding device YANG model. This is a declarative mapping in the sense that no sequencing is defined.

Observe that irrespective of the underlying device type and corresponding native device interface, the mapping is towards a YANG device model, not the native CLI for example. This means that as you write

the service mapping, you do not have to worry about the syntax of different devices' CLI commands or in which order these commands are sent to the devices. This is all taken care of by the NSO device manager. NSO reduces this problem to a single data-mapping definition for the "create" scenario. At run-time NSO will render the minimum change for any possible change like all the ones mentioned below. **This is managed by the FASTMAP algorithm.**

FASTMAP covers the complete service life-cycle: creating, changing and deleting the service. The solution requires a minimum amount of code for **mapping from a service model to a device model**. FASTMAP is based on generating changes from an initial create. **When the service instance is created the reverse of the resulting device configuration is stored together with the service instance.** If an NSO user later changes the service instance, NSO first applies (in a transaction) the reverse diff of the service, effectively undoing the previous results of the service creation code. Then it runs the logic to create the service again, and finally executes a diff to current configuration. This diff is then sent to the devices.



4.2 Accessing the Network (NEDs)

The NSO device manager is the centre of NSO. The device manager maintains a flat list of all managed devices. NSO keeps the master copy of the configuration for each managed device in CDB. Whenever a configuration change is done to the list of device configuration master copies, the device manager will partition this "network configuration change" into the corresponding changes for the actual managed devices. The device manager passes on the required changes to the NEDs, Network Element Drivers. A NED needs to be installed for every type of device OS, like Cisco IOS NED, Cisco XR NED, Juniper JUNOS NED etc. The NEDs communicate through the native device protocol southbound. The NEDs fall into the following categories:

- **NETCONF capable device.** The Device Manager will produce NETCONF edit-configuration RPC operations for each participating device.
- **SNMP device.** The Device Manager translates the changes made to the configuration into the corresponding SNMP SET PDUs
- **Device with Cisco CLI.** The device has a CLI with the same structure as Cisco IOS or XR routers. The Device Manager and a CLI NED is used to produce the correct sequence of CLI commands which reflects the changes made to the configuration.

Other devices which do not fit into any of the above mentioned categories a corresponding Generic NED is invoked. Generic NEDs are used for proprietary protocols like REST and for CLI

flavours that are not resembling IOS or XR. The Device Manager will inform the Generic NED about the made changes and the NED will translate these to the appropriate operations toward the device.

4.3 L2 vpn service example

A service that uses python is created:

```
[root@nso LabDir45]# cd packages/
[root@nso packages]# ncs-make-package --service-skeleton python-and-template geoPysnmp
```

Next, review what running this command accomplished:

```
[root@nso packages]# cd geoPysnmp/
[root@nso geoPysnmp]# find .
.
./load-dir
./load-dir/geoPysnmp.fxs
./package-meta-data.xml
./python
./python/geoPysnmp
./python/geoPysnmp/__init__.py
./python/geoPysnmp/main.py
./README
./src
./src/java
./src/java/src
./src/Makefile
./src/yang
./src/yang/geoPysnmp.yang
./templates
./templates/geoPysnmp-template.xml
(skip)
[root@nso geoPysnmp]#
```

The files you will be working with in this lab are:

```
./python/geoPysnmp/main.py          ← Python script
./src/yang/geoPysnmp.yang
./templates/geoPysnmp-template.xml
```

After having modified the yaml and xml files, you can build the service through the 'makefile'. From the Linux shell, do the following:

```
[root@nso yang]# cd ..
[root@nso src]# ls -al
total 8
drwxr-xr-x. 4 root root 43 Feb 22 00:36 .
drwxr-xr-x. 7 root root 4096 Feb 26 23:11 ..
drwxr-xr-x. 3 root root 16 Feb 26 23:11 java
-rw-r--r--. 1 root root 733 Feb 22 00:36 Makefile
drwxr-xr-x. 2 root root 27 Feb 26 18:53 yang

[root@nso src]# make clean all
rm -rf ../load-dir java/src//
mkdir -p ../load-dir
mkdir -p java/src//
/root/nso-4.5.3/bin/ncsc `ls geoPysnmp-ann.yang > /dev/null 2>&1 && echo "-a
geoPysnmp-ann.yang" ` \
-c -o ../load-dir/geoPysnmp.fxs yang/geoPysnmp.yang
```

Python script:

```
1 # -*- mode: python; python-indent: 4 -*-
2 import ncs
```

```

3 from ncs.application import Service
4
5
6 # -----
7 # SERVICE CALLBACK EXAMPLE
8 # -----
9 class ServiceCallbacks(Service):
10
11 # The create() callback is invoked inside NCS FASTMAP and
12 # must always exist.
13 @Service.create
14 def cb_create(self, tctx, root, service, proplist):
15     self.log.info('Service create(service=', service._path, ')')
16
17     #..[setp stufff].....
18     # create a generic variables dictionary
19     geoVars = ncs.template.Variables()
20     # this refers to the template for the service
21     template = ncs.template.Template(service)
22     # initialize vars...
23     geoVars.add('DEV', '')
24     geoVars.add('COMMUNITY', '')
25     geoVars.add('ACCESS', '')
26     geoVars.add('DOMNAME', '')
27     geoVars.add('DSRV', '')
28     geoVars.add('NTP', '')
29     #.....
30
31     #..[get the localTZ for the devices in this service instance].....
32     #..[all devices in this service instance are in this TZ].....
33     localTZ = service.localTZ
34     self.log.debug('localTZ: ', localTZ)
35     #..[now, read stuff from geo-catalog for this TZ].....
36     #..[dommain-name]..(leaf).....
37
38     # 'Root' is the variable to access ALL NCS data, that can be seen as
39     # databases that have been previously populated manually or in some other way.
40     # Variables are accessed through the '.', and can be used to perform complex
41     # checks (e.g. values that should be unique on all devices).
42
43     domName = root.GeoCatalog[localTZ].domainName
44     self.log.debug('domName: ', domName)
45     #..[nameServers]...(leaf-list).....
46     nameServers = root.GeoCatalog[localTZ].nameServer
47     self.log.debug('num nameServers: ', len(nameServers))
48     #..[ntpServers]...(leaf-list).....
49     ntpServers = root.GeoCatalog[localTZ].ntpServer
50     self.log.debug('num ntpServers: ', len(ntpServers))
51     #..[snmpRO] [snmpRW].....
52     snmpRO = root.GeoCatalog[localTZ].snmpCommRO
53     snmpRW = root.GeoCatalog[localTZ].snmpCommRW
54     self.log.debug('snmpRO: ', snmpRO, ' snmpRW: ', snmpRW)
55     #..
56     #..ok, now we have everyone - let's write data out to devices.....
57     #..walk through all the devices (leaf-list).....
58     for i, dev in enumerate(service.device):
59         self.log.debug('dev: ', i, ' name: ', dev)
60         geoVars.add('DEV', dev)
61         geoVars.add('COMMUNITY', snmpRO)
62         geoVars.add('ACCESS', 'ro')
63
64     # here is applied the right template specified as a string:
65     # 'geoPysnmp-template'
66     # ... passing the variables in the dictionary as they have been previously populated
67     # As specified by Cisco, the same template can be applied multiple times, NSO
68     # will only execute missing configurations related to the new added variables.
69     # Maybe not the best thing to be seen, but variables need to be ALL there, for
70     # this reason they are initialized as empty.
71
72     template.apply('geoPysnmp-template', geoVars)
73     geoVars.add('COMMUNITY', snmpRW)
74     geoVars.add('ACCESS', 'rw')
75     template.apply('geoPysnmp-template', geoVars)
76     geoVars.add('DOMNAME', domName)

```



```

60         template.apply('geoPysnmp-template', geoVars)
61     for dsrv in nameServers:
62         geoVars.add('DSRV', dsrv)
63         template.apply('geoPysnmp-template', geoVars)
64     for ntp in ntpServers:
65         geoVars.add('NTP', ntp)
66         template.apply('geoPysnmp-template', geoVars)
67
68
69 # -----
70 # COMPONENT THREAD THAT WILL BE STARTED BY NCS.
71 # -----
72 class Main(ncs.application.Application):
73     def setup(self):
74         self.log.info('Main RUNNING')
75         self.register_service('geoPysnmp-servicepoint', ServiceCallbacks)
76
77 def teardown(self):
78     self.log.info('Main FINISHED')

```

[1] lines 1 through 16, the first group, at the top (not highlighted), are created by the skeleton-builder. These lines provide the service decorator and "create" module definition. **This is required; do not modify this code.**

[2] lines 17 through 27, the second group, (highlighted in RED), are setting up the variable "geoVars" (which is a python "dictionary" and holds key/value pairs (KVP)), and initializing KVPs for the variables assigned in the XML template in the previous Step 4 to "empty strings".

[3] lines 29 through 32, the third group, (highlighted in BLUE), are reading the "local time zone" from the service YANG model (one of the inputs for the service instance). (Technically, there is no reason to store this as a variable (localTZ) in python; you could simply refer to "service.localTZ" each time it is required in the code. If this is your preferred coding style, please feel free to express yourself.

[4] lines 33 through 46, the fourth group, (highlighted in PURPLE), are reading data out the GeoCatalog - our external database, to obtain values that are appropriate to the "localTZ" we just read. (That is the purpose of having this input from the service-model; it defines how the GeoCatalog should be read. Here, a few additional notes are useful.

-- lines 35: reads the GeoCatalog YANG leaf called domainName; this is of type "string" and so it can be assigned to a variable ("domName" here).

-- lines 38: reads the GeoCatalog YANG leaf-list called nameServer; this is of type "inet:ip-address". Because it's a "leaf-list" (keyless list), the variable its assigned to ("nameServers" here) will contain the whole list. Later In the code (for writing), we'll need to loop through the list to read all elements.

-- lines 41: reads the GeoCatalog YANG leaf-list called ntpServer; this is of type "inet:ip-address". Because it's a "leaf-list" (keyless list), the variable its assigned to ("ntpServers" here) will contain the whole list. Later In the code (for writing), we'll need to loop through the list to read all elements.

-- lines 44: reads the GeoCatalog YANG leaf called snmpCommRO; this is of type "string" and so it can be assigned to a variable ("snmpRO" here).

-- lines 45: reads the GeoCatalog YANG leaf called snmpCommRW; this is of type "string" and so it can be assigned to a variable ("snmpRW" here).

[5] lines 50 through 66, the fifth group, (not highlighted), assigns values read from the GeoCatalog to the variables that will be passed to the XML template when it is applied.

-- line 50: "devices" are stored in a leaf-list (see service model YANG); there may be more than one, so the code needs to loop through the leaf-list. Line 50 initiates this "for loop".

-- line 52: each "device" in the leaf-list is assigned to "dev" (from the for-loop). In line 52, "dev" is assigned to the KVP "DEV" using the Python dictionary add operation.

-- line 53: In line 53, "snmpRO" is assigned to the KVP "COMMUNITY" using the Python dictionary add operation.

-- line 54: In line 54, "ro" (hard-coded) is assigned to the KVP "ACCESS" using the Python dictionary add operation. (The KVPs "COMMUNITY" and "ACCESS" are re-used after this for the "rw" case as well). This is solely to accommodate the XML template structure we've chosen. Alternatives of course can be used.

-- line 55: In line 55, the template "geoPysnmp-template" is applied. At this point, all assigned variables will be passed to the XML template and FastMap will determine what needs to be sent to the device. Because only some of the variables are set up to this point, only these parts of the configuration will be determined. (This is why all KVPs were initialized in item [2].)

-- lines 56, 57, and 58: similarly to lines 52, 53, and 54, lines 56 and 57 assign values to "COMMUNITY" (from snmpRW) and "ACCESS" (hard-coded as "rw") respectively, and line 58 applies the template "geoPysnmp-template".

-- lines 59 and 60: line 59 assigns a value to "DOMNAME" (from domName) and line 60 applies the template "geoPysnmp-template".

-- lines 61, 62, and 63: line 61 initiates a for-loop over the leaf-list "nameServers", line 62 assigns a value to "DSRV" (from dsrv), and line 63 applies the template "geoPysnmp-template".

-- lines 64, 65, and 66: line 64 initiates a for-loop over the leaf-list "ntpServers", line 65 assigns a value to "NTP" (from ntp), and line 66 applies the template "geoPysnmp-template".

[6] lines 69 through 78, the last group, are also provided by the skeleton-builder (but with commented out lines deleted). These register and tear down the service. **This is required; do not modify (other than deleting the commented-out lines from the builder).**

At the NSO CLI (juniper CLI mode shown):

```
admin@ncs> request packages reload
(skip)
reload-result {
package geoPysnmp
result true
}
(skip)
admin@ncs>
```

4.4 Loops and arrays inside the xml file

I'm attempting to create my first service template but have issues when referencing device-groups in my yang file.

```
user@ncs(config)# L2-BRIDGE TEST device-group LEAF-SWITCHES vlan 10 vni 10002
```

```
user@ncs(config-L2-BRIDGE-TEST)# commit dry-run outformat native
```

```
Aborted: L2-BRIDGE-template.xml:5 The node '/ncs:devices/device{LEAF-SWITCHES}'
was not created. An explicit tag 'merge' or 'create' is needed.
```

The device-group LEAF-SWITCHES exists within NSO and contains some switches.

```
leaf-list device-group {
  type leafref {
    path "/ncs:devices/ncs:device-group/ncs:name";
  }
}
```

If I replace 'device-group' with 'device' I am able to run the service against individual nodes. The service yaml file is the following:

! XML SERVICE CONFIGURATIONS

```
<config-template xmlns="http://tail-f.com/ns/config/1.0"
  servicepoint="L2-BRIDGE">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>{/device-group}</name> ← WRONG !!!!!!!!!!!
    <config>
      <vlan xmlns="http://tail-f.com/ned/cisco-nx">
        <vlan-list>
          <id>{/vlan}</id>
          <name>{/customer-name}</name>
          <vn-segment>{/vni}</vn-segment>
        </vlan-list>
      </vlan>
      <evpn xmlns="http://tail-f.com/ned/cisco-nx">
        <vni>
          <id>{/vni}</id>
          <l2/>
          <rd>auto</rd>
          <route-target>
            <method>import</method>
```

```

        <rt>auto</rt>
    </route-target>
    <route-target>
        <method>export</method>
        <rt>auto</rt>
    </route-target>
</vni>
</evpn>
<interface xmlns="http://tail-f.com/ned/cisco-nx">
    <nve>
        <name>1</name>
        <member>
            <vni>
                <id>{/vni}</id>
                <suppress-arp/>
                <ingress-replication>
                    <protocol>
                        <bgp/>
                    </protocol>
                </ingress-replication>
            </vni>
        </member>
    </nve>
    <port-channel>
        <name>200</name>
        <switchport>
            <trunk>
                <allowed>
                    <vlan>
                        <ids>{/vlan}</ids>
                    </vlan>
                </allowed>
            </trunk>
        </switchport>
    </port-channel>
</interface>
</config>
</device>
</devices>
</config-template>

```

! YAML SERVICE DESCRIPTION

```

module L2-BRIDGE {
    namespace "http://com/example/L2BRIDGE";
    prefix L2-BRIDGE;

    import ietf-inet-types {
        prefix inet;
    }
    import tailf-ncs {
        prefix ncs;
    }

    list L2-BRIDGE {
        key customer-name;

        uses ncs:service-data;
        ncs:servicepoint "L2-BRIDGE";

        leaf customer-name {
            type string;
        }
    }
}

```

```

leaf-list device-group {
    type leafref {
        path "/ncs:devices/ncs:device-group/ncs:name";
    }
}
leaf vlan {
    mandatory true;
    type uint16;
}

leaf vni {
    mandatory true;
    type uint16;
}
}
}

```

To reference a list inside the xml file, a for cycle has to be configured. Probably there are also some other way to do this, but it should be really managed here or through **FastMap** feature (?????).

```

<devices xmlns="http://tail-f.com/ns/ncs">
  <?foreach {/device-group}?>
    <device>
      <name>{deref(.) / ../member}</name>
      ...
    </device>
  <?end?>
</devices>

```

One other possible way to manipulate things is the following:

```

<?foreach {/device-group}?>           ← this is a list
  <device>
    <name>{deref(.) / ../member}</name>
    <?set devname = {.}?>               ← references the above variable
      <config>
        <configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm">
          <system>
            <ntp>
              <?foreach {/some-server}?> ← list of ntp servers
                <server>
                  <name>{$devname}</name>
                </server>
              </foreach>
            </ntp>
          </system>
        </configuration>
      </config>
    </device>
  <?end?>

```

Inside a python script (after importing ncs library):

```

devGrp = root.ncs__devices.device_group['nx-dev']
print ("members")
for member in devGrp.member:
    print (member)

for group in service.device_group:
    self.log.info('Group name is ', group)
    for member in device_group[group]:
        someserver['device_name'] = member

```

4.5 Programming statements inside xml

As seen before there are 'foreach' and 'set' statements that can be used inside xml, looks a bit like jinja2 templates. What else can be used ? for example there are also 'choice' statements (like if and else in programming languages):

```
choice interface {
  case ios-GigabitEthernet {
    leaf GigabitEthernet {
      type leafref {
        path
        "deref(..../endpoint1)/../ncs:config/ios:interface/ios:GigabitEthernet/ios:name";
      }
    }
  }
  case ios-TenGigabitEthernet {
    leaf TenGigabitEthernet {
      type leafref {
        path
        "deref(..../endpoint1)/../ncs:config/ios:interface/ios:TenGigabitEthernet/ios:name";
      }
    }
  }
}
```

4.6 L3 vpn service

<https://github.com/NSO-developer/Cisco-NSO-MPLS-VPN-service-reconciliation-example>

On Cisco community blog:

<https://community.cisco.com/t5/nso-developer-hub-discussions/nso-l3vpn-yang-template-help/td-p/3470056>

Hi, I'm trying to build a model for L3VPN provisioning with options. Options are: CE bgp routing and CE static routing. I must add I'm a newbie to NSO. Here's a snippet of my yang model. You'll see later on that variables like "intf-number" and "wan-ip-address" are being used properly, but variables within the "static" list are not. Or at least is not what I expected to happen and someone might offer me an explanation.

```
[snip]
  leaf intf-number {
    tailf:info "GigabitEthernet Interface ID";
    mandatory true;
    type string {
      pattern "[0-9]{1,2}(/[0-9]{1,2}){1,4}";
    }
  }
  leaf wan-ip-address {
    tailf:info "IP Address of WAN Interface (X.X.X.X)";
    mandatory true;
    type inet:ipv4-address {
      pattern "[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+";
    }
  }
[snip]
  leaf routing-protocol {
    tailf:info "PE<-->CE Routing Protocol";
    mandatory true;
    type enumeration {
      enum bgp;
      enum static;
    }
  }
  list static {
    tailf:info "Static Route for VRF";
```

[snip]

```
<ip xmlns="http://tail-f.com/ned/cisco-ios-xr">
  <route tags='merge' when="{routing-protocol='static'}">
    <vrf>
      <name>{/vpn-name}</name>
      <dest>{/static-prefix}</dest>
      <dest-mask>{/static-mask}</dest-mask>
      <forwarding-address>{/static-next-hop}</forwarding-address>
    </vrf>
  </route>
</ip>
```

```
admin@ncs(config)# services fullvpn test routing-protocol static static 1.1.1.0
static-mask 255.255.255.0 static-next-hop 172.16.1.2
Value for 'vpn-name' (<string>): test
Value for 'vpn-id' (<unsingedInt, 1 .. 9999>): 1
Value for 'vlan-id' (<unsingedInt, 1 .. 4000>): 1
Value for 'device' [IOS-0,IOS-1,XR-0,XR-1,...]: XR-0
Value for 'intf-number' (<string>): 0/0/0/0
Value for 'wan-ip-address' (<IPv4 address>): 172.16.1.1
```

```
result-xml {
  local-node {
    data <devices xmlns="http://tail-f.com/ns/ncs">
      <device>
        <name>XR-0</name>
        <config>
          <vrf xmlns="http://tail-f.com/ned/cisco-ios-xr">
            <vrf-list>
              <name>test</name>
              <address-family>
                <ipv4>
                  <unicast>
                    <import>
                      <route-target>
                        <address-list>
                          <name>1:1</name>
                        </address-list>
                      </route-target>
                    </import>
                  <export>
                    <route-target>
                      <address-list>
                        <name>1:1</name>
                      </address-list>
                    </route-target>
                  </export>
                </ipv4>
              </address-family>
            </vrf-list>
          </vrf>
        </config>
      </device>
    </data>
  }
}
```

```

        </address-list>
      </route-target>
    </export>
  </unicast>
</ipv4>
</address-family>
</vrf-list>
</vrf>
<ip xmlns="http://tail-f.com/ned/cisco-ios-xr">
  <route>
    <vrf>
      <name>test</name>
    </vrf>
  </route>
</ip>
<interface xmlns="http://tail-f.com/ned/cisco-ios-xr">
  <GigabitEthernet-subinterface>
    <GigabitEthernet>
      <id>0/0/0/0.1</id>
      <ipv4>
        <address>
          <ip>172.16.1.1</ip>
          <mask>255.255.255.252</mask>
        </address>
      </ipv4>
      <vrf>test</vrf>
      <encapsulation>
        <dot1q>
          <vlan-id>1</vlan-id>
        </dot1q>
      </encapsulation>
    </GigabitEthernet>
  </GigabitEthernet-subinterface>
</interface>
<router xmlns="http://tail-f.com/ned/cisco-ios-xr">
  <bgp>
    <bgp-no-instance>
      <id>1</id>
      <vrf>
        <name>test</name>
        <rd>1:1</rd>
        <address-family>
          <ipv4>
            <unicast>
              <redistribute>
                <connected/>
                <static/>
              </redistribute>
            </unicast>
          </ipv4>
        </address-family>
      </vrf>
    </bgp-no-instance>
  </bgp>
</router>
</config>
</device>
</devices>
<services xmlns="http://tail-f.com/ns/ncs">
  <fullvpn xmlns="http://com/example/fullvpn">
    <name>test</name>
    <vpn-name>test</vpn-name>
  </fullvpn>
</services>

```

```

        <vpn-id>1</vpn-id>
        <vlan-id>1</vlan-id>
        <device>XR-0</device>
        <intf-number>0/0/0/0</intf-number>
        <wan-ip-address>172.16.1.1</wan-ip-address>
        <routing-protocol>static</routing-protocol>
        <static>
            <static-prefix>1.1.1.0</static-prefix>
            <static-mask>255.255.255.0</static-mask>
            <static-next-hop>172.16.1.2</static-next-hop>
        </static>
    </fullvpn>
</services>
}
}

```

As you can see, the XR static ip route is not filling out properly and the template uses only the variables that are outside of the static list node. Yet the services template at the very end sees the correct values in the variables, but under "services". All those variables are never configured in the router. Any advice would be appreciated.

4.7 Nested Services

```

module lower1 {
    augment /ncs:services {          ← to be referenced
    list lower1 {
        key name;
        leaf name {
            type string;
        }
        uses ncs:service-data;
        ncs:servicepoint lower1-servicepoint;
        leaf device {
            mandatory true;
            type string;
        }}}
    ----
module lower2 {
    augment /ncs:services {          ← to be referenced
    list lower2 {
        key name;
        leaf name {
            type string;
        }
        uses ncs:service-data;
        ncs:servicepoint lower2-servicepoint;
        leaf device {
            mandatory true;
            type string;
        }}}
    ====
module higher {
    augment /ncs:services {
    list higher {
        key name;
        leaf name {
            type string;
        }

        uses ncs:service-data;
        ncs:servicepoint higher-servicepoint;
    }
}

```

```

container lower1 {
presence "lower1";
uses lower1:lower1;
}
container lower2 {
presence "lower2";
uses lower2:lower2;
}}}}

```

4.8 External data books

See the last example in the lab listed in the pdf. A service 'GeoCatalog' could be created in a standalone package:

```

26 /*.....
27 .. Geo Catalog here ..
28 .....*/
29
30 // cdb entries outside the service based on geoloc...
31 // certain parameters are applicable to associated devices
32
33 list GeoCatalog {
34 key timeZone;
35
36 leaf timeZone {
37 tailf:info "geoloc timezone";
38 type enumeration {
39 enum PST; Cisco dCloud dCloud: The Cisco Demo Cloud
40 enum MST;
41 enum CST;
42 enum EST;
43 }
44 }
45 leaf domainName {
46 tailf:info "ip domain-name";
47 default bigtelco.com;
48 type string;
49 }
50 leaf-list nameServer {
51 tailf:info "name-server ip";
52 min-elements 2;
53 max-elements 2;
54 type inet:ip-address;
55 }
56 leaf-list ntpServer {
57 tailf:info "ntp server ip";
58 min-elements 2;
59 max-elements 2;
60 type inet:ip-address;
61 }
62 leaf snmpCommRO {
63 tailf:info "snmp server RO community";
64 type string;
65 }
66 leaf snmpCommRW {
67 tailf:info "snmp server RW community";
68 type string;
69 }
70 } // list GeoCatalog

```

Then in ANOTHER package you can create the real service model, that references that package data:

```

73 /*.....
74 .. Service Model ..
75 .....*/
76 IMPORT GEOCATALOG HERE
77 list geoPysnmp {
78 description "This is a demonstration model

```



```

79 using an a separate database structure for services data";
80
81 key name;
82 leaf name {
83   tailf:info "unique ID string";
84   type string;
85 }
86
87 uses ncs:service-data;
88 ncs:servicepoint geoPysnmp-servicepoint;
89
90 // may replace this with other ways of referring to the devices.
91 leaf-list device {
92   type leafref {
93     path "/ncs:devices/ncs:device/ncs:name";
94   }
95 }
96 leaf localTZ {
97   tailf:info "Specify the TZ for the device(s)";
98   type leafref {
99     path "../GeoCatalog/timeZone";           ← reference to the above GeoLoc data
100 }
101 } // leaf localTZ
102
103 } // list geoPysnmp

```

IDEA: The "geo-catalog" YANG model could have been built in a separate package, with just the YANG model, and loaded separately into CDB. That is, there is no requirement or need to include this (or any) YANG model directly in this service model package. If the approach is used where the YANG model is defined separately, in order for the above service model to take advantage of the geo-catalog (for example, as is done here in the "localTZ" leaf by leafref), it would be necessary to **"import" this external package (along with the other imports in the top section) and add it to the Makefile for lookup during compilation**. However, because it was defined inside the same YANG file here, it's automatically available to the entirety of this package.

To populate the external database, manual commands can be used:

```

GeoCatalog PST ntpServer [ 1.1.1.1 1.1.1.2 ] nameServer [ 1.1.1.3 1.1.1.4 ]
GeoCatalog PST domainName pstbigtelco.com
GeoCatalog PST snmpCommRO foobarpst
GeoCatalog PST snmpCommRW barfoopst
GeoCatalog MST ntpServer [ 1.1.2.1 1.1.2.2 ] nameServer [ 1.1.2.3 1.1.2.4 ]
GeoCatalog MST domainName mstbigtelco.com
GeoCatalog MST snmpCommRO foobarmst
GeoCatalog MST snmpCommRW barfoomst
GeoCatalog CST ntpServer [ 1.1.3.1 1.1.3.2 ] nameServer [ 1.1.3.3 1.1.3.4 ]
GeoCatalog CST domainName cstbigtelco.com
GeoCatalog CST snmpCommRO foobarcst
GeoCatalog CST snmpCommRW barfoocst
GeoCatalog EST ntpServer [ 1.1.4.1 1.1.4.2 ] nameServer [ 1.1.4.3 1.1.4.4 ]
GeoCatalog EST domainName estbigtelco.com
GeoCatalog EST snmpCommRO foobarest
GeoCatalog EST snmpCommRW barfooest
Top

```

```

admin@ncs% commit dry-run
cli {
local-node {
data +GeoCatalog PST {
+ domainName pstbigtelco.com;
+ nameServer [ 1.1.1.3 1.1.1.4 ];
+ ntpServer [ 1.1.1.1 1.1.1.2 ];
+ snmpCommRO foobarpst;

```

```

+ snmpCommRW barfoopst;
+}
+GeoCatalog MST {
+ domainName mstbigtelco.com;
+ nameServer [ 1.1.2.3 1.1.2.4 ];
+ ntpServer [ 1.1.2.1 1.1.2.2 ];
+ snmpCommRO foobarmst;
+ snmpCommRW barfoomst;
+}
+GeoCatalog CST {
+ domainName cstbigtelco.com;
+ nameServer [ 1.1.3.3 1.1.3.4 ];
+ ntpServer [ 1.1.3.1 1.1.3.2 ];
+ snmpCommRO foobarcst;
+ snmpCommRW barfoocst;
+}
+GeoCatalog EST {
+ domainName estbigtelco.com;
+ nameServer [ 1.1.4.3 1.1.4.4 ];
+ ntpServer [ 1.1.4.1 1.1.4.2 ];
+ snmpCommRO foobarest;
+ snmpCommRW barfoolest;
+}
}
}
[ok][2018-02-27 01:04:21]

admin@ncs% commit
Commit complete.
[ok][2018-02-27 01:04:23]
[edit]
admin@ncs%

```

Great – now the GeoCatalog is packed with data. If you want to “show” the data for a particular timezone, just ask NSO. For example, show the PST data:

```

admin@ncs% show GeoCatalog PST
domainName pstbigtelco.com;
nameServer [ 1.1.1.3 1.1.1.4 ];
ntpServer [ 1.1.1.1 1.1.1.2 ];
snmpCommRO foobarpst;
snmpCommRW barfoopst;
[ok][2018-02-27 01:05:13]

```

4.9 Makefile

In your Makefile for your package, you can update the YANGPATH to include the path to the yang files you need. In each package there is a load-dir where fxs files are loaded. The default makefile for a package places the fxs files there. To make the module available to EVERY other package, probably the easiest way is that of moving the fxs file.

I put my custom test-external-pkg.yang into the `/opt/ncs/current/src/ncs/yang` directory where all the other default yang files (for example ietf-inet-types.yang). Then ran the **make** command, it created my test-external-pkg.fxs inside of the `/opt/ncs/current/src/ncs/yang`. After watching the debug logs of NSO trying to start I noticed that all the other default .fxs files were located in:

```
/opt/ncs/ncs-4.7.2/etc/ncs/
```

I manually moved my test-external-pkg.fxs to that above directory and it worked.

```
sudo mv /opt/ncs/current/src/ncs/yang/test-external-pkg.fxs /opt/ncs/ncs-4.7.2/etc/ncs/
```

Still looking for docs on the correct process of adding a custom yang then getting the .fxs to create in the proper load directory. Thinking something with the way I am running the make command.

4.10 Interfaces choice and references

Hi,

I have a question regarding a basic YANG model I am creating.

It is a YANG model to collect input data in NSO WebUI for a Site-to-Site VPN (IPSec) Service.

Please consider the following leafs/typedefs:

```
typedef interface_type {
    type enumeration {
        enum GigabitEthernet;
        enum Ethernet;
        enum FastEthernet;
    }
}

typedef interface_number {
    type string {
        pattern "(\\d+) (/) (\\d+)";           ← why double slash ?!?!
    }
}

[...]
leaf endpoint1 {
    type leafref {
        path "/ncs:devices/ncs:device/ncs:name";
    }
    mandatory true;
    description 'Endpoint number one for VPN Tunnel';
    tailf:info "Endpoint number one for VPN Tunnel";
}
leaf interfacel_type {
    type interface_type;
    //type leafref {
    //    path "????";
    //}
    mandatory true;
    description 'Interface type of interface 1';
    tailf:info "Interface type of interface 1";
}
leaf interfacel_number {
    type interface_number;
    //type leafref {
    //    path "????";
    //}
    mandatory true;
    description 'Interface number of interface 1';
    tailf:info "Interface number of interface 1";
}
```

```
}
```

The leaf endpoint1 gives me the possibility to choose only devices which exist in the NSO CDB (in WebUI). So far so good. I would like to have a similar dropdown for interface type and interface number (or name) which then don't use the typedefs at the top but rather reference the actual info in the CDB.

So the dropdown for "interface_type" should show all interface types which can be configured on an IOS device and the dropdown for "interface_number" should show only the interface numbers/names of actually available interfaces on the device specified in endpoint1.

I have tried various things which all led nowhere for me (importing the yang model of the cisco-ios NED into my own and tried referencing the grouping/choice -interface-name-grouping/interface-choice- which exist in the NED yang model to populate the dropdown.

During compilation I keep getting an error that my path argument is wrong: "should be of type path-arg". As you can probably tell I am not great with XPath...pretty sure the paths I tried aren't even valid XPath and therefore I get the error.

SOLUTION

```
import tailf-ned-cisco-ios{
    prefix ios;
}
```

Also the Makefile needs something uncommented and patched, so that the NED YANG model is considered at compilation time:

```
## Uncomment and patch the line below if you have a dependency to a NED
## or to other YANG files
```

```
YANGPATH += ../../cisco-ios/src/ncsc-out/modules/yang
```

NOTE:

```
YANGPATH += --yangpath ../../cisco-ios/src/ncsc-out/modules/yang
```

... the --yangpath option was needed in a previous version of the Makefiles and it is not longer needed.

PS: If you check your Makefile, you would find the "--yangpath" option already added here:

```
NCSCPATH = $(YANGPATH:%=--yangpath %)
```

Nevertheless I still get an error message:

```
error: the node 'interface' from module 'tailf-ned-cisco-ios' (in node 'config'
from 'tailf-ncs') is not found
```

This is the path expression I use:

```
path
"deref(..endpoint1)/../ncs:config/ios:interface/ios:GigabitEthernet/ios:name";
```

Follows hereafter an example to try to get a list of choosable interfaces and ip addresses:

```
case GigabitEthernet{
    leaf GigabitEthernet-1{
        type leafref {
            path
            "deref(..endpoint1)/../ncs:config/ios:interface/ios:GigabitEthernet/ios:name";
        }
    }
}
```

```

leaf ip-1{
  type leafref{
    path "deref(../GigabitEthernet-
1/../../ios:ip/ios:address/ios:primary/ios:address)"; //????
  }
}
}

```

I simply wanted to have the (primary) IP address corresponding to the selected interface on a leaf which I could then reference in my XML template, so there wouldn't be any need for fetching it in the service logic in Java/Python.

I guess you are right though, the selection process for the user finishes by selecting the interface, which implies a certain IP, which then gets fetched in the service logic using Java/Python. Thanks for putting me on that path which I am now likely to follow.

4.11 Matching devices of a specified group

While I don't see anything in the Yang RFC specifically forbidding the above expression, the ncsc compiler treats this as an error - you cannot have literals in a leafref path expression. So, the use of the literal key 'Hubs' would be an error.

If you really need this, you can have another leaf for the name of the device-group. So, something like

```

leaf group {
  type leafref {
    path "/ncs:devices/ncs:device-group/ncs:name";
  }
  default "Hubs";
}

leaf hub {
  type leafref {
    path "/ncs:devices/ncs:device-
group[ncs:name=current()/../group]/ncs:device-name";
  }
}

```

This way, you can get completions from the 'Hubs' device-group by default. You can also use "tailf:hidden full" in the "group" leaf above if you don't want to take it as a service input param.

The other thing is that, a device group can contain other device groups - meaning that referring to /devices/device-group/device-name will not work if you have other device-groups in your 'Hubs' device-group. I don't know if that is the case. If not, the above snippet will be fine. But if you do have nested device-groups, you need to refer to the 'member' leaf-list which contains a flat list of all the devices in the group. The problem is that 'member' is a config-false leaf-list, serviced by an internal callpoint. So, you cannot use that as a regular leafref path, as shown above. I came up with this:

```

leaf group {
  type leafref {
    path "/ncs:devices/ncs:device-group/ncs:name";
  }
  default "Hubs";
}

leaf hub {
  type leafref {
    path "/ncs:devices/ncs:device/ncs:name";
  }
}

```

```

tailf:non-strict-leafref {
    path "/ncs:devices/ncs:device-
group[ncs:name=current()/../group]/ncs:member";
}
}

```

Here, the actual type of the leaf is a reference to all devices - /devices/device/name. But the non-strict-leafref (which is a tailf/cisco specific extension) will make the CLI completion work only with those values from that specific device-group.

4.12 Useful NCS commands

```

echo $NCS_DIR
ncs_cli -u admin -C
run show devices list
request devices sync-from
run show packages package oper-status
cd packages
ncs-make-package -help
ncs-make-package --service-skeleton template snmpTemp1

```

```

admin@ncs# config
! note the config context/ned represented by 'ios:snmp-server'
admin@ncs(config)# devices device c0 config ios:snmp-server community BARFOO RW
admin@ncs(config-config)# devices device c0 config ios:snmp-server community
FOOBAR RO
admin@ncs(config-config)# commit dry-run outformat xml

```

```

admin@ncs(config)# devices device c0 config <cfg context> <configs>

```

```

! apply same config template to all devices
devices device-group ALL apply-template template-name SET-DNS-SERVER

```

```

show devices device dist-sw01 platform
show devices device dist-sw01 platform serial-number

```

```

! check for 'live status'
show devices device * platform serial-number
show devices device dist-sw01 live-status port-channel
show devices device dist-sw01 live-status ip route

```

```

devices device dist-sw01 live-status exec show license usage
devices device dist-sw01 live-status exec any dir

```

```

devices device dist* live-status exec show license usage | save
/home/developer/nexus-license-usage.txt

```

```

! check all devices if they are conformant to a defined template
compliance reports report COMPLIANCE-CHECK run outformat text
compliance reports report COMPLIANCE-CHECK run outformat html

```

```

show full-configuration devices device device0 | display xml

```

```

[root@nso yang]# cd ..
[root@nso src]# ls -al
total 4
drwxr-xr-x. 3 root root 32 Feb 14 00:56 .

```

```

drwxr-xr-x. 5 root root 71 Feb 14 00:56 ..
-rw-r--r--. 1 root root 722 Feb 14 00:56 Makefile
drwxr-xr-x. 2 root root 27 Feb 14 00:56 yang
[root@nso src]# make clean all
rm -rf ../load-dir
mkdir -p ../load-dir
/root/nso-4.5.3/bin/ncsc `ls snmpTemp1-ann.yang > /dev/null 2>&1 && echo "-a
snmpTemp1-ann.yang" ` \
-c -o ../load-dir/snmpTemp1.fxs yang/snmpTemp1.yang
[root@nso src]#

```

```
admin@ncs> request packages reload
```

```

admin@ncs% set snmpTemp1 BARFOO access ro device c0
[ok][2018-02-14 22:56:56]
[edit]
admin@ncs% commit dry-run
admin@ncs% commit dry-run outformat native
admin@ncs% commit

```

In case a rollback is needed, the following commands could be used inside a python script to perform a **selective rollback**:

```

def rollback_id( root, rollback_number):
apply_rollback_file = root.rollback_files.apply_rollback_file
args = apply_rollback_file.get_input()
args.fixed_number = rollback_number
args.selective.create()
apply_rollback_file(args)

```

If you would like to check the **netsim** device, you can log in directly from the Linux shell, as follows:

```

[root@nso LabDir45]# ncs-netsim cli-c c0
admin connected from x.x.x.x using ssh on nso.demo.dcloud.cisco.com
c0# show running-config snmp-server
snmp-server community BARFOO RO
c0#

```

From the NSO CLI, observe the device configuration – focusing on the snmp-server config:

```

admin@ncs% show devices device c0 config ios:snmp-server | display service-meta-
data
/* Refcount: 1 */
/* Backpointer: [ /snmpTemp1:snmpTemp1[snmpTemp1:comm-str='BARFOO'] ] */
community BARFOO {
/* Refcount: 1 */
/* Backpointer: [ /snmpTemp1:snmpTemp1[snmpTemp1:comm-str='BARFOO'] ] */
RO;
}
[ok][2018-02-14 23:58:36]

```

```
ncs-make-package --service-skeleton python-and-template snmpPyTemp5
```

```

[root@nso LabDir45]# cd logs/
[root@nso logs]# ls -al
(skip...)
-rw-r--r--. 1 root root 726328 Feb 23 19:51 devel.log
-rw-r--r--. 1 root root 72959 Feb 23 19:51 ncs-java-vm.log

```

```
-rw-r--r--. 1 root root 2117 Feb 23 19:51 ncs-python-vm-snmpPyTemp5.log
-rw-r--r--. 1 root root 2639213 Feb 23 19:51 xpath.trace    ← xml files
parsing logs
(skip...)
[root@nso logs]#admin@ncs>
```

Check on multiple devices' configurations:

```
admin@ncs# show running-config devices device ios* config ios:vrf definition foo
devices device ios-0
  config
    ios:vrf definition foo
  !
!
!
devices device ios-1
  config
    ios:vrf definition foo
  !
!
!
admin@ncs# show running-config devices device ios* config ios:vrf definition bar
% No entries found.
```

4.13 Troubleshooting and debug

Should be of type path-arg

In this case make sure your XPATH does not have a hardcoded variable.

```
yang/new_l2vpn.yang:60: error: bad argument value
"/ncs:devices/ncs:device{PE11}/ncs:config/ios:interface/ios:GigabitEthernet{1}/i
os:name", should be of type path-arg
```

error: the leafref refers to non-leaf and non-leaf-list node

```
yang/new_l2vpn.yang:38: error: the leafref refers to non-leaf and non-leaf-list
node 'FastEthernet' in module 'tailf-ned-cisco-ios' at
/opt/ncs/packages/neds/cisco-ios/src/ncsc-out/modules/yang/tailf-ned-cisco-
ios.yang:34121
```

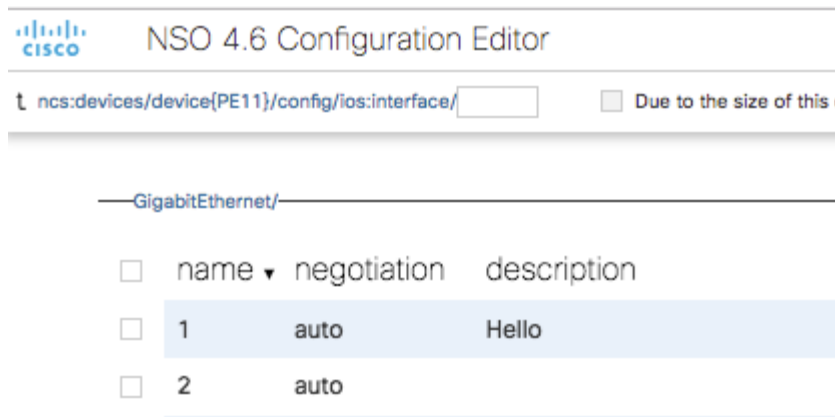
In this case make sure your XPATH points to a list, in this case it was `ios:name`

```
path
"/ncs:devices/ncs:device/ncs:config/ios:interface/ios:GigabitEthernet/ios:name";
`
```

The error occurs if you leave the path until here, expecting to list GigabitEthernet interfaces:

```
path "/ncs:devices/ncs:device/ncs:config/ios:interface/ios:GigabitEthernet";
```

Basically the key is, things like `name`, `negotiation`, `descripton` which is a list.



XPath error: Invalid namespace prefix: ios

```
yang/circuit.yang:57: error: XPath error: Invalid namespace prefix: ios  
make: *** [../load-dir/circuit.fxs] Error 1
```

Solution :

Import the required module:

```
import tailf-ned-cisco-ios {  
    prefix ios;  
}
```

And add the yang path below in Makefile.

```
## Uncomment and patch the line below if you have a dependency to a NED  
## or to other YANG files  
# YANGPATH += ../../<ned-name>/src/ncsc-out/modules/yang \  
#           ../../<pkt-name>/src/yang  
  
YANGPATH += ../../cisco-ios/src/ncsc-out/modules/yang
```

Aborted: no registration found for callpoint learning_defref/service_create of type=external

This message occurs when the you perform a commit or commit-dry-run . At this stage your YANG is merged with the corresponding XML Definition.

```
admin@ncs(config-link-PE21)# commit dry-run  
Aborted: no registration found for callpoint learning_defref/service_create of  
type=external  
admin@ncs(config-link-PE21)# exit
```

Solution :

Make sure that in the corresponding XML code , the servicepoint name is correct.

```
<?xml version="1.0" encoding="UTF-8"?>  
<config-template xmlns="http://tail-f.com/ns/config/1.0"  
servicepoint="learning_defref">
```

info [l2vpn-template.xml:2 Unknown servicepoint: l2vpn]

This message occurs when you have not compile the src folder with the `make` command. Since `make` isn't issued, the XML template cannot find the endpoint.

```
reload-result {
  package l2vpn
  result false
  info [l2vpn-template.xml:2 Unknown servicepoint: l2vpn]
}
```

Solution :

Run the `make` command in the `src` folder.

Another Set of error Messages

```
/opt/ncs/packages/neds/cisco-iosxr/src/ncsc-out/modules/yang/tailf-ned-cisco-
ios-xr.yang:17988: warning: Given dependencies are not equal to calculated:
../end-marker, ../start-marker. Consider removing tailf:dependency statements.
/opt/ncs/packages/neds/cisco-iosxr/src/ncsc-out/modules/yang/tailf-ned-cisco-
ios-xr.yang:26306: warning: when tailf:cli-drop-node-name is given, it is
recommended that tailf:cli-suppress-mode is used in combination. using
tailf:cli-drop-nodename in a list child without using tailf:cli-suppress-mode on
the list, might lead to confusing behaviour, where the user enters the submode
without being able to give further configuration.
yang/learning_defref.yang:55: warning: Given dependencies are not equal to
calculated: ../router_name, /ncs:devices/ncs:device/ncs:name,
/ncs:devices/ncs:device/ncs:device-type/ncs:cli/ncs:ned-id. Consider removing
tailf:dependency statements.
yang/learning_defref.yang:68: warning: Given dependencies are not equal to
calculated: ../router_name, /ncs:devices/ncs:device/ncs:name,
/ncs:devices/ncs:device/ncs:device-type/ncs:cli/ncs:ned-id. Consider removing
tailf:dependency statements.
```

Start with the **learning_defref lab**. And analyse the error.

You can delete an entire package out from the NSO by deleting the package from the `packages` dir followed by a `packages reload`. This needs to be forced for the deletion.

```
admin@ncs# packages reload
Error: The following namespaces will be deleted by upgrade:
l2vpn: http://com/example/l2vpn
If this is intended, proceed with 'force' parameter.
admin@ncs# packages reload force

>>> System upgrade is starting.
>>> Sessions in configure mode must exit to operational mode.
>>> No configuration changes can be performed until upgrade has completed.
>>> System upgrade has completed successfully.
reload-result {
  package cisco-ios
  result true
}
reload-result {
  package cisco-iosxr
  result true
}
```

ERROR: Aborted: no registration found for callpoint l2vpn/service_create of type=external

This error points to the the `ncs-run/packages/service-name/template/service_name-template.xml` . Check if the file exists and if any error exists.

```
admin@ncs (config-l2vpn-CE11-CE21)# commit dry-run
Aborted: no registration found for callpoint l2vpn/service_create of
type=external
```

*** make sure you perform `package reload`**

Debugging python scripts:

```
$ ncs_cli -u admin
admin@ncs> config
admin@ncs% set python-vm logging level level-debug
admin@ncs% commit
```

This sets the global logging level and will affect all started Python VMs. It is also possible to set the logging level for a single package (or multiple packages running in the same VM), which will take precedence over the global setting:

```
$ ncs_cli -u admin
admin@ncs> config
admin@ncs% set python-vm logging vm-levels pkg_name level level-debug
admin@ncs% commit
```

The debugging output are printed to separate files for each package and the log file naming is `ncs-python-vm-pkg_name.log`

Log file output example for package l3vpn:

```
$ tail -f logs/ncs-python-vm-l3vpn.log
2016-04-13 11:24:07 - l3vpn - DEBUG - Waiting for Json msgs
2016-04-13 11:26:09 - l3vpn - INFO - action name: double
2016-04-13 11:26:09 - l3vpn - INFO - action input.number: 21
```

NSO can be configured to use a custom start command for starting a Python VM. This can be done by first copying the file `$NCS_DIR/bin/ncs-start-python-vm` to a new file, then change the last lines of that file to start the desired version of Python. After that, edit `ncs.conf` and configure the new file as the start command for a new Python VM. When the file `ncs.conf` has been changed reload its content by executing the command `ncs --reload`.

```
$ cd $NCS_DIR/bin
$ pwd
/usr/local/nso/bin
$ cp ncs-start-python-vm my-start-python-vm
$ # Use your favourite editor to update the last lines of the new
$ # file to start the desired Python executable.
```

And add the following snippet to `ncs.conf`:

```
<python-vm>
  <start-command>/usr/local/nso/bin/my-start-python-vm</start-command>
</python-vm>
```

The new start-command will take effect upon the next restart or configuration reload.

4.14 Operations and maintenance

Because of a different issue, it was suggested to be I should be using the newer version of the XR CLI NED. I finally found it: `ncs-4.1.4.1-cisco-iosxr-6.0.tar`.

I unpacked it in my nso-4.4 packages/neds directory and did a 'make' on it. Then copied it into my nso-run directory. And you did complete rebuild of the XR NED before creating the NETSIMs?

```
make clean all -C packages/cisco-iosxr/src
```

The error indicates that the YANG modules are compiled with a different NCSC than what you're running.

4.15 Configuration file

Inside /etc/ncs you can 'cat ncs.conf' and see the configuration. To debug stuff:

```
<hide-group>
  <name>full</name>
</hide-group>
```

Then on the CLI:

```
ncs cli -u admin
unhide debug
show configuration progress
```

Some other commands, the level 'debug' should be carefully used, while 'very-verbose' can be activated with no problems in a production environment:

```
set verbosity [level]
set destination format [log|csv|...]
```

4.16 Complex checks on YANG models

It seems to be **impossible** to insert complex checks on the parameters defined in a yang model, even for simple stuff like creating a yang model to select on a specific device among disabled interfaces with no description. This is true also for many other network resources (ip addresses, vlans, rt, rd, ...). A possible solution seems to be that of delegating python for EVERY consistency check and resource choice necessary for the service ... even the interface itself should be selected by Python and not a user through the GUI (potentially receiving continuously error messages of 'interface already used'). A prerequisite to perform any kind of choice, would be the **re-sync** of the configuration, which should be up to date. This could lead to some other problems with the way NSO manages the processes, see the following blog post:

"I'm trying to perform a sync from on a device before the instantiation of a python based service. I'm using the cb_pre_modification to do it with this code:

```
@service.pre_modification
def cb_pre_modification(self, tctx, op, kp, root, proplist):
    self.log.info('Service premod(service=', kp, ')')
    self.log.info('Checking device {} is in sync'.format('svlngen4-fab6-
dmzdc-02-fw1'))
    with ncs.maapi.single_write_trans('admin', 'python') as trans:
        root = ncs.maagic.get_root(trans)
        device = root.devices.device['svlngen4-fab6-dmzdc-02-fw1']
        # check_sync_output = device.check_sync.request()
        sync_output = device.sync_from.request()
        self.log.info('Device has now been sync\'d:
{}'.format(sync_output.result))
```

However, the config never gets pushed to the device. I believe that we're getting some sort of timeout, as there's 2 minutes from the time the sync-from is started until it ends:

2017-09-28 07:28:19 - fw_acl_03 - INFO - Checking device svlngen4-fab6-dmzdc-02-fw1 is in sync

2017-09-28 07:30:21 - fw_acl_03 - INFO - Device has now been sync'd: True

In the meantime the transaction times out and not configuration is pushed. The next time I try to commit something on that service, I get:

No registration found for callpoint fw_acl_03-servicepoint/service_create of type=external

I have to reload the packages to clear this error. Is this the right approach to performing a sync-from before committing config changes to a device ?

Answer:

The above is due to a classic **deadlock situation**. When a transaction is started, various locks are taken. At some stage the preMod callback runs. It tries to do a sync-from which in turn tries to grab the same locks. In short, we cannot do updates to CDB (sync-from) while a transaction is in progress.”

Some designs require a certain parameter to be unique inside the whole network, which would require ALL the configurations to be re-synced and ‘locked’ ... quite a problem, right ? This could potentially lead to having a global database with such critical parameters, how should it be kept always up to date ?

```
def print_interfaces(device_name, interface_type):
    with ncs.maapi.single_write_trans('admin', 'python', groups=['ncsadmin']) as t:
        root = ncs.maagic.get_root(t)
        device = root.devices.device[device_name]
        for interface in device.interface[interface_type]:
            print (interface.name)
```

If you forget about the sync problem, this is probably the right way to approach the described problem: more specifically read the following post on Cisco community:

I've developed for a service in Python a validation check. It will validate if the service.vlan_id is already in use on multiple devices which are added to a list inside NSO.

Because the network is a brownfield environment we want to have this check in place so we are certain that the vlan_id number is not used on any subinterfaces on routers or switches. Otherwise the service would overwrite existing network configurations which will disrupt production traffic.

The check will raise an exception that the vlan_id is used on device X on port Y.

The python check works ok when creating new instances of the service. When I preconfigure a subinterface on device X with dot1q 100, and I want to create a new instance of the service, it reports that vlan_id 100 is already configured on device X.

But when I want to use the check-sync or re-deploy functionality, the check will break this functionality. If I create a valid service for customerA with vlan_id 200, the check-sync will report that vlan_id 200 is in use on all devices that are in scope of the service.

I previously build it also under the @ service.create callback, I thought maybe it will work under @ service.pre_modification but without any result.

Python needs to know if the service is being checked for sync or that the service is being redeployed. If Python is aware if this, I can build an IF statement so python will skip the check in those cases.

I need to check what value 'op' has:

```
@Service.pre_modification
def cb_pre_modification(self, tctx, op, kp, root, proplist):
```

op == 0 when doing a dry-run or a commit.

op == 1 when doing a **re-deploy** (reconcile), **check-sync**

op == 2 when deleting the service

The solution is:

```
@Service.pre_modification
def cb_pre_modification(self, tctx, op, kp, root, proplist):
    self.log.info('Service premod(service=', kp, ')')
    if op == 0:
        service = ncs.maagic.cd(root, kp)
        < do some checks over here >
```

4.16.1 Retrieving data from configs

Just as with ACI, you can potentially **move** some of the **checks inside** the **xlm configuration** template files, specifying the operation type create, read, update, delete. If you specify a 'create' operation for that object in the configuration, and the object already exists, the whole transaction will be rejected and you will get an error:

Tags = create

You gather the root object, from there you can cycle through devices and/or devices groups using the '.' as a reference for the key of a dictionary, surfing through the whole data, and the branches of the tree we are interested in. For example the device.name value can be used to refer to a specific configuration:

```
with ncs.maapi.single_write_trans('admin', 'python', groups=['ncsadmin']) as t:
    root = ncs.maagic.get_root(t)
    for box in root.devices.device:
        print(box.name,": ", box.platform.model)
        for acl in root.devices.device[box.name].config.ip.access_list.extended.
ext_named_acl:
            print(acl.name)
```

An example from the configuration of a router of SNAM IP backbone:

```
<config xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>BBIP-BOR-PE-01</name>
      <address>172.16.2.35</address>
      <ssh>
        <host-key>
          <algorithm>ssh-rsa</algorithm>
          <key-data>...</key-data>
        </host-key>
      </ssh>
```

Other examples to retrieve interfaces descriptions and ip addresses:

```
def print_interfaces(device_name, interface_type):
    with ncs.maapi.single_write_trans('admin', 'python', groups=['ncsadmin']) as
t:
        root = ncs.maagic.get_root(t)
        device = root.devices.device[device_name]
        for interface in device.interface[interface_type]:
            print (interface.name)
```

(prints 0/0, 0/1, 0/2, ...)

```
if_types = ['GigabitEthernet', 'Vlan']

for if_type in if_types:
    ifs = [x.name for x in list(device.config.ios__interface[if_type])]
    print(if_type)
    print(ifs)

-----
-----

import ncs
import ncs.maagic as maagic
import ncs.maapi as maapi

with ncs.maapi.single_read_trans('admin', 'system', db=ncs.OPERATIONAL) as m:
    root = ncs.maagic.get_root(m)
    intf_branch = root.devices.device['device_name'].config.ios__interface
    for intf_type in intf_branch:
        for intf in intf_branch[intf_type]:
            print (intf.name)
            print (intf.description)
            print (intf.ip.address.primary)
            ← list of interfaces items
```

4.16.2 Reconciling configs example

Step(1) Log into the service node and create the VRF service

```
ncs_cli -u admin

admin@srv-nso> config
Entering configuration mode private
[ok] [2018-06-26 10:55:56]

[edit]
admin@srv-nso% load merge VRF.xml
[ok] [2018-06-26 10:52:30]

[edit]
admin@srv-nso% show services vrf
vrf TEST-1 {
  description "NSO created VRF";
  devices xr0 {
    route-distinguisher 6500:100;
    import-route-target 6500:101;
    import-route-target 6500:102;
    import-route-target 6500:103;
    import-route-policy TEST-1-IMPORT;
    export-route-policy TEST-1-EXPORT;
  }
  devices xr1 {
    route-distinguisher 6500:200;
    import-route-target 6500:201;
    import-route-target 6500:202;
    import-route-target 6500:203;
    import-route-policy TEST-1-IMPORT;
    export-route-policy TEST-1-EXPORT;
  }
}
}
← on local configs
```

```
[ok][2018-06-26 10:52:34]
```

```
[edit]
```

```
admin@srv-nso% commit
```

← on-field configuration

```
Commit complete.
```

```
[ok][2018-06-26 10:52:52]
```

```
[edit]
```

```
admin@srv-nso%
```

Step(2) Log into nso-1 and modify the service configuration:

```
% ncs_cli -u admin -P 4692
```

```
admin connected from 127.0.0.1 using console on DANISULL-M-73NJ
```

```
admin@nso-1>
```

← configuring nso-1 device, manually

```
admin@nso-1> configure
```

```
Entering configuration mode private
```

```
[ok][2018-06-26 11:13:44]
```

```
[edit]
```

```
admin@nso-1% load merge MAX.xml
```

```
[ok][2018-06-26 11:13:55]
```

```
[edit]
```

```
admin@nso-1% load merge ROUTE-TARGET.xml
```

```
[ok][2018-06-26 11:14:02]
```

```
[edit]
```

```
admin@nso-1% show | compare
```

```
devices {
  device xr0 {
    config {
      cisco-ios-xr:vrf {
        vrf-list TEST-1 {
          address-family {
            ipv4 {
              unicast {
                import {
                  route-target {
                    address-list 6500:300 {
+
+
                    }
                  }
                }
                maximum {
                  prefix {
+
+
                    limit 1000;
                    mid-thresh 50;
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
[ok][2018-06-26 11:14:08]
```

```
[edit]
```

```
admin@nso-1% commit
```



```
Commit complete.  
[ok][2018-06-26 11:16:48]
```

```
[edit]  
admin@nso-1%
```

Step(3) Verify the service is now out of sync

```
admin@srv-nso> request services vrf TEST-1 check-sync  
Error: Network Element Driver: device nso-1: out of sync  
[error][2018-06-26 11:35:27]  
admin@srv-nso> *** ALARM out-of-sync: Device nso-1 is out of sync  
admin@srv-nso> request devices sync-from          ← copy remote cfg to local cfg  
    sync-result {  
        device nso-1  
        result true  
    }  
    sync-result {  
        device nso-2  
        result true  
    }  
[ok][2018-06-26 11:35:36]  
admin@srv-nso> request services vrf TEST-1 check-sync  
in-sync false  
[ok][2018-06-26 11:35:41]  
admin@srv-nso> request services vrf TEST-1 re-deploy dry-run  
cli {  
    lsa-node {  
        name nso-1  
        data devices {  
            device xr0 {  
                config {  
                    cisco-ios-xr:vrf {  
                        vrf-list TEST-1 {  
                            address-family {  
                                ipv4 {  
                                    unicast {  
                                        import {  
                                            route-target {  
-                                     address-list 6500:300 {  
-                                     }  
                                        }  
                                    }  
                                    maximum {  
                                        prefix {  
-                                     limit 1000;  
-                                     mid-thresh 50;  
                                        }  
                                    }  
                                }  
                            }  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

Step(4) Execute the out-of-band reconcile command

```
[ok] [2018-06-26 11:35:53]
admin@srv-nso>
admin@srv-nso> request services vrf TEST-1 oob-reconcile    ←registered Action
status success
[ok] [2018-06-26 11:39:25]
admin@srv-nso>
System message at 2018-06-26 11:39:25...
Commit performed by admin via tcp using OOB-REC-TEST-1.
admin@srv-nso>
```

Step(5) Display the updated service configuration

Display the service configuration and verify the newly reconciled values are now part of the service configuration:

```
admin@srv-nso> show configuration services vrf TEST-1
description "NSO created VRF";
devices xr0 {
    route-distinguisher 6500:100;
    import-route-target 6500:101;
    import-route-target 6500:102;
    import-route-target 6500:103;
    import-route-target 6500:300;          <=== Added
    import-route-policy TEST-1-IMPORT;
    export-route-policy TEST-1-EXPORT;
    max-prefix-limit 1000;                <=== Added
    max-prefix-threshold 50;              <=== Added
}
devices xr1 {
    route-distinguisher 6500:200;
    import-route-target 6500:201;
    import-route-target 6500:202;
    import-route-target 6500:203;
    import-route-policy TEST-1-IMPORT;
    export-route-policy TEST-1-EXPORT;
}
[ok] [2018-06-26 11:39:56]
admin@srv-nso>
```

The reconcile operation is complete. You can also cause the reconcile operation to fail by set the description field in the VRF which isn't automatically reconciled. (load merge DESC.xml on the device node and repeat the above steps). You change the service configuration for the specific vrf by gathering data from field configurations. Follows hereafter the yang file with reconciliation data and the action added:

```
module vrf {
    namespace "http://example.com/vrf";
    prefix vrf;

    import ietf-inet-types {
        prefix inet;
    }
    import tailf-common {
        prefix tailf;
    }
    import tailf-ncs {
        prefix ncs;
    }
}
```

```

description
    "VRF Service";
revision 2018-06-01 {
    description
        "Initial revision.";
}

////////////////////////////////////////
// Service
////////////////////////////////////////

list vrf {
    description "VRF Service";
    key name;
    leaf name {
        tailf:info "Unique service id";
        tailf:cli-allow-range;
        type string;
    }

    uses ncs:service-data;
    ncs:servicepoint vrf-servicepoint;

    leaf description {
        type string;
    }

    list devices {
        key name;

        leaf name {
            type string;
        }

        leaf route-distinguisher {
            type string;
        }
        list import-route-target {
            key name;
            leaf name {
                type string;
            }
        }
        list export-route-target {
            key name;
            leaf name {
                type string;
            }
        }
        leaf import-route-policy {
            type string;
        }
        leaf export-route-policy {
            type string;
        }
        leaf max-prefix-limit {
            type uint16;
        }
        leaf max-prefix-threshold {
            type uint16;
        }
    }
}

```

```

////////////////////////////////////
// Service oob-reconcile action
////////////////////////////////////
tailf:action oob-reconcile {           ← specific action for the service
    tailf:actionpoint vrf-reconcile-point;
    input {
        leaf attach {
            type empty;
        }
    }
    output {
        leaf status {
            type string;
        }
        leaf error-message {
            type string;
        }
    }
}
}
}

```

4.16.3 Sending notifications

Python script to send notifications to NSO:

```

csocket = socket.socket()
try:
    ctx = dp.init_daemon('send-notif')
    # making a control socket
    dp.connect(dx=ctx, sock=csocket, type=dp.CONTROL_SOCKET, ip='127.0.0.1',
port=4571)
    # getting all the required hashes
    ns_hash = _ncs.str2hash("http://cisco.barclays.com/rfs-notification")
    notif_name_hash = _ncs.str2hash('name')
    notif_id_hash = _ncs.str2hash('notif-id')
    mynotif_hash = _ncs.str2hash('my-notif')
    # making the notification
    message = []
    message += [_ncs.TagValue(_ncs.XmlTag(ns_hash, mynotif_hash),
_ncs.Value(mynotif_hash, ns_hash), _ncs.C_XMLBEGIN)]
    message += [_ncs.TagValue(ncs.XmlTag(ns_hash, notif_name_hash),
_ncs.Value('notif-name'))]
    message += [_ncs.TagValue(ncs.XmlTag(ns_hash, notif_id_hash),
_ncs.Value('notif-id'))]
    message += [_ncs.TagValue(_ncs.XmlTag(ns_hash, mynotif_hash),
_ncs.Value(mynotif_hash, ns_hash), _ncs.C_XMLEND)]
    # registering the stream
    livectx = dp.register_notification_stream(ctx, None, csocket, "mystream1")
    # time
    now = datetime.now(tzlocal())
    time = _ncs.DateTime(now.year, now.month, now.day, now.hour, now.minute,
now.second, now.microsecond, now.timetz().hour, now.timetz().minute)
    # sending the notification
    dp.notification_send(livectx, time, message)
except Exception as e:
    self.log.info("Exception : " + e.message)
finally:
    csocket.close()

```

Another way to do it:

```

import ncs, _ncs
from ncs.application import Service
import ncs.experimental
from _ncs import dp
import socket
import datetime
from ncs import maapi

#maapi needs to get schemas, to make str2hash work.
maapi = maapi.Maapi()

mysocket = socket.socket()

try:
    ctx = dp.init_daemon('send-notif')
    # making a socket
    dp.connect(dx=ctx, sock=mysocket, type=dp.CONTROL_SOCKET, ip='127.0.0.1',
port=_ncs.PORT)
    # getting all the required hashes
    ns_hash = _ncs.str2hash("http://dmytro.shytyi.net/netconf-notification-
example")
    notif_id_hash = _ncs.str2hash('state')
    mynotif_hash = _ncs.str2hash('netconf-notif')

    # making the notification
    message = []
    message += [_ncs.TagValue(_ncs.XmlTag(ns_hash, mynotif_hash),
    _ncs.Value((mynotif_hash, ns_hash), _ncs.C_XMLBEGIN))]
    message += [_ncs.TagValue(ncs.XmlTag(ns_hash, notif_id_hash),
    _ncs.Value("ready"))]
    message += [_ncs.TagValue(_ncs.XmlTag(ns_hash, mynotif_hash),
    _ncs.Value((mynotif_hash, ns_hash), _ncs.C_XMLEND))]
    # registering the stream
    livectx = dp.register_notification_stream(ctx, None, mysocket, "notif-
stream")
    # time
    now = datetime.datetime.now()
    time = _ncs.DateTime(now.year, now.month, now.day, now.hour, now.minute,
    now.second, now.microsecond, now.timetz().hour,
    now.timetz().minute)
    # sending the notification
    dp.notification_send(livectx, time, message)
except Exception as e:
    print (e)
finally:
    mysocket.close()

```

The following open source project on Gitlab is related to exporting telemetry data into an external flex database, reading the data ‘as a stream’ from NSO.

<https://gitlab.com/nso-developer/opentelemetry-exporter>

4.16.4 Python background worker for NSO

This part has been taken from here:

Network Automation ramblings by **Kristian Larsson**

<https://plajjan.github.io/2019-07-25-writing-a-background-worker-for-cisco-nso.html>

The `create()` callback is the primary means of which we get things done in Cisco NSO. NSO is most often used for configuration provisioning and so the `create()` callback, which reacts to changes on a YANG configuration subtree is the perfect tool; new configuration input leads to running of our `create()` code which renders new configuration output that we can push to devices or other services. In the YANG model we use a **servicepoints** for attaching a `create()` callback to a particular subtree in the YANG model. In addition to `create()` servicepoint we also have **actionpoints** which allow us to attach our code to YANG actions. Both servicepoint and actionpoint attach to the YANG model and lets code be executed upon external stimuli, either the request to run an action or the change of configuration. What if you want to decide yourself when something should happen or perhaps execute things at a certain periodicity? That's a job for a background worker which is running continuously. With such a background worker, it would be entirely up to you to shape and form the code of the worker to do whatever you need to accomplish. This post is about implementing such a background worker. It should be noted that there is functionality in NSO to schedule periodic activities in a cron job style but I find it somewhat lacking, not that it's worse than cron but cron just isn't the right tool for everything. You would probably do well in understanding it before deciding on how to solve your specific challenge. Either way, as is common with us technical people the question of why or why not is not the focus of this post. Rather, we want to focus on the **how**. *If* you feel the need for a background worker in NSO, how do you actually go about implementing one?

```
import threading
import time

import ncs
from ncs.application import Service

class BgWorker(threading.Thread):
    def run(self):
        while True:
            print("Hello from background worker")
            time.sleep(1)

class Main(ncs.application.Application):
    def setup(self):
        self.log.info('Main RUNNING')
        self.bgw = BgWorker()
        self.bgw.start()

    def teardown(self):
        self.log.info('Main FINISHED')
        self.bgw.stop()
```

I ripped out the `ServiceCallbacks` class with its `cb_create()` since we don't need that here and instead created a new thread definition called `BgWorker` which is instantiated and started from the `setup()` method of our `Application`. Let's try loading the package by running `request packages reload` on our NCS instance (I'm presuming you know how to start up NCS, put the package in the right place etc).

```
admin@ncs> request packages reload force
```

```
>>> System upgrade is starting.
>>> Sessions in configure mode must exit to operational mode.
>>> No configuration changes can be performed until upgrade has completed.
>>> System upgrade has completed successfully.
reload-result {
```

```
package bgworker
result true
}
[ok] [2019-07-01 13:43:04]
admin@ncs>
```

The only thing our background worker does at this point is print a message once a second. Since they are printed and not logged, they will show up in the main python log of NCS `ncs-python-vm.log`.

```
kll@nuc:~/ncs-4.7.4.2/ncs-run/logs$ tail -f ncs-python-vm.log
<INFO> 1-Jul-2019::13:43:04.534 nuc ncs[11832]: Started PyVM: <<"bgworker">> ,
Port=#Port<0.26560> , OSpid="26111"
<INFO> 1-Jul-2019::13:43:04.535 nuc ncs[11832]: bgworker :: Starting
/home/kll/ncs-4.7.4.2/src/ncs/pyapi/ncs_pyvm/startup.py -l info -f ./logs/ncs-
python-vm -i bgworker
<INFO> 1-Jul-2019::13:43:04.595 nuc ncs[11832]: bgworker :: Hello from
background worker
<INFO> 1-Jul-2019::13:43:05.597 nuc ncs[11832]: bgworker :: Hello from
background worker
<INFO> 1-Jul-2019::13:43:06.598 nuc ncs[11832]: bgworker :: Hello from
background worker
<INFO> 1-Jul-2019::13:43:07.599 nuc ncs[11832]: bgworker :: Hello from
background worker
<INFO> 1-Jul-2019::13:43:08.599 nuc ncs[11832]: bgworker :: Hello from
background worker
```

Et voilà! It's working.

`request packages reload` is the "standard" way of loading in new packages, including loading new packages, loading a newer version of an existing already loaded package as well as unloading package (in which case you have to also provide the `force` as NCS will complain over the removal of a namespace, which it thinks is a mistake). It covers all changes like config template changes, YANG model changes and code changes. It is however quite slow and if you have a lot of packages you will soon be rather annoyed over the time it takes (around 2 minutes with the packages we usually have loaded in my work environment). Code changes are perhaps the most common changes during development as you are changing lines, wanting to get them loaded immediately and then run your code again. There is a **redeploy** command for exactly this purpose which can redeploy the code for a single package. In our case, the package is called `bgworker` and so we can redeploy the code by running `request packages package bgworker redeploy`. It normally runs in a second or so.

Let's try:

```
admin@ncs> request packages package bgworker redeploy
result false
[ok] [2019-07-01 13:48:49]
admin@ncs>
```

uh oh. `result false`, why?

Well, our thread runs a `while True` loop and so it simply doesn't have a way of exiting. Unlike UNIX processes, there is no way to kill a thread. They can't be interrupted through signals or similar. If you want to stop a thread, the thread itself has to cooperate, so in effect what you are doing is to *ask* the thread to shut down. We can still forcibly stop our thread by stopping the entire Python VM for our NCS package, since it is running as a UNIX process and can thus be terminated, which will naturally bring down the thread as well. There is a `request python-vm stop` command in NCS or we can just run `request packages`

reload which also involves restarting the Python VM (restart being a stop of the old version and a start of the new version).

We want to be able to run `redploy` though, so how do we get our background worker to play nice? The requirement is that the work has to stop within 3 seconds or NCS thinks it's a failure.

Using a Python events might be the most natural way:

```
import threading
import time

import ncs
from ncs.application import Service

class BgWorker(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
        self.exit_flag = threading.Event()

    def run(self):
        while not self.exit_flag.wait(timeout=1):
            print("Hello from background worker")

    def stop(self):
        self.exit_flag.set()
        self.join()

class Main(ncs.application.Application):
    def setup(self):
        self.log.info('Main RUNNING')
        self.bgw = BgWorker()
        self.bgw.start()

    def teardown(self):
        self.log.info('Main FINISHED')
        self.bgw.stop()
```

We modify our code a bit, inserting a check on a threading. Event in the main loop and then set the Event externally in the thread `stop()` method. Since we can run `wait()` on the Event with a timeout of 1 second we no longer need the separate `time.sleep(1)` call. We override `__init__()` but since we have to call the overwritten `__init__` we do that by calling `threading.Thread.__init__(self)`. Now running `redploy` works just fine:

```
admin@ncs> request packages package bgworker redploy
result true
[ok][2019-07-01 15:02:09]
admin@ncs>
```

Maybe we should implement the main functionality of our program, to increment the counter, instead of just printing a message. Let's rewrite the `run` method. I've included the full module here but the changes are only in the `run` method.

```
import threading
import time

import ncs
from ncs.application import Service

class BgWorker(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
        self._exit_flag = threading.Event()
```



```

def run(self):
    while not self._exit_flag.wait(timeout=1):
        with ncs.maapi.single_write_trans('bgworker', 'system',
db=ncs.OPERATIONAL) as oper_trans_write:
            root = ncs.maagic.get_root(oper_trans_write)
            cur_val = root.bgworker.counter
            root.bgworker.counter += 1
            oper_trans_write.apply()

            print("Hello from background worker, increment counter from {} to
{}".format(cur_val, cur_val+1))

def stop(self):
    self._exit_flag.set()
    self.join()

class Main(ncs.application.Application):
    def setup(self):
        self.log.info('Main RUNNING')
        self.bgw = BgWorker()
        self.bgw.start()

    def teardown(self):
        self.log.info('Main FINISHED')
        self.bgw.stop()

```

We've added some code where we open a single MAAPI write transaction using `single_write_trans()` which allows us to open both a maapi context, session and transaction all in one call. We use it as a context manager to ensure we close all those resources in case of errors or normal exit. There are three parameters to this call. The first and second are the "authentication" information to the system. All of this is running over a trusted MAAPI session but we can tell it what user we are then running our session as. The `system` user is special and has access to pretty much everything. It doesn't rely on the AAA system and so it is a good candidate for writing these kinds of background workers - if someone messes up the AAA configuration you still don't risk your background workers stopping. The first parameter is a context name. I've found that it's very useful to use a good name (you can use an empty string) since it makes troubleshooting so much easier - this context name shows up in `ncs --status` and other places - if you want to be able to know who is holding a lock, you want to put something useful here. The third parameter is where we say we are only interested in the operational datastore, whereas if we wanted to change any configuration this would have to be `running`, which also is the default so we could just leave out the argument completely.

Once we have a transaction to the operational database we want to find our node, read out its value, add 1 and write it back which is what the following three lines accomplishes:

```

root = ncs.maagic.get_root(oper_trans_write)
cur_val = root.bgworker.counter
root.bgworker.counter += 1
oper_trans_write.apply()

```

Finally we `apply()` the transaction. In the logs we can now see our log message reflecting what it is doing:

```

<INFO> 1-Jul-2019::15:11:54.906 nuc ncs[11832]: Started PyVM: <<"bgworker">> ,
Port=#Port<0.34116> , OSpid="32328"
<INFO> 1-Jul-2019::15:11:54.906 nuc ncs[11832]: bgworker :: Starting
/home/kll/ncs-4.7.4.2/src/ncs/pyapi/ncs_pyvm/startup.py -l info -f ./logs/ncs-
python-vm -i bgworker

```

```

<INFO> 1-Jul-2019::15:11:55.956 nuc ncs[11832]: bgworker :: Hello from
background worker, increment counter from 0 to 1
<INFO> 1-Jul-2019::15:11:56.964 nuc ncs[11832]: bgworker :: Hello from
background worker, increment counter from 1 to 2
<INFO> 1-Jul-2019::15:11:57.977 nuc ncs[11832]: bgworker :: Hello from
background worker, increment counter from 2 to 3
<INFO> 1-Jul-2019::15:11:58.982 nuc ncs[11832]: bgworker :: Hello from
background worker, increment counter from 3 to 4
<INFO> 1-Jul-2019::15:11:59.997 nuc ncs[11832]: bgworker :: Hello from
background worker, increment counter from 4 to 5
<INFO> 1-Jul-2019::15:12:01.007 nuc ncs[11832]: bgworker :: Hello from
background worker, increment counter from 5 to 6

```

And if we go look at the value through the CLI we can see how it is being incremented:

```

admin@ncs> show bgworker counter
bgworker counter 845
[ok] [2019-07-01 15:26:08]
admin@ncs>

```

Success!

If we redeploy the `bgworker` package or reload all packages, the worker would continue incrementing the counter from where it left off. This is because we only restart the Python VM while NCS is still running and since the value is stored in CDB, which is part of NCS, it will not go back to the default value of 0 unless we restart NCS.

Let's clean up our code a bit. Instead of printing these messages to stdout we want to use standard Python logging (well, it's actually overridden by an NCS logging module but it acts the same, just allowing reconfiguration from within NCS itself). We want to hide this background thread and just make it look like our application is printing the messages and so we pass the log object down (you can do it in other ways if you want to):

```

# -*- mode: python; python-indent: 4 -*-
import threading
import time

import ncs
from ncs.application import Service

class BgWorker(threading.Thread):
    def __init__(self, log):
        threading.Thread.__init__(self)
        self.log = log
        self._exit_flag = threading.Event()

    def run(self):
        while not self._exit_flag.wait(timeout=1):
            with ncs.maapi.single_write_trans('bgworker', 'system',
db=ncs.OPERATIONAL) as oper_trans_write:
                root = ncs.maagic.get_root(oper_trans_write)
                cur_val = root.bgworker.counter
                root.bgworker.counter += 1
                oper_trans_write.apply()

                self.log.info("Hello from background worker, increment counter from
{} to {}".format(cur_val, cur_val+1))

    def stop(self):
        self._exit_flag.set()

```

```

        self.join()

class Main(ncs.application.Application):
    def setup(self):
        self.log.info('Main RUNNING')
        self.bgw = BgWorker(log=self.log)
        self.bgw.start()

    def teardown(self):
        self.log.info('Main FINISHED')
        self.bgw.stop()

```

And looking in the log `ncs-python-vm-bgworker-log` (notice the package name `bgworker` in the file name) we see how it is now logging there as expected:

```

<INFO> 01-Jul-2019::15:30:06.582 bgworker MainThread: - Python 2.7.16 (default,
Apr  6 2019, 01:42:57) [GCC 8.3.0]
<INFO> 01-Jul-2019::15:30:06.582 bgworker MainThread: - Starting...
<INFO> 01-Jul-2019::15:30:06.583 bgworker MainThread: - Started
<INFO> 01-Jul-2019::15:30:06.602 bgworker ComponentThread:main: - Main RUNNING
<INFO> 01-Jul-2019::15:30:07.607 bgworker Thread-5: - Hello from background
worker, increment counter from 1061 to 1062
<INFO> 01-Jul-2019::15:30:08.620 bgworker Thread-5: - Hello from background
worker, increment counter from 1062 to 1063
<INFO> 01-Jul-2019::15:30:09.624 bgworker Thread-5: - Hello from background
worker, increment counter from 1063 to 1064
<INFO> 01-Jul-2019::15:30:10.628 bgworker Thread-5: - Hello from background
worker, increment counter from 1064 to 1065

```

Now that we've opened a transaction towards CDB there is one issue we will inevitable face. The running datastore has a global lock and while there are no locks on the operational datastore, applying a transaction can still take some time. For example, in a HA cluster the operational data is synchronously replicated and if other nodes are busy or there are other things ahead of us queued up, it can take some time to apply a transaction. Remember that we have to exit in three seconds. The way we structured our code, we read the `self._exit_flag` waiting for up to a second for any values to happen, then we open the transaction and write some data and then we come back to looking at our exit flag again. If we spend more than three seconds in the transaction part of the code we won't observe the exit flag and we will fail to exit in three seconds.

How do we avoid this? How can we leave a guarantee on being able to exit in three seconds?

One solution is to avoid threads altogether and instead use separate processes and this is the route which we will go down. A process can be interrupted by signals like `TERM` or `KILL`, which is the functionality we are after here. Also, David Beazley did an interesting talk on killable threads

<https://www.youtube.com/watch?v=U66KuyD3T0M> which you're encouraged to check out. It's rather interesting... but back to our background worker process!

Python has a very convenient library called `multiprocessing` which is close to a drop in replacement for the `threading` library and as we'll see, we can simplify the code quite a bit since we no longer have to do cooperative shutdown - we can just terminate the background worker process when we want to stop it.

```

# -*- mode: python; python-indent: 4 -*-
import multiprocessing
import time

import ncs
from ncs.application import Service

```

```

def bg_worker(log):
    while True:
        with ncs.maapi.single_write_trans('bgworker', 'system',
db=ncs.OPERATIONAL) as oper_trans_write:
            root = ncs.maagic.get_root(oper_trans_write)
            cur_val = root.bgworker.counter
            root.bgworker.counter += 1
            oper_trans_write.apply()

        log.info("Hello from background worker process, increment counter from
{} to {}".format(cur_val, cur_val+1))
        time.sleep(1)

class Main(ncs.application.Application):
    def setup(self):
        self.log.info('Main RUNNING')
        self.bgw = multiprocessing.Process(target=bg_worker, args=[self.log])
        self.bgw.start()

    def teardown(self):
        self.log.info('Main FINISHED')
        self.bgw.terminate()

```

Much simpler, no? And the result is the same, in fact, since we are passing in the logging object, it is inseparable from the threading solution in the log:

```

<INFO> 01-Jul-2019::21:12:42.897 bgworker ComponentThread:main: - Main RUNNING
<INFO> 01-Jul-2019::21:12:42.905 bgworker ComponentThread:main: - Hello from
background worker process, increment counter from 21271 to 21272
<INFO> 01-Jul-2019::21:12:43.911 bgworker ComponentThread:main: - Hello from
background worker process, increment counter from 21272 to 21273

```

well, I changed the log message slightly so I'd actually see it was from the background worker **process**. What happens if something goes wrong with our worker process? Let's try.

```

def bg_worker(log):
    while True:
        with ncs.maapi.single_write_trans('bgworker', 'system',
db=ncs.OPERATIONAL) as oper_trans_write:
            root = ncs.maagic.get_root(oper_trans_write)
            cur_val = root.bgworker.counter
            root.bgworker.counter += 1
            oper_trans_write.apply()

        log.info("Hello from background worker process, increment counter from
{} to {}".format(cur_val, cur_val+1))
        if random.randint(0, 9) == 9:
            raise ValueError("bad dice value")
        time.sleep(1)

```

so we'll throw our ten sided dice and if we hit 9 we'll throw an error which should lead to termination of the python vm in the background process.

```

kll@nuc:~/ncs-4.7.4.2/ncs-run/logs$ tail -f ncs-python-vm-bgworker.log ncs-
python-vm.log
...
==> ncs-python-vm-bgworker.log <==
<INFO> 01-Jul-2019::21:21:56.770 bgworker ComponentThread:main: - Hello from
background worker process, increment counter from 21804 to 21805

```

```

<INFO> 01-Jul-2019::21:21:57.783 bgworker ComponentThread:main: - Hello from
background worker process, increment counter from 21805 to 21806
<INFO> 01-Jul-2019::21:21:58.788 bgworker ComponentThread:main: - Hello from
background worker process, increment counter from 21806 to 21807
<INFO> 01-Jul-2019::21:21:59.798 bgworker ComponentThread:main: - Hello from
background worker process, increment counter from 21807 to 21808
<INFO> 01-Jul-2019::21:22:00.807 bgworker ComponentThread:main: - Hello from
background worker process, increment counter from 21808 to 21809
<INFO> 01-Jul-2019::21:22:01.824 bgworker ComponentThread:main: - Hello from
background worker process, increment counter from 21809 to 21810
<INFO> 01-Jul-2019::21:22:02.841 bgworker ComponentThread:main: - Hello from
background worker process, increment counter from 21810 to 21811
<INFO> 01-Jul-2019::21:22:03.859 bgworker ComponentThread:main: - Hello from
background worker process, increment counter from 21811 to 21812
<INFO> 01-Jul-2019::21:22:04.873 bgworker ComponentThread:main: - Hello from
background worker process, increment counter from 21812 to 21813
<INFO> 01-Jul-2019::21:22:05.880 bgworker ComponentThread:main: - Hello from
background worker process, increment counter from 21813 to 21814
<INFO> 01-Jul-2019::21:22:06.898 bgworker ComponentThread:main: - Hello from
background worker process, increment counter from 21814 to 21815

```

```
==> ncs-python-vm.log <==
```

```

<INFO> 1-Jul-2019::21:22:06.899 nuc ncs[11832]: bgworker :: Process Process-1:
Traceback (most recent call last):
  File "/usr/lib/python2.7/multiprocessing/process.py", line 267, in _bootstrap
<INFO> 1-Jul-2019::21:22:06.899 nuc ncs[11832]: bgworker ::      self.run()
  File "/usr/lib/python2.7/multiprocessing/process.py", line 114, in run
    self._target(*self._args, **self._kwargs)
  File "/home/kll/ncs-4.7.4.2/ncs-run/state/packages-in-
use/1/bgworker/python/bgworker/main.py", line 19, in bg_worker
    raise ValueError("bad dice value")
ValueError: bad dice value
^C

```

Lo and behold, it did. After this, nothing more happens as our process is dead. If we want the process restarted, we are going to have to do it ourselves. First, we need to monitor for liveness of the process and take action based on that... but before we do that, let's think through some other things that might happen and which we should react to.

Since you are reading this you probably haven't implemented a background worker yet so let me share some advice - add an **off** button. When you are troubleshooting your system it can be rather difficult with lots of things going on, triggered by these background workers. Having multiple background workers both of different type and multiple instances of the same type exacerbate the issue. With an off button we can easily turn them off and troubleshoot the interesting parts. It might seem crude, and I think it is, but in lack of better instrumentation in NCS, it is the best we have.

The most intuitive way of doing this, and the way I've done it so far, is to simply add some configuration that controls whether the background worker is enabled or not. Going back to our YANG model, we add an `enabled` leaf to control if the worker is enabled or not.

```

module bgworker {
  namespace "http://example.com/bgworker";
  prefix bgworker;

  container bgworker {
    leaf enabled {
      type boolean;
      default true;
    }

    leaf counter {

```

```

    config false;
    type uint32;
    default 0;
}
}
}

```

4.16.4.1 1.9 Reacting to HA events

Finally, we have to react to High Availability (HA) events. Depending on which type of worker we are implementing we might want different behaviour. I've so far only had to deal with background workers that write configuration and since that can only be done on the master of a HA system, our background worker should only run on the master node. If you on the other hand are operating on some other data or perhaps not writing anything to CDB, it is possible to still run the worker on all nodes. Assuming you only want to run on the HA master we have to determine;

- if HA is enabled
- what the HA mode is

Getting HA mode is quite simple, it's available from `/ncs:ncs-state/ha/mode`.

I wrote this simple decision table for the behaviour we are looking for:

HA enabled	mode	run worker?
enabled	master	true
enabled	slave	false
enabled	none	false
disabled	none	true

The sort of tricky thing is that when we are in mode `none` we should either run or not depending on if the whole HA functionality is enabled or not, which means we need to look at both. `/ncs:ncs-state/ha` is a presence container and is only present when HA is enabled, thus allowing us to determine if HA is enabled or not.

Another problem around HA event monitoring is that the `/ncs:ncs-state/ha` path isn't in CDB oper as one might have thought, it is actually data provider (DP) backed meaning that we can't use the CDB subscriber design pattern to listen to events. Instead there is a new API that was introduced with NCS 4.7.3 that allows us to subscribe to various events. I'm not sure how I feel about this because one of the strengths of NCS was the YANG modeled nature of everything and that's been effectively abandoned here in benefit of some other interfaces. I've written code that repetitively reads from the `/ncs:ncs-state/ha` path but as it turns out, it's not very fast, probably due to **the DP simply not being very fast**. We should avoid hammering this path with reads and instead try to subscribe to changes.

4.16.4.2 1.10 Rube Goldberg

Okay, so we've gathered all our requirements and are ready to write, as we will see, the Rube Goldberg of NSO background worker process frameworks! To sum up, we want:

- react to NCS package events (redeploy primarily)
- react to the background worker dying (supervisor style)
- react to changes of the configuration for our background worker (enabled or not)
- react to HA events

The basic challenge is that we have multiple different data sources we want to read and monitor but they come in different shape and form. For example, we can write some code that listens for HA events:

```
mask = events.NOTIF_HA_INFO
event_socket = socket.socket()
events.notifications_connect(event_socket, mask, ip='127.0.0.1',
port=ncs.NCS_PORT)
while not self._exit_flag.wait(timeout=1):
    notification = events.read_notification(event_socket)
```

The standard way of monitoring say multiple sockets would be by using a select loop, but then everything has to be a socket. While the HA event socket is, the CDB subscriber is not nor is the main queue we use to signal events. Instead we end up in some form of loop where we need to run various read or wait calls on the things we want to monitor. If we do that using non-blocking calls on all the things it means we will busy loop, which is bad due to CPU usage. If we do blocking calls with a timeout on at least one item, then it means we are blocking on item X while an event could come in on item Y. Maybe the sleep isn't long enough to make it a real problem but it's not an elegant solution and means we are bound to always (statistically) wait for some time before reacting to events. We'll solve all this by defining multiple cooperating pieces:

- a worker that is running as its own UNIX process through the multiprocessing library
- a supervisor thread that starts and stop the worker process
 - the supervisor has a queue over which it receives events from other components
 - it also monitors the process itself merely checking if the worker process is alive and restarts it if not
- a CDB subscriber for monitoring the configuration of the background worker (if it's enabled or not) and puts these as messages on the supervisor queue
- a HA event listener thread that subscribes to HA mode changes and notifies the supervisor through the supervisor queue

It's only the worker process that is an actual UNIX process as I believe we can write all the other components in a way that allows them to exit in a guaranteed time. We ended the first part with a chunk of code that implements all the functionality we had listed as our requirements. You'll find the code [in this commit](#) in case you want to look at it in more detail and how it then evolved, which is what I'll detail in this post.

First thing I did, after having played around with this code myself for some time, was to hand it over to my colleague Marko. Marko just happened to be implementing an NSO component that needed to run in the background and so he was in need of the background_process design that I had been working on. I couldn't find a better beta tester than Marko, as he was in fact involved in the design of this background worker process pattern and so was familiar with the whole idea but hadn't, up to this point, written any code on it. I expected to get some feedback on if the interface was any good or how documentation was lacking. Instead he comes back saying NSO *hangs*.

[4.16.4.3 1.1 Debugging a hung Python VM](#)

At work we have a rather sleek way of starting up a development environment for NSO. We call it a *playground* and it consists of one or more containers running NSO and a number of virtual routers or other support containers that together form a small topology. Starting a playground is just a matter of executing one script, waiting a few minutes (both NSO and virtual routers are slow in starting up) and then you can start coding. There are different topologies you can choose from as well as options for controlling persistence of data or restarting playgrounds until tests fail (for finding bug reproduction cases). We start the same topologies in CI for testing. All in all it gives a rather good guarantee that the environment we develop and test in are all uniform.

When Marko said his NSO had hung I was perplexed as it had worked so well for me. We started looking into differences but there were none, or very few. We ran the same playground topology using the same code versions and we were even running it on the same physical machine. Yet mine worked fine, being able to consistently redeploy the bgworker package and his didn't. It was clear we needed to find out what was going on and fix it.

To reproduce the hang, using the bgworker [as it looked then](#) all that is required is to redeploy the package a few times and you are likely to encounter the hang. The background worker would periodically log a message and when it was hung there would simply be no message. We tried adding print statements but it made little difference. We tried to instead write some data to a temporary file but it didn't work.

Increasing the log level to debug from a lower level seemed to increase the likelihood of the hang but we still didn't understand why. We added a stack dumper and a signal handler so when we sent a USR1 signal we would dump the stack to a file which we could inspect to understand what the program was doing. The stack:

```
File "/usr/lib/python3.5/logging/__init__.py", line 1487, in callHandlers
    hdlr.handle(record)
```

```
File "/usr/lib/python3.5/logging/__init__.py", line 853, in handle
    self.acquire()
```

```
File "/usr/lib/python3.5/logging/__init__.py", line 804, in acquire
    self.lock.acquire()
```

Finally some progress. We repeated this, taking a stack dump for multiple invocations when Python had hung and they all showed the same stack. We were waiting on acquiring a lock!

[4.16.4.4 A ten year old bug](#)

The stack dump further reveals that the lock is being acquired in the logging module, which simply takes a lock around the output in the handler when it is emitting a log message. I didn't actually know the logging module took locks - mostly because I never thought about how it worked internally - but it's fairly natural since we would otherwise risk two log messages emitted at the same time overwriting each other.

The bgworker uses a mix of logging, threads and multiprocessing. As it turns out, [there is a bug related to this exact mix](#) that Marko found. **The bug is ten years old and hadn't yet been fixed.**

The multiprocessing module uses fork to create a child process. `fork()` works by creating a copy of itself, the running process, so that there are two processes after the `fork()` call, one parent and one child process. The child process thus has a copy of all the file handles, sockets and other things in memory.. and if a lock happened to be taken while the child was forked then the lock would be held in both the parent and child, although it is at this time different locks as they had been copied.

Just using multiprocessing and logging is safe since a single threaded program will only do one thing at a time; it will either be logging a message or starting a new child process through fork. As we also mix in the threading module and run multiple threads there is a possibility that we will be forking at the exact moment when a separate thread is logging a message and then the lock will be held! In the parent process the thread emitting a log message will release the lock as soon as it is done but in the child process that never happens since the logging thread isn't running there and the lock is thus held indefinitely. When we then try to log something in the child we first try to acquire the lock but since it is indefinitely held, our Python VM *hangs*.

When bgworker starts up, it is starting multiple threads and forks off the child all around the same time. It also logs quite a few things on startup which means the likelihood of forking while emitting a log message is actually quite high. Raising the log level to debug naturally increases the likelihood even more.

It now also became clear why this hadn't manifested itself for me as I was developing bgworker in a standalone instance whereas Marko was integrating it into a larger package that also ran a number of other threads - also emitting log messages on startup, thus increasing the likelihood of a hang.

4.16.4.5 Incredible timing

Note that the version of bgworker that we are talking about was committed on the 5th of July. After that it took us a few days, till the 8th, to hunt this bug down. 8th of July is also the day that Python 3.7.4rc2 was released and 3.7.4rc2 should address this issue!

We find a bug that is over ten years old and on the day we find it, a fixed version of Python is released. What are the odds?

4.16.4.6 Revamping the NSO docker container

We run NSO in a docker container so switching Python version is relatively simple, but we still have various components and NEDs that include a couple too many dependencies, which does make it trickier to upgrade. We also try to rely on official packages from the distribution repositories rather than pulling down and compiling our own Python build... but we wanted to make progress, so that had to go out the window. Marko and I each rewrote the Dockerfile producing our NSO image, getting multiple alternatives of how we could get a newer Python version in there. We tried pyenv, manual install, some PPA (which wasn't on 3.7.4 yet but we figured we could have waited a few days).

In the end however, it turns out **3.7.4 doesn't really fix our issue**. While it does bring improvements it ultimately does not address our issue. More code would be needed for that (you can't just remove the locks - so trying to log to the same file from what is now two UNIX processes would instead require inter-process locks). Anyway, we needed to fix the real problem through a redesign of our application code.

4.16.4.7 A fresh start

The multiprocessing library uses `fork()` per default on UNIX like operating systems but this can be influenced through an argument and instead of `fork()` we can tell the multiprocessing library to `spawn` which I'm pretty sure maps to `posix_spawn()` under the hood. Unlike `fork()`, `posix_spawn()` doesn't copy all of the memory of the parent to the child process and so we won't get a copy of the held locks - instead we start off fresh. It also means we don't have any loggers at all so we have to set those up. What we want to provide to the user of our background_process micro-framework is a smooth experience and it's just nice if the loggers are already set up for you so you can focus on your own code instead of overhead stuff like logging.

We wanted to continue using the same log files that NCS uses per default, like `ncs-python-vm-bgworker.log`, so that the operations aspect remain the same regardless if you are writing standard Python components for NSO or if you are using the background process design. Since we couldn't write to the same log file from two processes we would have to ship the logs from the child process to the parent process which then could write the messages to the file.

Marko quickly put together a queue listener and emitter so we could send log messages over a multiprocessing Queue. The logging tree in Python is a singleton so we can attach a queue emitting handler in the child process and wrap all this away so that the user of our framework don't have to think about this. We do all this through a wrapping function, like this:

```
def _bg_wrapper(bg_fun, q, log_level, *bg_fun_args):
    """Internal wrapper for the background worker function.
    Used to set up logging via a QueueHandler in the child process. The other end
    of the queue is observed by a QueueListener in the parent process.
    """
    queue_hdlr = logging.handlers.QueueHandler(q)
```

```

root = logging.getLogger()
root.setLevel(log_level)
root.addHandler(queue_hdlr)
logger = logging.getLogger(bg_fun.__name__)
bg_fun(logger, *bg_fun_args)

```

The queue log handler is set up and after this we run the `bg_function` provided to us by the user of the framework.

4.16.4.8 1.6 The promise of efficient logging

Logging can be tricky. We often want to add log messages in various places to easily understand what our program is doing. However, logging itself comes at a cost, not just writing the messages but actually doing the string formatting of them can be relatively expensive. Python's standard logging module is pretty clever and will only format+emit a message if the log level is set high enough. If you have a tight loop and a `log.debug()` statement it won't actually run unless the log level is set to debug. This makes it possible to leave the logging statements in your code and know it will normally run fast. You will only incur a performance penalty when you actually turn on debugging. We can easily show this using a simple program:

```

#!/usr/bin/env python3

import logging
import timeit

log = logging.getLogger()
def noop():
    a = 1

def log_some():
    a = 1
    log.debug("foo")

if __name__ == '__main__':
    print("Noop                                : {}".format(timeit.timeit(noop,
number=100000)))
    log.setLevel(logging.INFO)
    print("Without debug log level  : {}".format(timeit.timeit(log_some,
number=100000)))
    log.setLevel(logging.DEBUG)
    print("With debug log level      : {}".format(timeit.timeit(log_some,
number=100000)))
    ch = logging.FileHandler('foo.log')
    ch.setLevel(logging.DEBUG)
    log.addHandler(ch)
    print("With debug log level + FH: {}".format(timeit.timeit(log_some,
number=100000)))
    log.info("foo")

kll@nuc:~$ python3 slowlog.py
Noop                                : 0.005788944661617279
Without debug log level  : 0.03270535729825497
With debug log level      : 0.7911096690222621
With debug log level + FH: 1.8663266659714282
kll@nuc:~$

```

As we can see, calling a function that doesn't call `log.debug()` at all is vastly faster than calling a function that does call `log.debug()` - it's roughly an order of magnitude. Then enabling the DEBUG log level makes it roughly an order of magnitude slower and finally, actually writing the messages to a file slows it down to about half the speed.

I think of this as a promise to the programmer. You should be able to put log statements in tight loops that need to run fast (obviously not the tightest of loops - there you simply need to strip out your log statements for running in production). The log calls, when the debug logging isn't enabled, should be very very cheap.

Reconfiguring log levels in NCS will only reconfigure the log handler level in the parent process. The child process will remain oblivious. For us to be able to capture all log messages the child process must therefore always emit all log messages, including debug messages, to the queue and then we can filter them away in the parent process in case the current log level doesn't include debug messages.

This is however a rather naive implementation design and it breaks the promise of cheap logging. We need to do better.

4.16.4.9 Log control queue

To uphold the promise of cheap logging in an environment with multiple processes like ours we need to propagate the log level configuration to the child process so it doesn't need to format and emit the log messages unless enabled by the currently configured logging level.

4.17 Writing a Python background worker for Cisco NSO - finale

Always from here:

Network Automation ramblings by **Kristian Larsson**

<https://plaijan.github.io/2019-07-25-writing-a-background-worker-for-cisco-nso.html>

(**NOTE:** this guy is really smart in this kind of stuff ...)

Having read through the previous two parts ([1](#), [2](#)) you know we now have an implementation that actually works and behaves, at least in regards to the most important properties, the way we want. In this last part I would like to explain some of the implementation details, in particular how we efficiently wait for events. All daemons, or computer applications running in the background, typically have a number of things they need to continuously perform. If it's a web server, we need to look for new incoming connections and then perform the relevant HTTP query handling. A naive implementation could repeatedly ask, or *poll*, the socket if there is anything new. If we want to react quickly, then we would have to poll very frequently which in turn means that we will use up a lot of CPU. If there are a lot of requests then this could make sense, in fact many network interface drivers use polling as it is much more efficient than being interrupt based. Polling 1000 times per second means that we will notice anything incoming within a maximum of 1 millisecond and for each poll we can efficiently batch handle all the incoming requests (assuming there are more than 1000 request per second). If there aren't that many incoming requests though, it is probably better finding a pattern where we can sleep and be awoken only when there is a request. This is what interrupts can provide to the CPU. UNIX processes don't have interrupts, instead there are signals which act in a similar way and can interrupt what the program is currently doing. It's often used for stopping the application (KILL signal) or reconfiguring it (HUP signal).

There are good use cases for signals but overall they are not the primary means of driving an application control flow. They are meant as simple inputs to a UNIX process from the *outside*.

Smart people realized many moons ago that we needed ways to efficiently wait for input and this is why we have things like **`select()` and `poll()`** which can efficiently wait for a file descriptor to become readable.

By *efficiently*, I mean it is able to wait for something to happen without consuming a lot of CPU resources yet is able to immediately wake up and return when something has happened. Not only can `select()` efficiently wait for something to happen on a file descriptor, it can wait on **multiple** file descriptors.

Everything in UNIX is a file, which means that sockets have file descriptors too, so we can *wait* on a bunch of sockets and multiple files all using the same `select()` call. This is the basis for many of the daemons

that exist today. Now for the background worker in NCS we have multiple events we want to observe and react to:

- react to NCS package events (redeploy primarily)
- react to the background worker dying (supervisor style)
- react to changes of the configuration for our background worker (enabled or not)
- react to HA events

Are all these sockets or file descriptors that we can wait on with a `select()` call? As it turns out, yes, but it wasn't very obvious from the beginning.

4.17.1 Thread whispering

When a package is redeployed or reloaded, NCS will stop the currently running instance by calling the `teardown()` function of each component thread. This is where we then in turn can call the `stop()` function on the various threads or processes we are running. Thus, the interface here for the incoming data is a function but we then have to propagate this information from our function to our main thread. If you remember from the [first part](#) we had a silly naive implementation of a background worker that looked like this:

```
import threading
import time

import ncs
from ncs.application import Service

class BgWorker(threading.Thread):
    def run(self):
        while True:
            print("Hello from background worker")
            time.sleep(1)

class Main(ncs.application.Application):
    def setup(self):
        self.log.info('Main RUNNING')
        self.bgw = BgWorker()
        self.bgw.start()

    def teardown(self):
        self.log.info('Main FINISHED')
        self.bgw.stop()
```

and since it had no way to signal to the threads `run()` function that it should stop, it would never stop. We quickly realized this and improved it to:

```
import threading
import time

import ncs
from ncs.application import Service

class BgWorker(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
        self._exit_flag = threading.Event()
```

```

def run(self):
    while not self._exit_flag.wait(timeout=1):
        print("Hello from background worker")

def stop(self):
    self._exit_flag.set()
    self.join()

class Main(ncs.application.Application):
    def setup(self):
        self.log.info('Main RUNNING')
        self.bgw = BgWorker()
        self.bgw.start()

    def teardown(self):
        self.log.info('Main FINISHED')
        self.bgw.stop()

```

Where we use a `threading.Event` as the means to signal into the thread `run()` method that we want it to stop. In `run()` we read the `threading.Event` exit flag in a blocking fashion for a second and then perform our main functionality only to return and wait on the `Event`.

Using `wait()` on that `Event` means we can only wait for a single thing at a time. That's not good enough - we have multiple things we need to observe. In the main supervisor thread we replaced this with a queue since we can feed things into the queue from multiple publishers. Something like this (this isn't our actual supervisor function, just an example showing how we would wait on a queue instead):

```

import threading
import time

import ncs
from ncs.application import Service

class BgWorker(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
        self.q = queue.Queue()

    def run(self):
        while True:
            print("Hello from background worker")

            item = self.q.get(timeout=1)
            try:
                item = self.q.get(timeout=1)
            except queue.Empty:
                continue

            if item == 'exit':
                return

    def stop(self):
        self.q.put('exit')
        self.join()

class Main(ncs.application.Application):
    def setup(self):
        self.log.info('Main RUNNING')
        self.bgw = BgWorker()

```

```

self.bgw.start()

def teardown(self):
    self.log.info('Main FINISHED')
    self.bgw.stop()

```

Thus far we have just replaced the `threading.Event` with a `queue.Queue` and unlike the `Event`, which effectively just carries a boolean value, the queue could carry close to anything. We use a simple string value of `exit` to signal the thread that it should stop. Now that we have a queue though, we can put more things on the queue and this is why a queue was picked for the supervisor.

When implementing a thread for monitoring something, the important thing to remember is that the `run()` loop of the thread has to be able to monitor its primary object and be signaled from the `stop()` function using the same method so that it can be efficiently waited upon in the `run()` loop.

4.17.2 CDB configuration changes

CDB subscribers are using a design pattern provided by Cisco and we can't influence it much. Instead we have to integrate with it. With a queue to the supervisor this becomes trivial. The config CDB subscriber simply runs as a separate thread and will take whatever updates it receives on CDB changes and publish them on the queue so the supervisor can react to it.

4.17.3 Monitoring HA events

HA events come from NSO over the notifications API which we access through a socket from Python. Unlike the CDB subscriber, there is no ready to go class that we can just subclass and get moving with. Instead we have to implement the thread `run()` method ourselves. Efficiently waiting on a socket is easy, as already covered, we can use `select()` for this. However, how can we signal the thread to stop using something that is selectable? I chose to implement a **WaitableEvent** that sends its data over a pipe, which has a file descriptor and thus is waitable. The code for that looks like this:

```

class WaitableEvent:
    """Provides an abstract object that can be used to resume select loops with
    indefinite waits from another thread or process. This mimics the standard
    threading.Event interface."""
    def __init__(self):
        self._read_fd, self._write_fd = os.pipe()

    def wait(self, timeout=None):
        rfd, _, _ = select.select([self._read_fd], [], [], timeout)
        return self._read_fd in rfd

    def is_set(self):
        return self.wait(0)

    def isSet(self):
        return self.wait(0)

    def clear(self):
        if self.isSet():
            os.read(self._read_fd, 1)

    def set(self):
        if not self.isSet():
            os.write(self._write_fd, b'1')

    def fileno(self):
        """Return the FD number of the read side of the pipe, allows this
        object to be used with select.select()

```

```

    """
    return self._read_fd

def __del__(self):
    os.close(self._read_fd)
    os.close(self._write_fd)

```

and we can use it much the same way as `threading.Event` since it implements the same interface, however, note how the underlying transport is an `os.pipe` and we thus can use that in our `select()` call simply by digging out `self._read_fd`. Also note that I didn't write the code for this myself. After realizing what I needed I searched and the Internet delivered.

Here is the code for the HA event monitor using a `WaitableEvent`:

```

class HaEventListener(threading.Thread):
    """HA Event Listener
    HA events, like HA-mode transitions, are exposed over a notification API.
    We listen on that and forward relevant messages over the queue to the
    supervisor which can act accordingly.

    We use a WaitableEvent rather than a threading.Event since the former
    allows us to wait on it using a select loop. The HA events are received
    over a socket which can also be waited upon using a select loop, thus
    making it possible to wait for the two inputs we have using a single select
    loop.
    """
    def __init__(self, app, q):
        super(HaEventListener, self).__init__()
        self.app = app
        self.log = app.log
        self.q = q
        self.log.info('{} supervisor: init'.format(self))
        self.exit_flag = WaitableEvent()

    def run(self):
        self.app.add_running_thread(self.__class__.__name__ + ' (HA event
listener)')
        self.log.info('run() HA event listener')
        from ncs import events
        mask = events.NOTIF_HA_INFO
        event_socket = socket.socket()
        events.notifications_connect(event_socket, mask, ip='127.0.0.1',
port=ncs.PORT)
        while True:
            rl, _, _ = select.select([self.exit_flag, event_socket], [], [])
            if self.exit_flag in rl:
                event_socket.close()
                return

            notification = events.read_notification(event_socket)
            # Can this fail? Could we get a KeyError here? Afraid to catch it
            # because I don't know what it could mean.
            ha_notif_type = notification['hnot']['type']

            if ha_notif_type == events.HA_INFO_IS_MASTER:
                self.q.put(('ha-master', True))
            elif ha_notif_type == events.HA_INFO_IS_NONE:
                self.q.put(('ha-master', False))
            elif ha_notif_type == events.HA_INFO_SLAVE_INITIALIZED:
                self.q.put(('ha-master', False))

    def stop(self):

```

```

        self.exit_flag.set()
        self.join()
        self.app.del_running_thread(self.__class__.__name__ + ' (HA event
listener)')

```

It selects on the `exit_flag` (which is a `WaitableEvent`) and the event socket itself. The `stop()` method simply sets the `WaitableEvent`. If `exit_flag` is readable it means the thread should exit while if the `event_socket` is readable we have a HA event.

We use multiple threads with different methods so we can efficiently monitor different *classes* of objects.

4.17.4 Child process liveness monitor

If the process we started to run the background worker function dies for whatever reason, we want to notice this and restart it. How can we efficiently monitor the liveness of a child process?

This was the last thing we wanted to monitor that remained as a half busy poll. The supervisor would wait for things coming in on the supervisor queue for one second, then go and check if the child process was alive only to continue monitoring the queue.

The supervisor `run()` function:

```

def run(self):
    self.app.add_running_thread(self.name + ' (Supervisor)')

    while True:
        should_run = self.config_enabled and (not self.ha_enabled or
self.ha_master)

        if should_run and (self.worker is None or not self.worker.is_alive()):
            self.log.info("Background worker process should run but is not
running, starting")
            if self.worker is not None:
                self.worker_stop()
            self.worker_start()
            if self.worker is not None and self.worker.is_alive() and not
should_run:
                self.log.info("Background worker process is running but should not
run, stopping")
                self.worker_stop()

        try:
            item = self.q.get(timeout=1)
        except queue.Empty:
            continue

        k, v = item
        if k == 'exit':
            return
        elif k == 'enabled':
            self.config_enabled = v

```

This irked me. Waking up once a second to check on the child process doesn't exactly qualify as busy polling - only looping once a second won't increase CPU utilization by much, yet child processes dying should be enough of a rare event that not reacting quicker than 1 second is still good enough. It was a simple and pragmatic solution that was enough for production use. But it irked me.

I wanted to remove the last *busy* poll and so I started researching the problem. It turns out that it is possible, through a rather clever hack, to detect when a child process is no longer alive.

When the write end of a POSIX pipe is in the sole possession of a process and that process dies, the read end becomes readable. And so this is exactly what we've implemented.

- setup a pipe
- fork child process (actually 'spawn'), passing write end of pipe to child
- close write end of pipe in parent process
- wait for read end of pipe to become readable, which only happens when the child process has died

Since a pipe has a file descriptor we can wait on it using our `select()` loop and this is what we do in the later versions of bgworker.

4.17.5 Hiding things from the bg function

We want to make it dead simple to use the background process library. Passing in a pipe, and the logging objects that need to be set up, as described in the [previous part](#), should have to be done by the user of our library. We want to take care of that, but how?

When we start the child process, it doesn't immediately run the user provided function. Instead we have a wrapper function that takes care of these things and then hands over control to the user provided function! Like this:

```
def _bg_wrapper(pipe_unused, log_q, log_config_q, log_level, bg_fun,
                *bg_fun_args):
    """Internal wrapper for the background worker function.

    Used to set up logging via a QueueHandler in the child process. The other
end
    of the queue is observed by a QueueListener in the parent process.
    """
    queue_hdlr = logging.handlers.QueueHandler(log_q)
    root = logging.getLogger()
    root.setLevel(log_level)
    root.addHandler(queue_hdlr)

    # thread to monitor log level changes and reconfigure the root logger level
    log_reconf = LogReconfigurator(log_config_q, root)
    log_reconf.start()

    try:
        bg_fun(*bg_fun_args)
    except Exception as e:
        root.error('Unhandled error in {} - {}: {}'.format(bg_fun.__name__,
type(e).__name__, e))
        root.debug(traceback.format_exc())
```

Note how the first argument, accepting the pipe is unused, but it is enough to receive the write end of the pipe. Then we configure logging etc and implement a big exception handler.

4.17.6 A selectable queue

Monitoring the child process liveness happens through a pipe which is waitable using `select`. As previously described though, we placed a queue at the center of the supervisor thread and send messages from other threads over this queue. Now we have a queue and a pipe to wait on, how?

We could probably abandon the queue and have those messages be sent over a pipe, which we could then `select()` on... but the queue is so convenient!

The `multiprocessing` library also has a queue which works across multiple processes. It uses a pipe under the hood to pass the messages and deals with things like sharing the the file descriptor when you spawn your child process (which is what we do). By simply switching from `queue.Queue` to `multiprocessing.Queue` (they feature the exact same interface) we have gained a pipe under the hood that we can `select()` on. Voilà!

Here's the code for the supervisor thread showing both the selectable queue (well, pipe) and the clever child process liveness monitor. To read the full up to date code for the whole background process library, just head over to [the bgworker repo on Github](#).


```

else:
    # if there is no config_path we assume the process is
always enabled

    self.config_enabled = True

# In the 2nd transaction read operational data regarding HA.
# This is an expensive operation invoking a data provider, thus
# we don't want to incur any unnecessary locks
with m.start_read_trans(db=ncs.OPERATIONAL) as oper_t_read:
    # check if HA is enabled
    if oper_t_read.exists("/tfnm:ncs-state/tfnm:ha"):
        self.ha_enabled = True
    else:
        self.ha_enabled = False

    # determine HA state if HA is enabled
    if self.ha_enabled:
        ha_mode = str(ncs.maagic.get_node(oper_t_read,
'/tfnm:ncs-state/tfnm:ha/tfnm:mode'))
        self.ha_master = (ha_mode == 'master')

def run(self):
    self.app.add_running_thread(self.name + ' (Supervisor)')

    while True:
        try:
            should_run = self.config_enabled and (not self.ha_enabled or
self.ha_master)

            if should_run and (self.worker is None or not
self.worker.is_alive()):
                self.log.info("Background worker process should run but is
not running, starting")
                if self.worker is not None:
                    self.worker_stop()
                self.worker_start()
            if self.worker is not None and self.worker.is_alive() and not
should_run:
                self.log.info("Background worker process is running but
should not run, stopping")
                self.worker_stop()

            # check for input
            waitable_rfds = [self.q._reader]
            if should_run:
                waitable_rfds.append(self.parent_pipe)

            rfd, _, _ = select.select(waitable_rfds, [], [])
            for rfd in rfd:
                if rfd == self.q._reader:
                    k, v = self.q.get()

                    if k == 'exit':
                        return
                    elif k == 'enabled':
                        self.config_enabled = v
                    elif k == "ha-master":
                        self.ha_master = v

            if rfd == self.parent_pipe:
                # getting a readable event on the pipe should mean the
                # child is dead - wait for it to die and start again

```

```

        # we'll restart it at the top of the loop
        self.log.info("Child process died")
        if self.worker.is_alive():
            self.worker.join()

    except Exception as e:
        self.log.error('Unhandled exception in the supervisor thread: {}'.format(
            type(e).__name__, e))
        self.log.debug(traceback.format_exc())
        time.sleep(1)

def stop(self):
    """stop is called when the supervisor thread should stop and is part of
    the standard Python interface for threading.Thread
    """
    # stop the HA event listener
    self.log.debug("{}: stopping HA event listener".format(self.name))
    self.ha_event_listener.stop()

    # stop config CDB subscriber
    self.log.debug("{}: stopping config CDB subscriber".format(self.name))
    if self.config_path is not None:
        self.config_subscriber.stop()

    # stop log config CDB subscriber
    self.log.debug("{}: stopping log config CDB
subscriber".format(self.name))
    self.log_config_subscriber.stop()

    # stop the logging QueueListener
    self.log.debug("{}: stopping logging QueueListener".format(self.name))
    self.queue_listener.stop()

    # stop us, the supervisor
    self.log.debug("{}: stopping supervisor thread".format(self.name))

    self.q.put(('exit', None))
    self.join()
    self.app.del_running_thread(self.name + ' (Supervisor)')

    # stop the background worker process
    self.log.debug("{}: stopping background worker
process".format(self.name))
    self.worker_stop()

def worker_start(self):
    """Starts the background worker process
    """
    self.log.info("{}: starting the background worker
process".format(self.name))
    # Instead of using the usual worker thread, we use a separate process
    here.
    # This allows us to terminate the process on package reload / NSO
    shutdown.

    # using multiprocessing.Pipe which is shareable across a spawned
    # process, while os.pipe only works, per default over to a forked
    # child
    self.parent_pipe, child_pipe = self.mp_ctx.Pipe()

    # Instead of calling the bg_fun worker function directly, call our

```

```

# internal wrapper to set up things like inter-process logging through
# a queue.
args = [child_pipe, self.log_queue, self.log_config_q,
self.current_log_level, self.bg_fun] + self.bg_fun_args
self.worker = self.mp_ctx.Process(target=_bg_wrapper, args=args)
self.worker.start()

# close child pipe in parent so only child is in possession of file
# handle, which means we get EOF when the child dies
child_pipe.close()

def worker_stop(self):
    """Stops the background worker process
    """
    if self.worker is None:
        self.log.info("{}: asked to stop worker but background worker does
not exist".format(self.name))
        return
    if self.worker.is_alive():
        self.log.info("{}: stopping the background worker
process".format(self.name))
        self.worker.terminate()
        self.worker.join(timeout=1)
    if self.worker.is_alive():
        self.log.error("{}: worker not terminated on time, alive: {}
process: {}".format(self, self.worker.is_alive(), self.worker))

```

4.17.7 A library with a simple user interface

As we've gone through over in these three posts, there's quite a bit of code that needs to be written to implement a proper NSO background worker. The idea was that we would write it in such a way that it could be reused. Someone wanting to implement a background worker should not have to understand, much less implement, all of this. This is why we've structured the surrounding code for running a background worker as a library that can be reused. We effectively hide the complexity by exposing a simple user interface to the developer. Using the bgworker background process library could look like this:

```

# -*- mode: python; python-indent: 4 -*-
import logging
import random
import sys
import time

import ncs
from ncs.application import Service

from . import background_process

def bg_worker():
    log = logging.getLogger()

    while True:
        with ncs.maapi.single_write_trans('bgworker', 'system',
db=ncs.OPERATIONAL) as oper_trans_write:
            root = ncs.maagic.get_root(oper_trans_write)
            cur_val = root.bgworker.counter
            root.bgworker.counter += 1
            oper_trans_write.apply()

        log.debug("Hello from background worker process, increment counter from
{} to {}".format(cur_val, cur_val+1))

```

```

        if random.randint(0, 10) == 9:
            log.error("Bad dice value")
            sys.exit(1)
        time.sleep(2)

class Main(ncs.application.Application):
    def setup(self):
        self.log.info('Main RUNNING')
        self.p = background_process.Process(self, bg_worker,
config_path='/bgworker/enabled')
        self.p.start()

    def teardown(self):
        self.log.info('Main FINISHED')
        self.p.stop()

```

This is the example in the [bgworker repo](#) and it shows the simple worker that increments an operational state value once per second. Every now and then it dies, which then shows that the supervisor correctly monitors the child process and restarts it. You can disable it by setting `/bgworker/enabled` to `false`. The main functionality is implemented in the `bg_worker()` function and we use the background process library to run that function in the background.

It is the following lines, which are part of a standard NSO Application definition, where we hook in and run the background process by instantiating `background_process.Process()` and feeding it the function we want it to run. Further we tell it that the path to the enable/disable leaf of this background worker is `/bgworker/enabled`. The library then takes over and does the needful.

```

class Main(ncs.application.Application):
    def setup(self):
        self.log.info('Main RUNNING')
        self.p = background_process.Process(self, bg_worker,
config_path='/bgworker/enabled')
        self.p.start()

    def teardown(self):
        self.log.info('Main FINISHED')
        self.p.stop()

```

We aimed for a simple interface and I think we succeeded.

The idea behind placing all of this functionality into a library is that we hide the complexity from the user of our library. A developer that needs a background worker wants to spend 90% of the time on the actual functionality of the background worker rather than writing the surrounding overhead code necessary for running the background worker function. This is essentially the promise of any programming language or technique ever written - *spend your time on your business logic and not on overhead tasks* - nonetheless I think we accomplished what we set out to do.

4.17.8 Finale

And with that, we conclude the interesting parts of how to implement a background worker for Cisco NSO. I hope you've found it interesting to read about. It was fun and interesting implementing, in particular as it's been a while for me since I was this deep into the low level workings of things. I am a staunch believer that we generally need to use libraries when implementing network automation or other application/business logic so we can focus on the right set of the problems rather than interweaving say the low level details of POSIX processes with our application logic. In essence; using the right abstraction layers. Our human brains can only focus on so many things at a time and using the right abstractions is therefore crucial. Most of the time I mostly deal with high level languages touching high level things - exactly as I want it to be. However,

once in a while, when the framework (NSO in this case) doesn't provide what you need (a way to run background workers), you just have to do it yourself.

I hope you will also appreciate the amount of energy that went into writing the [bgworker](#) package.py library that you can reuse. It is a rewrite of the common set of core functionality of the background workers we already had, resulting in a much better and cleaner implementation using a generic library style interface allowing anyone to use it. If you have the need for running background workers in NSO I strongly recommend that you make use of it. There's a lot of lessons learned here and starting over from scratch with a naive implementation means you will learn all of them the hard way. If bgworker doesn't fit in your architecture then feel free to give feedback and perhaps we can find a way forward together.

4.18 Python checks on CLI output

Hi,

am trying to write an action (I actually want to do this from a service later on, just writing a simple Action to try it out first) which goes to an IOS device and fetches the cdp neighbor infos from it.

I am aware that I can call "devices device ios0 live-status ios-stats:exec show cdp neighbors" or "devices device ios0 config exec "do show cdp neighbors" " (when in configure terminal mode) directly from the NSO CLI.

In my main.py I try the following but already assumed it probably won't work:

```
with ncs.maapi.single_read_trans('admin', 'python') as t:
    root = ncs.maagic.get_root(t)
    neighborinfo = root.devices.device[input.dev]['live-status']['ios-stats:exec
show cdp neighbors']
```

And, as assumed it doesn't work (Error: Python cb_action error. 'ios-stats:exec show cdp neighbors')

I have also tried:

```
neighborinfo = root.devices.device[input.dev]['live-status']['ios-
stats:exec']['show cdp neighbors']
```

That also didn't work. How can I access the info from show commands via the Python API? Or is there a better way altogether to do this?

I think your approach is sensible. But, calling actions can be a little bit tricky sometimes, because the parameters have to be sent in a special format. This is documented in the API documentation and examples if you look carefully, but, here is an example that works where I call the RPC:

```
devs = root.devices.device
show = devs[devname].live_status.__getitem__('exec').show
inp = show.get_input()

# Check OSPF settings
inp.args = ['ip ospf neigh']
r = show.request(inp)
self.log.info('Show OSPF: {}'.format(r.result))
```

It runs "show ip ospf neigh". Hopefully this is a helpful starting point. Two things to note, first we use get_input() to get the input structure that we fill in with arguments. Potentially this could be much more complex than a single "args" string, secondly it uses a trick to get around the fact that exec is a built-in in python3 - there might well be a prettier way around that.

Hi,

thanks for the reply first of all (Hope you had a good flight back from Frankfurt last week).

I am now trying this code (just want to get version for now, but could just as well use 'cdp neighbor' as args), which is almost an exact copy of the code you provided:

```
with ncs.maapi.single_read_trans('admin', 'python') as t:
    root = ncs.maagic.get_root(t)
    devs = root.devices.device
    show = devs[input.dev].live_status.__getitem__('exec').show
    inp = show.get_input()
    inp.args = ['version']
    r = show.request(inp)
    self.log.info('Show Version: {}'.format(r.result))
```

Unfortunately this gives me a key error. In NSO CLI I get: Error: Python cb_action error. 'exec'
I had to change 'exec' to 'ios-stats:exec' ...like this:

```
devs[input.dev].live_status.__getitem__('ios-stats:exec').show
```

And then it worked like a charm, thank you very much!!!

Thank you. I will take a closer look in the Docs. And yes, I have also got the IOS XR NED loaded. So is it normal that I have to call use 'ios-stats:exec' instead of just 'exec' because I have another NED loaded (XR)? It turns out the customer doesn't need the XR NED after all, so should I get rid of it and have my code use 'exec' instead of 'ios-stats:exec' ?

4.19 Python action dry-run commit custom

```
container action {
    tailf:action trystuff {
        tailf:actionpoint tempBar-action;

        input {
            (...skip)
            leaf doDryCommit {
                tailf:info "dryrun or commit";
                type enumeration {
                    enum dry-run;
                    enum commit;
                }
                default "dry-run";
            }
        } // input
    }
}
```

then you Python will look (something) like this:

```
with ncs.maapi.single_write_trans('admin', 'tempBar') as t:
    try:
        vars = ncs.template.Variables()
        vars.add('DEVICE', input.device)
        vars.add('HNAME', input.hname)
        vars.add('DNAME', input.dname)
        #
        root = ncs.maagic.get_root(t)
        context_node = root.ncs__devices
        tpl = ncs.template.Template(context_node)
        tpl.apply('tempBar-template', vars)
```



```

        #....
        # dryrun-native
        #....
    if str(input.doDryCommit) == 'dry-run':
        dryRun = root.services.commit_dry_run
        drInput = dryRun.get_input()
        drInput.outformat = 'native'
        drOutput = dryRun(drInput)
        if drOutput.native != None :
            devlist = drOutput.native.device
            if len(devlist) != 0:
                for dev in devlist:
                    self.log.info('tempBar: Dry-run device : ',
dev.name)
                    self.log.info('tempBar: Dry-run output : ',
dev.data)
            #....
        else:
            self.log.info('tempBar: Dry-run device : ',
dev.name)
            self.log.info('tempBar: Dry-run output : NO Dry-Run
found')

        #....
        # commit
        #....
    else: # commit selected
        self.log.info('tempBar: commit template apply')
        # commit transaction
        t.apply()

        self.log.info('tryAction: Host ... : ', input.hname)
        output.result = "all done"

    except Exception, cause:
        res = str(cause)
        self.log.info('tryAction: Exception: %s ', res)
        output.result = "exception"

    finally:
        t.finish_trans()

```

4.20 Python action callback example

```

class ActionCallbacks(object):
    def __init__(self, log):
        self.log = log
        self.logPrefix = 'ActionCallbacks:'
        self.daemon = Daemon("l3vpn-cli-completion-daemon", log=log)
        self.completion_action = 'if-action'
        register_action_cbs(self.daemon.ctx(), self.completion_action, self)
        self.daemon.start()

    def cb_init(self, uinfo):
        self.log.info(self.logPrefix, 'cb_init')
        self.log.info(self.logPrefix, 'cb_init daemon alive : ',
            self.daemon.isAlive())
        name = 'usid-{0}-{1}'.format(uinfo.usid, self.completion_action)
        key = '{0}-{1}'.format(id(self), uinfo.usid)
        wsock = ncs.dp.take_worker_socket(self.daemon, name, key)
        # dp.action_set_timeout(unifo, 2)
        _tm.dp.action_set_fd(uinfo, wsock)
        self.log.info(self.logPrefix, 'cb_init', ' complete')

```

```

def cb_abort(self, uinfo):
    self.log.info(self.logPrefix, 'cb_abort')

def cb_action(self, uinfo, name, kp, params):
    self.log.info(self.logPrefix, 'cb_action')

def cb_command(self, uinfo, path, argv):
    self.log.info(self.logPrefix, 'cb_command')

def cb_completion(self, uinfo, cli_style, token, completion_char,
                    kp, cmdpath, cmdparam_id, simpleType, extra):
    self.log.info(self.logPrefix,
'cb_completion({}, {}, {}, {}, {}, {}, {}, {}, {}, {})'
        .format(
            uinfo, cli_style, token, completion_char, kp, cmdpath, cmdparam_id,
            simpleType, extra))

    if_list = [(0, 'ge-0/1', None), (0, 'fe-0/0', None),
                (0, 'ge-0/2', None)]
    action_reply_completion(uinfo, if_list)

# -----
# COMPONENT THREAD THAT WILL BE STARTED BY NCS.
# -----
class L3VPN(ncs.application.Application):
    def setup(self):
        self.log.info('L3VPN RUNNING')

        self.acb = ActionCallbacks(self.log)
        self.register_service('l3vpn-servicepoint', ServiceCallbacks)

    def teardown(self):
        self.log.info('L3VPN FINISHED')

```

4.21 REST queries

You can use POSTMAN for this kind of interactions, after having logged in into NSO. The RESTCONF protocol can exchange data in different formats, with JSON and XML being common. RESTCONF servers are not required to support both, but NSO does. You specify the format by using Accept and Content-Type headers. The Accept header instructs NSO to return data in the selected format, while the Content-Type corresponds to the format of the data in the request.

Unlike traditional REST APIs, RESTCONF uses two specific media types: `application/yang-data+json` and `application/yang-data+xml`. If you examine the headers of the last response, you will see the Content-Type is not `application/xml` but is actually `application/yang-data+xml`.

To switch to JSON encoding, modify the Postman request. On the Headers tab, add two new keys: 'Accept' and 'Content-Type'. Set the value for both to 'application/yang-data+json'. Resend the request and observe how the response body changes to JSON format.

```

{
  "ietf-restconf:restconf": {
    "data": {},
    "operations": {},
    "yang-library-version": "2019-01-04"
  }
}

```

<https://sandbox-nso-1.cisco.com/restconf/data/taillf-ncs:devices/device-group>

show running-config devices device-group



GET /restconf/data/tailf-ncs:devices/device-group

namespace

```
GET /restconf/data/tailf-ncs:devices/device-group=IOS-DEVICES

{
  "tailf-ncs:device-group": [
    {
      "name": "IOS-DEVICES",
      "device-name": [
        "dist-rtr01",
        "dist-rtr02",
        "internet-rtr01"
      ],
      "member": [
        "dist-rtr01",
        "dist-rtr02",
        "internet-rtr01"
      ],
      "ned-id": [
        {
          "id": "cisco-ios-cli-6.67:cisco-ios-cli-6.67"
        }
      ],
      "tailf-ncs:alarms:alarm-summary": {
        "indeterminates": 0,
        "criticals": 0,
        "majors": 0,
        "minors": 0,
        "warnings": 0
      }
    }
  ]
}
```

Actually, RESTCONF responses by default include configuration and non-configuration (operational) data. Non-configuration data is different from configuration data in that you can't directly change it. It is provided by the system and usually conveys the current status or state. You would typically use different **show** commands in the NSO CLI to display it.

In effect, the RESTCONF response is composed of data from two commands:

```
admin@ncs# show running-config devices device-group IOS-DEVICES
```

and

```
admin@ncs# show devices device-group IOS-DEVICES
```

You can influence the type of data that is present in the response with the `content` query parameter. Valid values are `config`, `nonconfig`, and `all`.

Modify the request in Postman by appending the `?content=<VALUE>` to the URL and explore how it changes the response. Here's an example for `config`:

```
GET /restconf/data/tailf-ncs:devices/device-group=IOS-DEVICES?content=config
```

```
{
  "tailf-ncs:device-group": [
    {
      "name": "IOS-DEVICES",
      "device-name": [
        "dist-rtr01",
        "dist-rtr02",
        "internet-rtr01"
      ]
    }
  ]
}
```

Another useful query parameter is named `depth` and takes a positive integer value. It specifies how many levels of data to return. This allows you to further limit the response to just the values that are of interest.

Feel free to try the last request by appending `&depth=1` at the end. You will, however, not see much of a difference because device groups have a flat structure. Instead, let's look at the more complex data model NSO has for the managed devices.

At this point you have the knowledge required to retrieve any data from the NSO, including device configurations. In the NSO data model every managed device has its own entry in the **devices** **device** list. This entry contains all the information on how NSO can access it, as well as the configuration on the device itself.

Device configurations can grow pretty big and, if you are careless, can result in poor performance. It is where the `depth` query parameter comes in handy. Here's an example.

Construct a new request in the Postman application for the *dist-rtr01* device by duplicating the last request. Change the URL to <https://sandbox-nso-1.cisco.com/restconf/data/tailf-ncs:devices/device=dist-rtr01>. Run it with and without the `?depth=1` query parameter at the end to see the difference in how long the request takes, as well as the data it returns.

```
GET /restconf/data/tailf-ncs:devices/device=core-rtr01?depth=1
```

```
{
  "tailf-ncs:device": [
    {
      "name": "core-rtr01",
      "address": "10.10.20.173",
      "ssh": {},
      "authgroup": "labadmin",
      "device-type": {},
      "ned-settings": {},
      "commit-queue": {},
      "active-settings": {},
    }
  ]
}
```

```

        "state": {},
        "capability": [],
        "module": [],
        "platform": {},
        "config": {},
        "netconf-notifications": {},
        "tailf-ncs-alarms:alarm-summary": {}
    }
}

```

4.21.1 Listing device interfaces

Another common use-case is enumerating network interfaces on a device. Fortunately, NSO keeps a copy of the device configuration under the **devices device config** path. To see the interfaces on *dist-rtr01* you could enter the following command on the NSO CLI:

```
admin@ncs# show running-config devices device dist-rtr01 config ios:interface
```

The thing to note is the `ios:` part before the `interface` keyword. It signifies a different namespace than the default one. It allows NSO to expose configuration for various devices under the same `config` path. However, you must also specify the namespace when constructing the URL.

Unfortunately, the RESTCONF namespaces do not match those on the CLI, that is, the namespace is not just *ios*. It is *tailf-ned-cisco-ios*. What is more, it depends on the device type. To quickly identify the namespace, you can use the *depth* query parameter again to list the elements right below config, like so:

```
GET /restconf/data/tailf-ncs:devices/device=dist-rtr01/config?depth=1
```

With the namespace you can finally construct the URL to display the interfaces of *dist-rtr01*:

<https://sandbox-nso-1.cisco.com/restconf/data/tailf-ncs:devices/device=dist-rtr01/config/tailf-ned-cisco-ios:interface>. It's quite a handful, isn't it? To summarize, it's made up of the following parts:

- NSO server address: `https://sandbox-nso-1.cisco.com/`
- RESTCONF data resource: `restconf/data`
- NSO device config path: `tailf-ncs:devices/device=dist-rtr01/config`

and

- device-specific interface data: `tailf-ned-cisco-ios:interface`

4.22 Stacked services (one father service calls multiple children)

4.22.1 How to use Stacked Services to improve performance?

Sometime back, I had a discussion about NSO performance with a one or two engineers from the BU. There was a suggestion that Stacked Services can be used to improve NSO performance. I took this on faith, and repeated it a few times... but then someone asked me to explain how this would work in detail... and I have to confess, now I am not sure!

Please can someone comment on the accuracy of the below explanation as to how stacked services can be used to improve performance? Thank you!

Using Stacked Services to improve NSO Service Performance

Example Scenario:

The customer has a pair of ASR5Ks for EPC, and a pair of ASR1Ks for APN VPNs. The full service has a 1000s lines of configuration on the ASR5K EPCs, but these only change very occasionally, and a few 10s of lines on the ASR1Ks that change frequently.

Problem:

NSO diff calculation speed is dependent on the number of lines of configuration in your change set. When the full service is created, your service templates result in 10000 elements of config. This is unavoidable. However, when any change is made to the Service, **FASTMAP reverts those changes**, your service re-applies those changes, and each element of the change set is compared to calculate the diffs. This is a full object tree walk, and that takes a lot of time with 10000 elements.

Solution:

To avoid this, the infrequently changed lines can be placed in a stacked service, which means that every time fastmap is ran for a Update to the service, you don't need recalculate the diffs on the ASR5K, just for the stacked service. So far so good.

However, if this is a classic parent-child stacked service (composition), whenever you modify the parent service, fastmap applies the service reverse diff. I assume that fastmap must also apply the reverse diff of the child service(s), and so on.

So a parent-child stacked service, will not help with performance issues!

But what about a side by side stacked service (aggregation) that we will call aggregator-aggregated? The aggregated service is importantly NOT created by the parent service inside fastmap (this would be composition, parent-child) - so if the config that doesn't change very often is in a separate service, then obviously the diff's wouldn't get calculated!

So, this could help, but now I have two services, not one... so how is this stacked services? If the stacked aggregated service parameters are set and changed by the aggregator, then we have stacked services. Specifically, this works well if the aggregator is creating a list in an entry in the aggregated service, because these parameters will then be fastmap'ed.

Now, this might sound a bit messy to implement, and yes, potentially using preModification to create the aggregated service/delete it upon service deletion, and adding an entry into the aggregated service list, is not very clean. But it could result in good performance improvement.

Hello NSO community members, I have 2 questions regarding stacked services. Any help is appreciated.

1) This is related to benefit of using stacked services to improve performance..

In documentation, there are a couple of places which say , by using stacked services performance will be increased, as diff calculation will be adjusted. I decided to refer to the following thread as a reference for my question as there is already explanation here from Jan Lindblad.

Jan Lindblad : That's right. When the parent service create() function runs, it will recompute the input configs for the child services, and invoke each child's create() etc, rippling all the way down to the bottom of the dependency structure. Except - and here's where the savings come in fastmap won't call the child's create() function if the input config data exactly matches the previously stored input config for that service. In this case, fastmap already knows what the child service will result in (same as last time, i.e. what's stored in the database), and just pulls that in. Since what's pulled in is exactly what's already stored in the database, it wo

n't add anything to the diff that will need to be traversed in the end, so saves (potentially a lot of) CPU cycle s.

So, in order to try this i have created a simple parent (fff) service and a simple child (ccc) service.

Parent (fff) service has inputs like that :

```
module fff {
  namespace "http://example.com/fff";
  prefix fff;
  list fff {
    description "This is an RFS skeleton service";
    key hostname;
    leaf hostname {
      tailf:info "Device name is also service unique name";
      type leafref {
        path "/ncs:devices/ncs:device/ncs:name";
      }
    }
    uses ncs:service-data;
    ncs:servicepoint fff-servicepoint;
    leaf logging1 {      type string;    }
    leaf logging2 {      type string;    }
  }
}
```

... and pushes config to child (ccc) service via template:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0">
  <ccc xmlns="http://example.com/ccc">
    <hostname>{/hostname}</hostname>
    <logging1>{/logging1}</logging1>
    <logging2>{/logging2}</logging2>
  </ccc>
</config-template>
```

Child service has yang file like this:

```
module ccc {
  namespace "http://example.com/ccc";
  prefix ccc;
  list ccc {
    description "This is an RFS skeleton service";
    key hostname;
    leaf hostname {
      tailf:info "Device name is also service unique name";
      type leafref {
        path "/ncs:devices/ncs:device/ncs:name";
      }
    }
    uses ncs:service-data;
    ncs:servicepoint ccc-servicepoint;
    leaf logging1 {      type string;    }
    leaf logging2 {      type string;    }
  }
}
```

and child service's template is :

```
<config-template xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
```

```

<device>
  <name>{/hostname}</name>
  <config>
    <logging xmlns="urn:ios">
      <host tags="replace">
        <ipv4>
          <host>{/logging1}</host>
        </ipv4>
        <ipv4>
          <host>{/logging2}</host>
        </ipv4>
      </host>
    </logging>
  </config>
</device>
</devices>
</config-template>

```

So, before i do commit of parent service, child service is already committed with SAME loggingx values, but device-manager level one of the loggingx values are deleted. And when i do commit dry-run, child service calculates the diff for the deleted logging entry..

```

admin@ncs(config-fff-Cisco_Access_1)# commit dry-run
cli {
  local-node {
    data +fff Cisco_Access_1 {
      +   logging1 6.6.6.6;
      +   logging2 7.7.7.7;
      +}
    devices {
      device Cisco_Access_1 {
        config {
          logging {
            host {
              +             ipv4 7.7.7.7 {
              +             }
            }
          }
        }
      }
    }
    +bbb Cisco_Access_1 {
    +}
  }
}

```

And going further, i commit. Then i again delete the one of the loggingx entries from device-manager. Do re-deploy dry-run, and it shows me again the diff.

```

admin@ncs(config-fff-Cisco_Access_1)# re-deploy dry-run
cli {
  local-node {
    data devices {
      device Cisco_Access_1 {
        config {
          logging {
            host {
              +             ipv4 6.6.6.6 {
              +             }
            }
          }
        }
      }
    }
  }
}

```



```

    }
  }
}

```

So, may someone explain me how can we save from the diff calculation with stacked services. I was thinking that for this child (ccc) service, diff calculation wouldn't take place during commit and or at least during re-deploy as per Jan Lindblad's explanation above.

2) when i commit service fff, i do deep-check-sync and i get following in return :

Error: No forward diff found for this service. Either /services/global-settings/collect-forward-diff is false, or the forward diff has become invalid. A re-deploy of the service will correct the latter.

I checked that collect-service-diff is set to true, and i did re-deploy. deep-check-sync result is again the same.

5 NSO installation, cfg examples

5.1 NCS Installation

This lab exercise was conducted with `nso-4.6.linux.x86_64.installer.bin` It is advisable to use the same while following this document.

Make sure Java 10 (not Open JDK) Installed `java -version` and `ant` are installed.

5.2 Install NCS

```

root@mininet-vm:/home/mininet# chmod +x nso-4.6.linux.x86_64.installer.bin
root@mininet-vm:/home/mininet# ./nso-4.6.linux.x86_64.installer.bin /opt/ncs

```

5.3 Setup NCS

```

root@mininet-vm:/home/mininet# source /opt/ncs/ncsrc
root@mininet-vm:/home/mininet# ncs-setup --dest $NCS_DIR/ncs-run
root@mininet-vm:/home/mininet# cd $NCS_DIR/ncs-run
# Start the NCS Server
root@mininet-vm:~/ncs-run# ncs

```

When NSO starts and fails to initialize, the following exit codes can occur:

- Exit codes 1 and 19 mean that an internal error has occurred. A text message should be in the logs, or if the error occurred at startup before logging had been activated, on standard error (standard output if NSO was started with `--foreground --verbose`). Generally the message will only be meaningful to the NSO developers, and an internal error should always be reported to support.
- Exit codes 2 and 3 are only used for the ncs "control commands" (see the section COMMUNICATING WITH NCS in the ncs(1) in NSO 4.2.1 Manual Pages manual page), and mean that the command failed due to timeout. Code 2 is used when the initial connect to NSO didn't succeed within 5 seconds (or the TryTime if given), while code 3 means that the NSO daemon did not complete the command within the time given by the `--timeout` option.
- Exit code 10 means that one of the init files in the CDB directory was faulty in some way. Further information in the log.

- Exit code 11 means that the CDB configuration was changed in an unsupported way. This will only happen when an existing database is detected, which was created with another configuration than the current in ncs.conf .
- Exit code 13 means that the schema change caused an upgrade, but for some reason the upgrade failed. Details are in the log. The way to recover from this situation is either to correct the problem or to re-install the old schema (fxs) files.
- Exit code 14 means that the schema change caused an upgrade, but for some reason the upgrade failed, corrupting the database in the process. This is rare and usually caused by a bug. To recover, either start from an empty database with the new schema, or re-install the old schema files and apply a backup.
- Exit code 15 means that A.cdb or C.cdb is corrupt in a non-recoverable way. Remove the files and re-start using a backup or init files.
- Exit code 20 means that NSO failed to bind a socket.
- Exit code 21 means that some NSO configuration file is faulty. More information in the logs.
- Exit code 22 indicates a NSO installation related problem, e.g. that the user does not have read access to some library files, or that some file is missing.

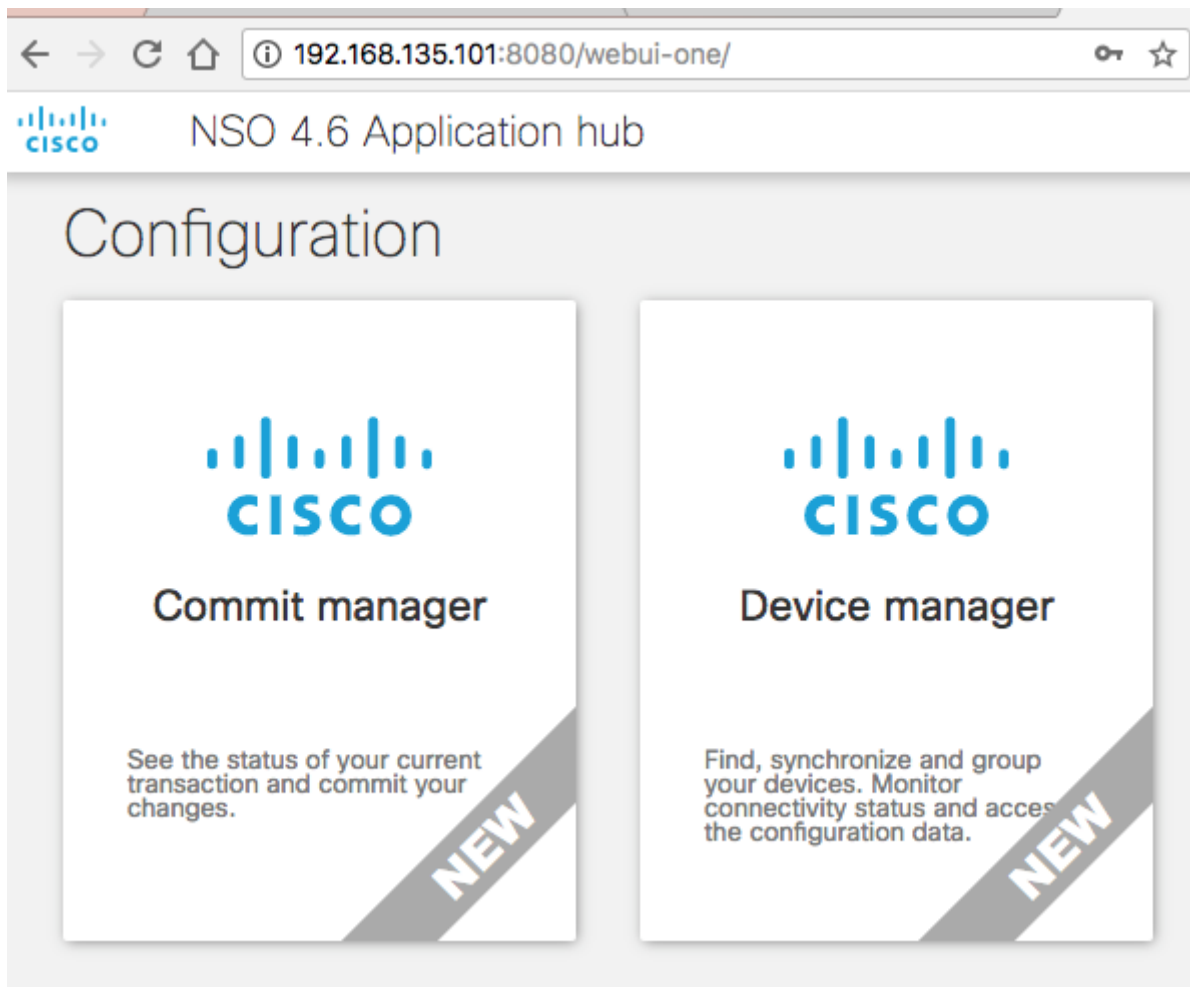
If the NSO daemon starts normally, the exit code is 0 .

Check NCS Status

```
ncs --status | grep status ncs --version 3.4.2
```

NCS/NSO via the WebUI Type <http://127.0.0.1:8080/login.html>

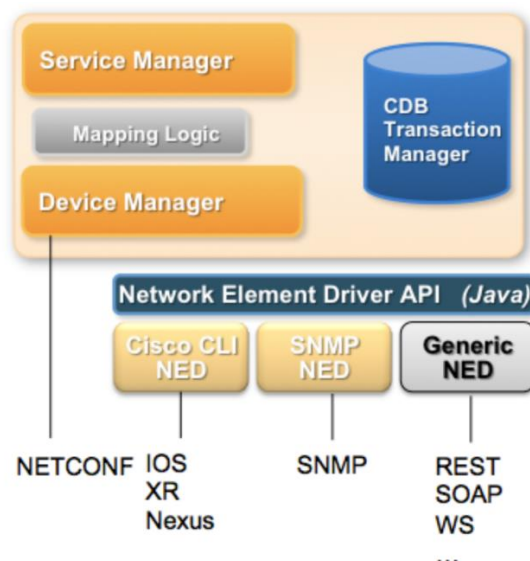
Connect to the NCS CLI `ncs_cli -u admin -C`



5.4 What are the packages available

```
root@mininet-vm:~# cd $NCS_DIR
root@mininet-vm:/opt/ncs# ls packages/neds/
a10-acos  cisco-ios  cisco-iosxr  cisco-nx  dell-ftos  juniper-junos
```

5.5 Compile a new NED



Unfortunately, the majority of existing devices in current networks do not speak NETCONF and SNMP is usually mostly used to retrieve data from devices. By far the most common way to configure network devices is through the CLI. Management systems typically connect over SSH to the CLI of the device and issue series of CLI configuration commands. Some devices do not even have a CLI, and thus SNMP, or even worse, various proprietary protocols, are used to configure the device. **NSO can speak southbound not only to NETCONF-enabled devices, but through the NED architecture it can speak to an arbitrary management interface.**

[nso-ned-4.6.pdf](#)

Compile the Cisco IOS NED package by issuing the make command. Make sure that the compilation of the NED and netsim (the part used to emulate Cisco IOS CLI which will be used throughout this course) is successful.

```
root@mininet-vm:/opt/ncs/packages/neds/cisco-ios/src# make
cd java && ant -q all

BUILD SUCCESSFUL
Total time: 0 seconds
cd ../netsim && make all
make[1]: Entering directory /opt/ncs/packages/neds/cisco-ios/netsim
make[1]: Nothing to be done for all.
make[1]: Leaving directory /opt/ncs/packages/neds/cisco-ios/netsim
```

Do the same for IOS-XR:

```
root@mininet-vm:/opt/ncs/packages/neds/cisco-iosxr# cd src
root@mininet-vm:/opt/ncs/packages/neds/cisco-iosxr/src# make
cd java && ant -q all

BUILD SUCCESSFUL
Total time: 0 seconds
cd ../netsim && make all
make[1]: Entering directory /opt/ncs/packages/neds/cisco-iosxr/netsim
make[1]: Nothing to be done for all.
make[1]: Leaving directory /opt/ncs/packages/neds/cisco-iosxr/netsim
```

The two NEDs are now compiled and available for use but they are not available to the running instance of NSO. One of the ways to make them available to the running instance is by linking a directory in the ncs-run/packages directory to point to the location of the NEDs in the packages subdirectory in the installation directory (i.e. \$NCS_DIR).

5.6 To link the newly compiled NEDS

```
cd $HOME/ncs-run/packages
ln -s /opt/ncs/packages/neds/cisco-ios cisco-ios
ln -s /opt/ncs/packages/neds/cisco-iosxr cisco-iosxr
```

Login and reload the packages

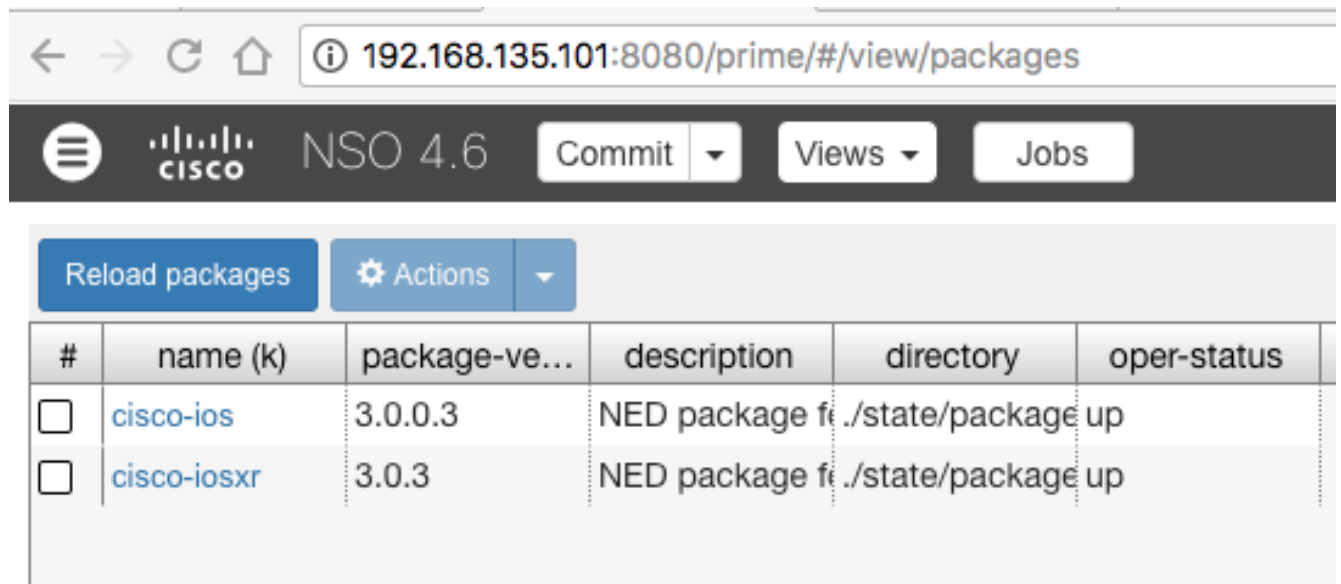
```
ncs_cli -C -u admin
root@ncs# packages reload

>>> System upgrade is starting.
>>> Sessions in configure mode must exit to operational mode.
>>> No configuration changes can be performed until upgrade has completed.
>>> System upgrade has completed successfully.
```

```

reload-result {
  package cisco-ios
  result true
}
reload-result {
  package cisco-iosxr
  result true
}

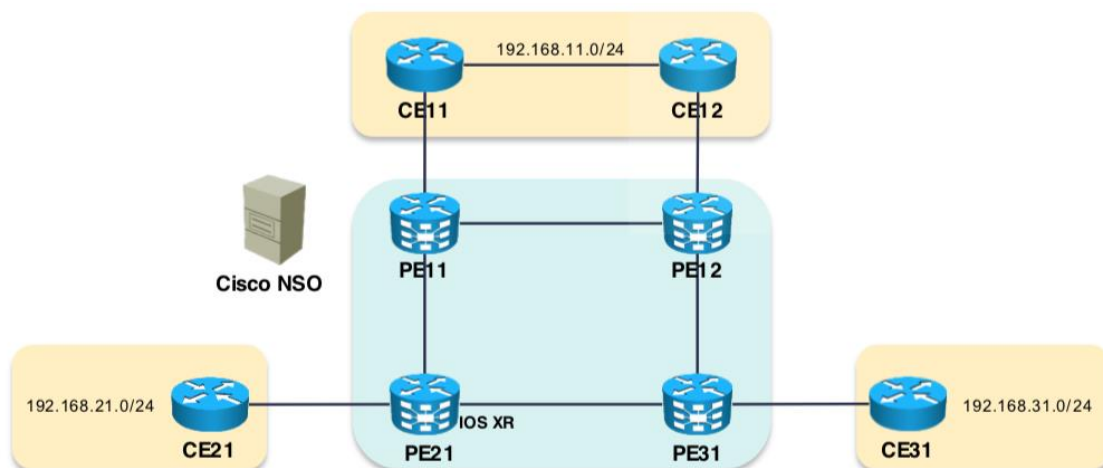
```



#	name (k)	package-ve...	description	directory	oper-status
<input type="checkbox"/>	cisco-ios	3.0.0.3	NED package fe	./state/package	up
<input type="checkbox"/>	cisco-iosxr	3.0.3	NED package fe	./state/package	up

You can also verify this with `show packages` as well

Next Objective is to Add emulated devices to the NSO and perform some initial tasks like Synchronize Config, Create Device Groups, and make sure all devices share the same config.



5.7 Add Emulated Devices to the Network

Now we will neither use real or virtual devices but use emulated devices

5.8 Create device with Net Sim

```

ncs-netsim create-device cisco-ios PE11
ncs-netsim add-device cisco-ios PE12
ncs-netsim add-device cisco-iosxr PE21
ncs-netsim add-device cisco-ios PE31
ncs-netsim add-device cisco-ios CE11

```

```
ncs-netsim add-device cisco-ios CE12
ncs-netsim add-device cisco-ios CE21
ncs-netsim add-device cisco-ios CE31
```

```
# Bulk Export for easy import
ncs-netsim ncs-xml-init > devices.xml
```

```
# And then Bulk Import to NCS
ncs_load -l -m devices.xml
```

```
admin@ncs# show devices brief
NAME  ADDRESS      DESCRIPTION  NED ID
-----
CE11   127.0.0.1    -           cisco-ios
CE12   127.0.0.1    -           cisco-ios
CE21   127.0.0.1    -           cisco-ios
CE31   127.0.0.1    -           cisco-ios
PE11   127.0.0.1    -           cisco-ios
PE12   127.0.0.1    -           cisco-ios
PE21   127.0.0.1    -           cisco-ios-xr
PE31   127.0.0.1    -           cisco-ios
R1     127.0.0.1    -           cisco-ios
```

You can actually connect to a Virtual Emulated router by the `ncs-netsim`

```
root@cisco-virtual-machine:/opt/ncs/ncs-run/packages/l2vpn/templates# ssh
admin@127.0.0.1 -p 10022
admin@127.0.0.1's password:admin

admin connected from 127.0.0.1 using ssh on cisco-virtual-machine
PE11> en
PE11#
```

Or you can connect to virtual device like this:

```
$ ncs-netsim cli-i c1
c1> enable
c1# show running-config
```

Now connect to the your NCS CLI `ncs_cli -C -u admin` and add the device to the NCS

```
config
devices device R1
address 127.0.0.1 port 10022
device-type cli ned-id cisco-ios protocol ssh
authgroup default
state admin-state unlocked
commit
```

Now connect to the device and fetch the configuration from them

```
devices fetch-host-keys
devices sync-from
end
```

5.9 Create Device Group

Now lets add the devices added above to a device group

```
admin@ncs(config)# devices device-group "PE Routers"
admin@ncs(config-device-group-PE Routers)# device-name [ PE11 PE12 PE21 PE31 ]
```

```

admin@ncs(config-device-group-PE Routers)# top
admin@ncs(config)# devices device-group "CE Routers"
admin@ncs(config-device-group-CE Routers)# device-name [ CE11 CE12 CE21 CE31 ]
admin@ncs(config-device-group-CE Routers)# top
admin@ncs(config)# devices device-group "All Routers"
admin@ncs(config-device-group-All Routers)# device-group [ "PE Routers" "CE
Routers" ]
admin@ncs(config-device-group-All Routers)# top

```

Now lets verify the configuration and commit it .

```

admin@ncs(config)# show configuration
devices device-group "All Routers"
!
devices device-group "PE Routers"
  device-name [ PE11 PE12 PE21 PE31 ]
!
devices device-group "CE Routers"
  device-name [ CE11 CE12 CE21 CE31 ]
!
devices device-group "All Routers"
  device-group [ "CE Routers" "PE Routers" ]
!
admin@ncs(config)# commit
Commit complete.
admin@ncs(config)#

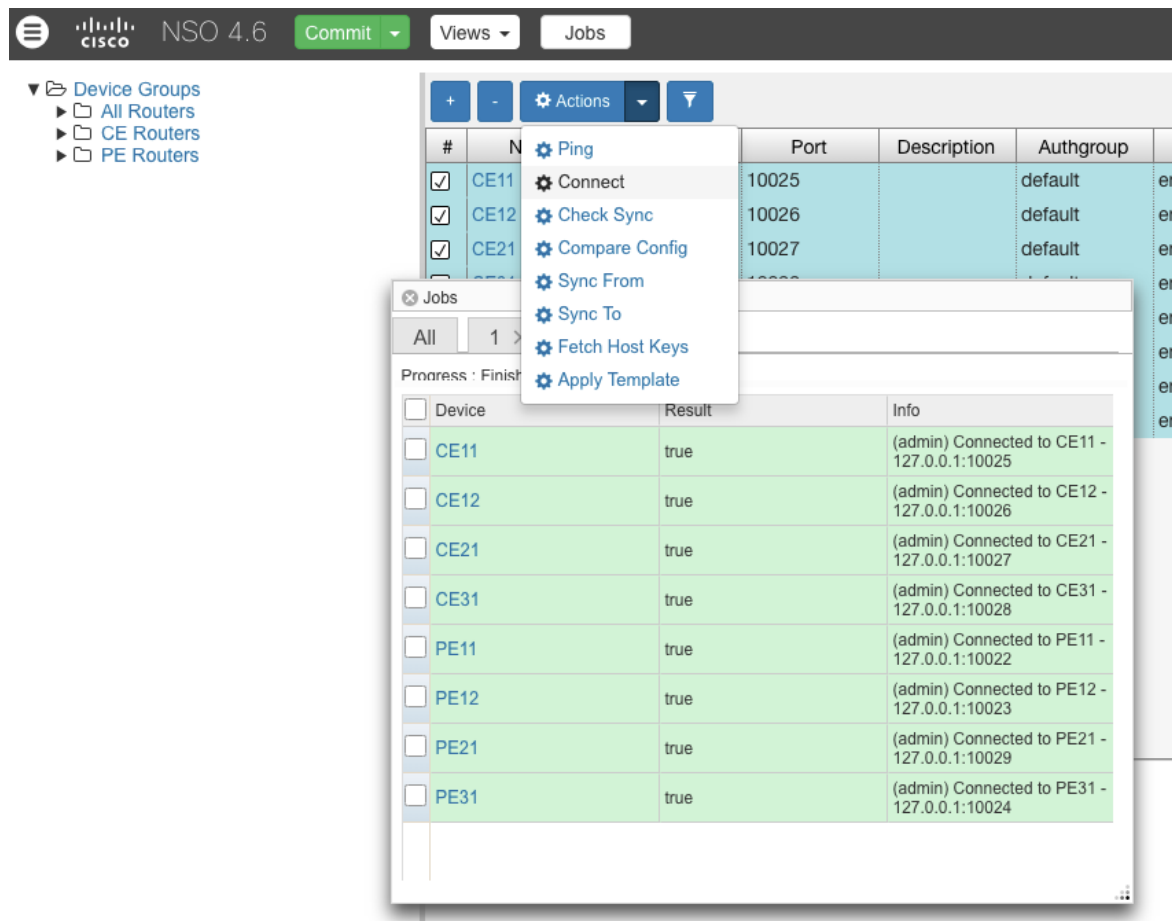
```

You can verify the above by

```

admin@ncs# show devices device-group member
NAME                MEMBER
-----
All Routers         [ CE11 CE12 CE21 CE31 PE11 PE12 PE21 PE31 ]
CE Routers           [ CE11 CE12 CE21 CE31 ]
PE Routers           [ PE11 PE12 PE21 PE31 ]

```



5.10 Create Customer

Now lets create a customer.

```
admin@ncs(config)# customers customer ACME
admin@ncs(config-customer-ACME)# rank 1
admin@ncs(config-customer-ACME)# status active
admin@ncs(config-customer-ACME)# top
admin@ncs(config)# commit
Commit complete.
admin@ncs(config)#
```

```
admin@ncs# show running-config customers
customers customer ACME
  rank    1
  status  active
!
```

5.11 Lets now create a Device template

```
admin@ncs# config
admin@ncs(config)# devices template "Common Device Parameters" config
admin@ncs(config-config)# ios:?
admin@ncs(config-config)# ios:ip domain name cisco.com
admin@ncs(config-config)# ios:ip name-server [ 192.168.133.1 ]
admin@ncs(config-config)# ios:ntp server server-list 10.0.0.1
admin@ncs(config-server-list-10.0.0.1)# exit
admin@ncs(config-config)# cisco-ios-xr:domain name cisco.com
admin@ncs(config-config)# cisco-ios-xr:domain name-server 192.168.133.1
admin@ncs(config-name-server-192.168.133.1)# exit
```



```
admin@ncs(config-config)# cisco-ios-xr:ntp server 10.0.0.1
admin@ncs(config-server-10.0.0.1)# exit
admin@ncs(config-config)# commit
Commit complete.
```

Verify the configuration above.

```
admin@ncs(config)# show full-configuration devices template Common\ Device\
Parameters
devices template "Common Device Parameters"
config
  cisco-ios-xr:domain name cisco.com
  cisco-ios-xr:domain name-server 192.168.133.1
  !
  cisco-ios-xr:ntp server 10.0.0.1
  !
  ios:ip domain name cisco.com
  ios:ip name-server [ 192.168.133.1 ]
  ios:ntp server server-list 10.0.0.1
  !
!
!
admin@ncs(config)#
```

Now apply the configuration above to all the routers.

```
admin@ncs(config)# devices device-group All\ Routers apply-template template-
name Common\ Device\ Parameters
apply-template-result {
  device CE11
  result ok
}
apply-template-result {
  device CE12
  result ok
}
apply-template-result {
  device CE21
  result ok
}
apply-template-result {
  device CE31
  result ok
}
apply-template-result {
  device PE11
  result ok
}
apply-template-result {
  device PE12
  result ok
}
apply-template-result {
  device PE21
  result ok
}
apply-template-result {
  device PE31
  result ok
}
```

Note that from the above command the configuration is still not applied on the routers unless it is committed . Lets do a dry run before committing it.

```
admin@ncs(config)# commit dry-run
cli {
  local-node {
    data devices {
      device CE11 {
        config {
          ios:ip {
            domain {
+              name cisco.com;
            }
+            name-server 192.168.133.1;
          }
          ios:ntp {
            server {
+              server-list 10.0.0.1 {
+              }
            }
          }
        }
      }
      device CE12 {
        config {
          ios:ip {
            domain {
+              name cisco.com;
            }
+            name-server 192.168.133.1;
          }
          ios:ntp {
            server {
+              server-list 10.0.0.1 {
+              }
            }
          }
        }
      }
    }
  }
}
```

```
admin@ncs(config)# commit
Commit complete.
```

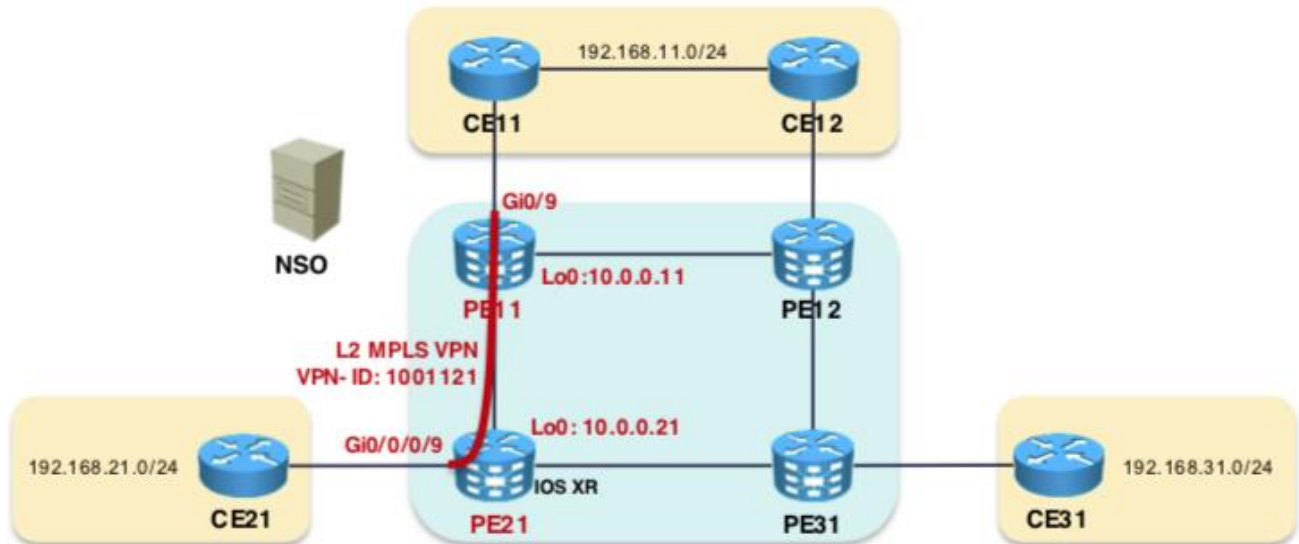
Now that we have the configuration applied, lets check on the devices individually.

```
admin@ncs(config)# show full-configuration devices device PE21
devices device PE21
  address 127.0.0.1
  port 10029
  ssh host-key ssh-rsa
  key-data
...
!
authgroup default
device-type cli ned-id cisco-ios-xr
state admin-state unlocked
config
  cisco-ios-xr:domain name cisco.com
  cisco-ios-xr:domain name-server 192.168.133.1
  cisco-ios-xr:ntp
    server 10.0.0.1
```

exit

5.12 Now Lets create a Service

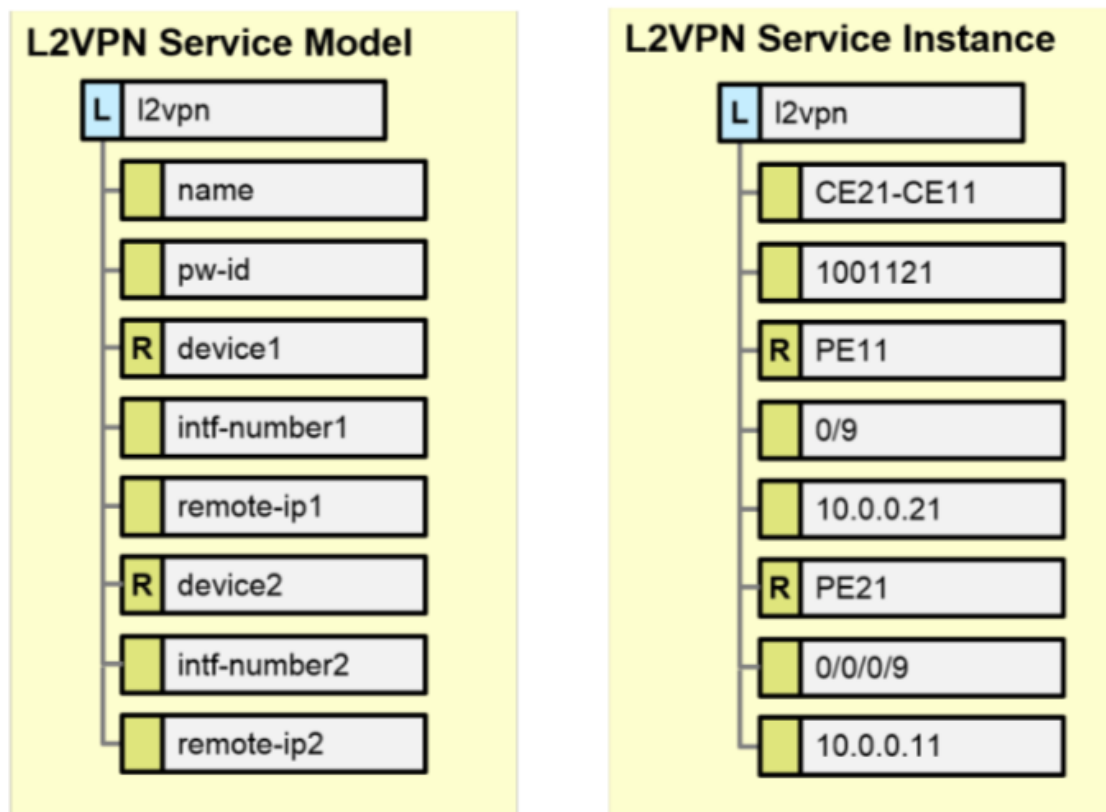
In this activity we will create a simple point to point Layer 2 MPLS VPN. Take a detour a learn a little bit about Layer 2 MPLS VPN , Here is the link to the [blog](#).



5.13 Lets Start with creating an L2 VPN YANG template

In this step a YANG file is created which is basically having the variables to configured in a service . For example in the above VPN configuration in the picture above , the variables would be the VPN-ID,the loopback interfaces , the physical interfaces on which xconnect is configured . etc .

Look in the Model Below and notice how the real values on the the right map the variables



Create the skeleton:

```
root@cisco-virtual-machine:/opt/ncs/ncs-run/packages/l2vpn/src/yang# cd
/opt/ncs/ncs-run/packages/
root@cisco-virtual-machine:/opt/ncs/ncs-run/packages# ncs-make-package --
service-skeleton template l2vpn
root@cisco-virtual-machine:/opt/ncs/ncs-run/packages# ls
cisco-ios  cisco-ios-xr  l2vpn
```

Now edit the `l2vpn.yang` file and put in the data below. More Details on the YANG file format on the other blog but for now proceed.

```
module l2vpn {
  namespace "http://com/example/l2vpn";
  prefix l2vpn;

  import ietf-inet-types { prefix inet; }
  import tailf-ncs { prefix ncs; }
  import tailf-common { prefix tailf; }

  augment "/ncs:services" {
    list l2vpn {
      key "name";                                ← must be unique in a list
      unique "pw-id";
      uses ncs:service-data;
      ncs:servicepoint "l2vpn";

      leaf name {
        tailf:info "Service Instance Name";
        type string;
      }

      leaf pw-id {
        tailf:info "Unique Pseudowire ID";
        mandatory true;
        type uint32 {
          range "1..4294967295";
        }
      }

      leaf device1 {
        tailf:info "PE Router1";
        mandatory true;
        type leafref {
          path "/ncs:devices/ncs:device/ncs:name";
        }
      }

      # could be another path reference ...
      leaf intf-number1 {
        tailf:info "GigabitEthernet Interface ID";
        mandatory true;
        type string {
          pattern "[0-9]{1,2}(/[0-9]{1,2}){1,4}";
        }
      }

      leaf remote-ip1 {
        tailf:info "Loopback0 IP Address of Remote PE (10.0.0.X)";
        mandatory true;
        type inet:ipv4-address {
          pattern "10\\.0\\.0\\. [0-9]+";
        }
      }
    }
  }
}
```



```

}
reload-result {
    package l2vpn
    result true
}
admin@ncs#

```

Check CLI for the new l2vpn commands

```

admin@ncs(config)# services l2vpn ?
Possible completions:
  Service Instance Name  range
admin@ncs(config)# services l2vpn

```

You still cannot start provisioning services. A service template is missing and it will be created in the next task.

5.14 Now Lets create a Device Template based on the Actual CLI configuration

Following are the “real” CLI commands used in a typical MPLS L2 VPN configuration. We will configure this on the NSO and then work on templating the config with variables.

Cisco IOS

```

interface GigabitEthernet0/9
    xconnect 10.0.0.21 1001121 encapsulation mpls

```

Cisco IOS XR

```

l2vpn
xconnect group ACME
p2p CE11-to-CE21
    interface GigabitEthernet0/0/0/9
        neighbor 10.0.0.11 pw-id 1001121
!
!
interface GigabitEthernet0/0/0/9
    l2transport
!

```

Now configure the devices in question as per the provided values in the picture below. **Note: We will not apply the configuration but just get the XML output and abort**

```

admin@ncs(config)# show configuration

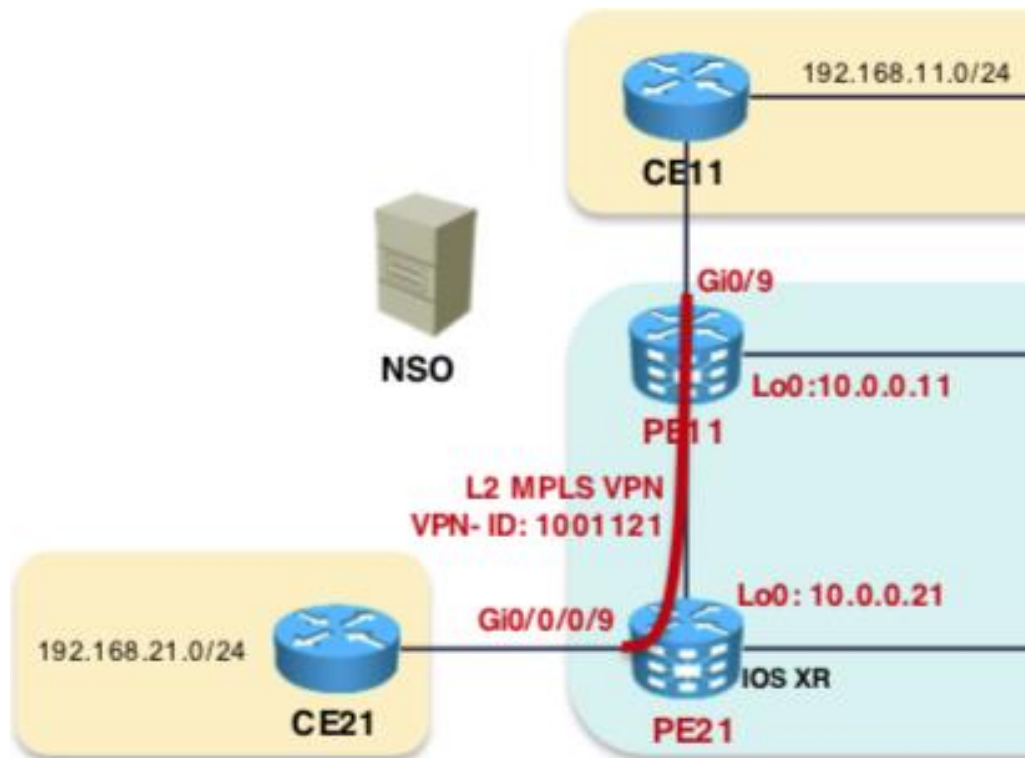
devices device PE11
config
    ios:interface GigabitEthernet0/9
        xconnect 10.0.0.21 1001121 encapsulation mpls
    exit
!
!
devices device PE21
config
    cisco-ios-xr:interface GigabitEthernet 0/0/0/9
        l2transport
    exit
    exit
    cisco-ios-xr:l2vpn
        xconnect group GROUP
        p2p CE11-to-CE21
            interface GigabitEthernet0/0/0/9
                neighbor 10.0.0.11 pw-id 1001121

```

```

exit
exit
exit
exit
!
!

```



Use the `commit dry-run outformat xml` command to retrieve the **XML version of the configuration** above.

```

admin@ncs(config)# commit dry-run outformat xml
result-xml {
  local-node {
    data <devices xmlns="http://tail-f.com/ns/ncs">
      <device>
        <name>PE11</name>
        <config>
          <interface xmlns="urn:ios">
            <GigabitEthernet>
              <name>0/9</name>
              <xconnect>
                <address>10.0.0.21</address>
                <vcid>1001121</vcid>
                <encapsulation>mpls</encapsulation>
              </xconnect>
            </GigabitEthernet>
          </interface>
        </config>
      </device>
      <device>
        <name>PE21</name>
        <config>
          <interface xmlns="http://tail-f.com/ned/cisco-ios-xr">
            <GigabitEthernet>

```

```

        <id>0/0/0/9</id>
        <l2transport/>
      </GigabitEthernet>
    </interface>
    <l2vpn xmlns="http://tail-f.com/ned/cisco-ios-xr">
      <xconnect>
        <group>
          <name>GROUP</name>
          <p2p>
            <name>CE11-to-CE21</name>
            <interface>
              <name>GigabitEthernet0/0/0/9</name>
            </interface>
            <neighbor>
              <address>10.0.0.11</address>
              <pw-id>1001121</pw-id>
            </neighbor>
          </p2p>
        </group>
      </xconnect>
    </l2vpn>
  </config>
</device>
</devices>
}
}

```

Now go to `$NCS_DIR/ncs-run/packages/l2vpn/templates` and edit the `l2vpn-template.xml` file. In the template below we will put the config generated above for each device. **Twice in a section as the device could be either IOS or IOSXR.**

```

<config-template xmlns="http://tail-f.com/ns/config/1.0"
  servicepoint="l2vpn">
  <devices xmlns="http://tail-f.com/ns/ncs">

    <!-- DEVICE1 -->
    <device>
      <name>{/device1}</name>
      <config>

        <!-- DEVICE1/IOS -->

        <!-- DEVICE1/IOS-XR -->

      </config>
    </device>

    <!-- DEVICE2 -->
    <device>
      <name>{/device2}</name>
      <config>

        <!-- DEVICE1/IOS -->

        <!-- DEVICE1/IOS-XR -->

      </config>
    </device>
  </devices>
</config-template>

```



```
</devices>
</config-template>
```

So after putting the XML Data generated as a part of Dry run in the above template the XML file now looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<config-template xmlns="http://tail-f.com/ns/config/1.0"
servicepoint="l2vpn">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <!-- DEVICE1 -->
    <device>
      <name>{/device1}</name>
      <config>
        <!-- DEVICE1/IOS -->

        <interface xmlns="urn:ios">
          <GigabitEthernet>
            <name>0/9</name>
            <xconnect>
              <address>10.0.0.21</address>
              <vcid>1001121</vcid>
              <encapsulation>mpls</encapsulation>
            </xconnect>
          </GigabitEthernet>
        </interface>

        <!-- DEVICE1/IOS-XR -->

        <interface xmlns="http://tail-f.com/ned/cisco-ios-xr">
          <GigabitEthernet>
            <id>0/0/0/9</id>
            <l2transport/>
          </GigabitEthernet>
        </interface>
        <l2vpn xmlns="http://tail-f.com/ned/cisco-ios-xr">
          <xconnect>
            <group>
              <name>GROUP</name>
              <p2p>
                <name>CE11-to-CE21</name>
                <interface>
                  <name>GigabitEthernet0/0/0/9</name>
                </interface>
                <neighbor>
                  <address>10.0.0.11</address>
                  <pw-id>1001121</pw-id>
                </neighbor>
              </p2p>
            </group>
          </xconnect>
        </l2vpn>
      </config>
    </device>

    <!-- DEVICE2 -->
    <device>
      <name>{/device2}</name>
      <config>
        <!-- DEVICE2/IOS -->
```

```

<interface xmlns="urn:ios">
  <GigabitEthernet>
    <name>0/9</name>
    <xconnect>
      <address>10.0.0.21</address>
      <vcid>1001121</vcid>
      <encapsulation>mpls</encapsulation>
    </xconnect>
  </GigabitEthernet>
</interface>

<!-- DEVICE2/IOS-XR -->

<interface xmlns="http://tail-f.com/ned/cisco-ios-xr">
  <GigabitEthernet>
    <id>0/0/0/9</id>
    <l2transport/>
  </GigabitEthernet>
</interface>
<l2vpn xmlns="http://tail-f.com/ned/cisco-ios-xr">
  <xconnect>
    <group>
      <name>GROUP</name>
      <p2p>
        <name>CE11-to-CE21</name>
        <interface>
          <name>GigabitEthernet0/0/0/9</name>
        </interface>
        <neighbor>
          <address>10.0.0.11</address>
          <pw-id>1001121</pw-id>
        </neighbor>
      </p2p>
    </group>
  </xconnect>
</l2vpn>
</config>
</device>
</devices>
</config-template>

```

Now in the next step we will replace the hardcoded value in the XML above to variables so that it is templated. Here is the file after the values are templated.

```

<?xml version="1.0" encoding="UTF-8"?>
<config-template xmlns="http://tail-f.com/ns/config/1.0"
servicepoint="l2vpn">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <!-- DEVICE1 -->
    <device tags="nocreate">
      <name>{/device1}</name>
      <config tags="merge">
        <!-- DEVICE1/IOS -->

        <interface xmlns="urn:ios">
          <GigabitEthernet>
            <name>{/intf-number1}</name>
            <xconnect>
              <address>{/remote-ip1}</address>
              <vcid>{/pw-id}</vcid>
              <encapsulation>mpls</encapsulation>
            </xconnect>

```

```

    </GigabitEthernet>
</interface>

<!-- DEVICE1/IOS-XR -->

<interface xmlns="http://tail-f.com/ned/cisco-ios-xr">
  <GigabitEthernet>
    <id>{/intf-number1}</id>
    <l2transport/>
  </GigabitEthernet>
</interface>
<l2vpn xmlns="http://tail-f.com/ned/cisco-ios-xr">
  <xconnect>
    <group>
      <name>GROUP</name>
      <p2p>
        <name>{/name}</name>
        <interface>
          <name>GigabitEthernet{/intf-number1}</name>
        </interface>
        <neighbor>
          <address>{/remote-ip1}</address>
          <pw-id>{/pw-id}</pw-id>
        </neighbor>
      </p2p>
    </group>
  </xconnect>
</l2vpn>
</config>
</device>

<!-- DEVICE2 -->
<device tags="nocreate">
  <name>{/device2}</name>
  <config tags="merge">
    <!-- DEVICE1/IOS -->

    <interface xmlns="urn:ios">
      <GigabitEthernet>
        <name>{/intf-number2}</name>
        <xconnect>
          <address>{/remote-ip2}</address>
          <vcid>{/pw-id}</vcid>
          <encapsulation>mpls</encapsulation>
        </xconnect>
      </GigabitEthernet>
    </interface>

    <!-- DEVICE1/IOS-XR -->

    <interface xmlns="http://tail-f.com/ned/cisco-ios-xr">
      <GigabitEthernet>
        <id>{/intf-number2}</id>
        <l2transport/>
      </GigabitEthernet>
    </interface>
    <l2vpn xmlns="http://tail-f.com/ned/cisco-ios-xr">
      <xconnect>
        <group>
          <name>GROUP</name>
          <p2p>
            <name>{/name}</name>
            <interface>

```

```

        <name>GigabitEthernet{/intf-number2}</name>
    </interface>
    <neighbor>
        <address>{/remote-ip2}</address>
        <pw-id>{/pw-id}</pw-id>
    </neighbor>
</p2p>
</group>
</xconnect>
</l2vpn>
</config>
</device>
</devices>
</config-template>

```

After saving the above file (/opt/ncs/ncs-run/packages/l2vpn/templates) perform a reload of the packages for NCS:

```

admin@ncs# packages reload
reload-result {
    package cisco-ios
    result true
}
reload-result {
    package cisco-iosxr
    result true
}
reload-result {
    package l2vpn
    result true
}

```

You should now see more options in the L2VPN Service configuration

```

admin@ncs(config)# services l2vpn CE11-CE21 ?
Possible completions:
  check-sync          Check if device config is according to the service
  commit-queue
  deep-check-sync     Check if device config is according to the service
  device1             PE Router1
  device2             PE Router2
  get-modifications   Get the data this service created
  intf-number1        GigabitEthernet Interface ID
  intf-number2        GigabitEthernet Interface ID
  log
  pw-id               Unique Pseudowire ID
  re-deploy            Run/Dry-run the service logic again
  reactive-re-deploy   Reactive redeploy of service logic
  remote-ip1          Loopback0 IP Address of Remote PE (10.0.0.X)
  remote-ip2          Loopback0 IP Address of Remote PE (10.0.0.X)
  touch               Touch a service
  un-deploy            Undo the effects of this service
  <cr>

```

Now deploy the service on the routers in the example:

```

admin@ncs(config)# services l2vpn CE11-CE21 pw-id 1001121 device1 PE11 intf-
number1 0/9 remote-ip1 10.0.0.21 device2 PE21 intf-number2 0/0/0/9 remote-ip2
10.0.0.11

admin@ncs(config-l2vpn-CE11-CE21)# commit dry-run

```

```

cli {
  local-node {
    data devices {
      device PE11 {
        config {
          ios:interface {
+           GigabitEthernet 0/9 {
+             xconnect {
+               address 10.0.0.21;
+               vcid 1001121;
+               encapsulation mpls;
+             }
+           }
+         }
      }
      device PE21 {
        config {
          cisco-ios-xr:interface {
+           GigabitEthernet 0/0/0/9 {
+             l2transport {
+             }
+           }
+         }
          cisco-ios-xr:l2vpn {
            xconnect {
+             group GROUP {
+               p2p CE11-CE21 {
+                 interface GigabitEthernet0/0/0/9;
+                 neighbor 10.0.0.11 1001121;
+               }
+             }
+           }
+         }
      }
    }
  }
  services {
+    l2vpn CE11-CE21 {
+      pw-id 1001121;
+      device1 PE11;
+      intf-number1 0/9;
+      remote-ip1 10.0.0.21;
+      device2 PE21;
+      intf-number2 0/0/0/9;
+      remote-ip2 10.0.0.11;
+    }
  }
}
}

```

After the above is finally committed by a `commit` you see the actual configuration changes on the device by the following command:

```

admin@ncs(config)# show full-configuration devices device PE11
devices device PE11
  address 127.0.0.1
  port 10022
  ssh host-key ssh-rsa
  key-data
"AAAAB3NzaC1yc2EAAAADAQABAAQAC8dMrFKaxMJWsJZgHkV+LJqqIbxxUB3ntam9BpES32\n86fDH
aMQPHqVxMqXGnTP5kJYHL8y5hmR9q6pcdDCZ+eCoLZznjZvWu74Osqa1Av2e9NW4yoX\nSheaj3GRur7

```

```

roLcv0rzenpipKUnlzv6Hl9yw2nA3E1FFljMEZZ2yhaZ0j8JlMEIvSVEmGbzd\nATlnblRw0sgAVPtC0
ssoEwBQaRX8iicN3GsDcMpW7/mkVfkROws1JTO5C/UC0Um0bL5miJ7z\n+eKvIZiRY80pBKTjY17e8L4
iKOvwvPV9VdlHB3ePJuZw4NNqcd5m2NuiqtnGBlu0MIxexoIhu\nnnTa+xAyqoKH3"
!
authgroup default
device-type cli ned-id cisco-ios
state admin-state unlocked
config
  no ios:service pad
  ios:ip vrf my-forward
    bgp next-hop Loopback 1
!
ios:ip community-list 1 permit
ios:ip community-list 2 deny
ios:ip community-list standard s permit

```

Also verify the current service deployment by the following command:

```

admin@ncs# show running-config services l2vpn
services l2vpn CE11-CE21
  pw-id      1001121
  device1    PE11
  intf-number1 0/9
  remote-ip1 10.0.0.21
  device2    PE21
  intf-number2 0/0/0/9
  remote-ip2 10.0.0.11
!

```

```

admin@ncs# show running-config services l2vpn | display xpath
/services/l2vpn:l2vpn[name='CE11-CE21']/pw-id 1001121
/services/l2vpn:l2vpn[name='CE11-CE21']/device1 PE11
/services/l2vpn:l2vpn[name='CE11-CE21']/intf-number1 0/9
/services/l2vpn:l2vpn[name='CE11-CE21']/remote-ip1 10.0.0.21
/services/l2vpn:l2vpn[name='CE11-CE21']/device2 PE21
/services/l2vpn:l2vpn[name='CE11-CE21']/intf-number2 0/0/0/9
/services/l2vpn:l2vpn[name='CE11-CE21']/remote-ip2 10.0.0.11

```

```

admin@ncs# show running-config services l2vpn | tab

```

NAME	PW ID	DEVICE1	INTF NUMBER1	REMOTE IP1	DEVICE2	INTF NUMBER2	REMOTE IP2
CE11-CE21	1001121	PE11	0/9	10.0.0.21	PE21	0/0/0/9	10.0.0.11

You can un-deploy a service:

```

services l2vpn CE11-to-CE31 un-deploy

```

Check to see if the service is still present in the NSO configuration database.

```

show full-configuration services l2vpn

```

Check the state of the un-deployed service instance using the check-sync command.

```

services l2vpn CE11-to-CE31 check-sync

```

Now re-deploy the service instance to make it operational again.

```

services l2vpn CE11-to-CE31 re-deploy

```

Check the status:

```

services l2vpn CE11-to-CE31 check-sync

```

5.15 Assign Service Instances to Customers

In this task, you will modify the `service model` (The YANG Model) to include the assignment of service instances to customers.

```
# this could be replaced with an 'enum' list
leaf customer {
    tailf:info "Customer name";
    type leafref {
path "/ncs:customers/ncs:customer/ncs:id"; }
}
```

After adding the above , make the file and reload packages. You will now be able to assign Customer to the Services:

```
Entering configuration mode terminal
admin@ncs(config)# services l2vpn CE11-to-CE31 customer ?
Description: Customer name
Possible completions:
  ACME
admin@ncs(config)# services l2vpn CE11-to-CE31 customer ACME
admin@ncs(config-l2vpn-CE11-to-CE31)# top
admin@ncs(config)# commit

admin@ncs# show running-config services l2vpn CE11-to-CE31
services l2vpn CE11-to-CE31
  customer      ACME
  pw-id         1011131
  device1      PE11
  intf-number1 0/7
  remote-ip1    10.0.0.21
  device2      PE31
  intf-number2 0/0/0/7
  remote-ip2    10.0.0.11
```

5.16 Creating a Custom Auth Group for “Real Cisco Devices”

Get the Remote Device Ready for Connection:

```
enable password cisco
username vikassri pass cisco

line vty 0 4
  password cisco
  login local
  transport input telnet
!
```

1. Go to the following screen (can be done via CLI)

NSO 4.6
 Commit
Views
Jobs

[/ ncs:devices](#) / [authgroups](#) / [group](#)

Authentication settings for a group of devices

group

Authentication settings for a group of devices

+
-
Actions
▼
▼

#	
<input type="checkbox"/>	default
<input type="checkbox"/>	telnetauthgroup

2. Define the Parameters and the users

NSO 4.6
 Commit
Views
Jobs

[/ ncs:devices](#) / [authgroups](#) / [group](#) { [telnetauthgroup](#) }

Authentication settings for a group of devices

group
default-map
umap

Remote authentication parameters for users not in umap

default-map

☒

Choice - remote-user

remote-name

default-map

☒

remote-name

vikassri

Choice - remote-auth

remote-password

default-map

☒

remote-password

.....

3.Create a local user

NSO 4.6 Commit Views Jobs Alarms 1

/ ncs:devices / authgroups / group { telnetauthgroup } / umap

Map NCS users to remote authentication parameters

umap

Map NCS users to remote authentication parameters

#	local-user (k)	remote-secondary-password
<input type="checkbox"/>	vikassri	\$8\$wGurvsVjQvGY69yLJOH8/f6Zu99ai+RaP/M46PbxZnM=

4.User Parameters

NSO 4.6 Commit Views Jobs

/ ncs:devices / authgroups / group { telnetauthgroup } / umap { vikassri }

Map NCS users to remote authentication parameters

umap

INFO

local-user

vikassri

Choice - remote-user

remote-name

remote-name

vikassri

Choice - remote-auth

remote-password

remote-password

.....

remote-secondary-password

.....

CLI Way of doing the above :

```
# devices authgroups group JUN_Auth default-map remote-name root remote-password cisco
# devices device JUN1 address 10.66.77.39 port 22 authgroup JUN_Auth device-type netconf
# devices device JUN1 state admin-state unlocked
# devices device JUN2 connect
$ devices device JUN2 sync-from
```

Now once you have the above configuration done , try adding OSPF configuration for the device from CLI and update !

5.17 You Can Install Custom Commands at `/ncs-run/scripts/command`

```
cisco@NCS:~/ncs-run/scripts/command$ more show-trace #!/bin/sh
set -e
newline=cat
while [ $# -gt 0 ]; do
    case "$1" in
        --command)
            cat << EOF
begin command
modes: oper
styles: c i j
cmdpath: show trace
help: Show the contents of trace log files
more: true
end
EOF
        exit
;; list)
        cd /home/cisco/ncs-run/logs/
        ls -l *.trace
        exit
        ;;
        file)
        cd /home/cisco/ncs-run/logs/
        more $2
        exit
        ;;
    *)
break
;; esac

Step 3

Step 4
shift done
echo Specify a trace log file to display.
echo Usege:
echo show trace list ... lists trace log files
echo show trace file \<file\> ... displays the selected log file
exit
```

And then load the above command in NSO

```
admin@ncs# script reload
/opt/ncs/ncs-run/scripts: ok
```

You can verify the activity by

```
admin@ncs# show trace file ned-cisco-ios-PE11.trace
>> 28-Jan-2016::01:58:14.909 CLI CONNECT to PE11-127.0.0.1:10101 as admin
(Trace=false)
<< 28-Jan-2016::01:58:15.178 CONNECTED 0
>> 28-Jan-2016::01:58:15.179 IS_ALIVE 0
<< 28-Jan-2016::01:58:15.181 IS_ALIVE true
>> 28-Jan-2016::01:58:46.807 CLOSE 0: (Pool: discard) << 28-Jan-
2016::01:58:46.817 CLOSED
admin@ncs#
```

Note that only the scripts created in the above format will be available in the cli after `scripts reload`. The script **below** will not be available; As it is not the NSO script format. Another example of the Script

```
#!/bin/sh
{ ncs_cli -u admin -C<< EOF;
config
customers customer $1 rank 128 status active
commit
exit no-confirm
exit
EOF
}
if [ $? != 0 ]; then echo 'create-customer: script failed'; exit 1; fi
```

5.18 NCS Troubleshooting commands

```
ncs --stop
ncs --foreground -v
```

5.19 Service Model Optimisation

In the service deployed earlier for the MPLS VPN the user had to manually find out from somewhere else the `interface numbers` and the `IP Address` of the remote end before typing it in. This introduces risk for **errors**.

We can optimise the service model by collecting that information programmatically . Here are some of the optimisations which we would follow:

1. Place circuits in a list called `link` with a `max` and `min` elements to 2.
2. **Link to the devices interface** so that they can be selected instead of typed in
3. **Automatically retrieve** the other PE's loopback address.

Note by R.Andreetta: an even better way to do it, would be to write down a python script that automatically selects a free interface and configured everything. The operator through the guy only needs to select the ingress/egress PEs and the interface type (e.g. GigabitEthernet / TenGigabitEthernet).

Alright so lets tackle the task 1 above. Optimized files for the exercise.

5.20 `optimized.l2vpn.yang`

```
module l2vpn {
  namespace "http://com/example/l2vpn";
  prefix l2vpn;

  import ietf-inet-types {
    prefix inet;
  }
  import tailf-ncs {
    prefix ncs;
  }
  import tailf-common {
    prefix tailf;
  }

  import tailf-ned-cisco-ios {
    prefix ios;
  }
  import tailf-ned-cisco-ios-xr {
    prefix cisco-ios-xr;
  }
}
```

```

}

augment "/ncs:services" {
  list l2vpn {
    key "name";
    unique "pw-id";
    uses ncs:service-data;
    ncs:servicepoint "l2vpn";
    leaf name {
      tailf:info "Service Instance Name";
      mandatory true;
      type string;
    }
    leaf pw-id {
      tailf:info "Unique Pseudowire ID";
      mandatory true;
      type uint32 {
        range "1..4294967295";
      }
    }
  }
  list link {
    tailf:info "Attachment Circuits";
    min-elements 2;
    max-elements "2";
    key "device";
    leaf device {
      tailf:info "PE Router";
      mandatory true;
      type leafref {
        path "/ncs:devices/ncs:device/ncs:name";
      }
    }
  }
  container ios {
    # clause that add a constraint about the device being an IOS device
    # this is necessary because also the if-type refers to 'ios:'
    when
      "/ncs:devices/ncs:device[ncs:name=current()/../device]/ncs:device-
type/ncs:cli/ncs:ned-id='ios-id:cisco-ios'" {
      tailf:dependency "../device";
      tailf:dependency "/ncs:devices/ncs:device/ncs:device-type";
    }
    leaf intf-number {
      tailf:info "GigabitEthernet Interface ID";
      mandatory true;
      type leafref {
        path
"deref(..../device)/../ncs:config/ios:interface/ios:GigabitEthernet/ios:name";

      }
    }
  }
  container iosxr {
    when
      "/ncs:devices/ncs:device[ncs:name=current()/../device]/ncs:device-
type/ncs:cli/ncs:ned-id='cisco-ios-xr-id:cisco-ios-xr'" {
      tailf:dependency "../device";
      tailf:dependency "/ncs:devices/ncs:device/ncs:device-type";
    }
    leaf intf-number {
      tailf:info "GigabitEthernet Interface ID";
      mandatory true;
      type leafref {

```



```

<device tags="nocreate">
  <name>{/link[2]/device}</name>
  <config tags="merge">
    <interface xmlns="urn:ios" tags="nocreate">
      <GigabitEthernet>
        <name>{/link[2]/ios/intf-number}</name>
        <xconnect tags="merge">
          <address>{/link[2]/remote-ip}</address>
          <vcid>{/pw-id}</vcid>
          <encapsulation>mpls</encapsulation>
        </xconnect>
      </GigabitEthernet>
    </interface>
    <l2vpn xmlns="http://tail-f.com/ned/cisco-ios-xr" tags="merge">
      <xconnect>
        <group>
          <name>GROUP</name>
          <p2p>
            <name>{/name}</name>
            <interface>
              <name>GigabitEthernet{/link[2]/iosxr/intf-number}</name>
            </interface>
            <neighbor>
              <address>{/link[2]/remote-ip}</address>
              <pw-id>{/pw-id}</pw-id>
            </neighbor>
          </p2p>
        </group>
      </xconnect>
    </l2vpn>
    <interface xmlns="http://tail-f.com/ned/cisco-ios-xr" tags="nocreate">
      <GigabitEthernet>
        <id>{/link[2]/iosxr/intf-number}</id>
        <l2transport/>
      </GigabitEthernet>
    </interface>
  </config>
</device>
</devices>
</config-template>

```

In the above configuration; if you are doing it with simulated routers (Dynamips Based) and if you would like to show the interfaces on the simulated router you will have to modify the leafref in intf-number for container ios like the following, **NOTICE** the “FastEthernet”

```

leaf intf-number {
  tailf:info "FastEthernet Interface ID";
  mandatory true;
  type leafref {
    path
    "deref(..../device)/../ncs:config/ios:interface/ios:FastEthernet/ios:name";
  }
}

```

5.22 optimized.l2vpn.Makefile

```

all: fxs
.PHONY: all

# Include standard NCS examples build definitions and rules
include $(NCS_DIR)/src/ncs/build/include.ncs.mk

SRC = $(wildcard yang/*.yang)

```

```

DIRS = ../load-dir
FXS = $(SRC:yang/%.yang=../load-dir/%.fxs)

## Uncomment and patch the line below if you have a dependency to a NED
## or to other YANG files
#YANGPATH += ../../<ned-name>/src/ncsc-out/modules/yang \
#           ../../<pkt-name>/src/yang

YANGPATH += /opt/ncs/packages/neds/cisco-ios/src/yang \
            /opt/ncs/packages/neds/cisco-iosxr/src/yang \
            /opt/ncs/packages/neds/cisco-ios/src/ncsc-out/modules/yang \
            /opt/ncs/packages/neds/cisco-iosxr/src/ncsc-out/modules/yang

NCSCPATH = $(YANGPATH:%=--yangpath %)
YANGERPATH = $(YANGPATH:%=--path %)

fxs: $(DIRS) $(FXS)
.PHONY: fxs

$(DIRS):
    mkdir -p $@

../load-dir/%.fxs: yang/%.yang
    $(NCSC) `ls $*-ann.yang` > /dev/null 2>&1 && echo "-a $*-ann.yang" ` \
    $(NCSCPATH) -c -o $@ $<
clean:
    rm -rf $(DIRS)
.PHONY: clean

```

5.23 NSO 200 Base Yang

```

module l3mplsvpn {
    namespace "http://com/example/l3mplsvpn";
    prefix l3mplsvpn;

    import ietf-inet-types {
        prefix inet;
    }
    import tailf-ncs {
        prefix ncs;
    }
    import tailf-common {
        prefix tailf;
    }

    augment "/ncs:services" {
        leaf l3mplsvpn-id-cnt {
            description
                "Provides a unique 32-bit number used as VPN instance identifier";
            type uint32;
            default "1";
        }
    }
    augment "/ncs:services" {
        list l3mplsvpn {
            tailf:info "Layer-3 MPLS VPN Service";
            key "vpn-name";
            uses ncs:service-data;
            ncs:servicepoint "l3mplsvpn-servicepoint";
            leaf vpn-name {
                tailf:info "Service Instance Name";
                type string;
            }
        }
    }
}

```


CE Device

```
!  
interface Loopback0  
  no shutdown  
  ip address 1.1.1.1 255.255.255.0 <ce_device_lo0_ipaddress>  
!  
  
interface GigabitEthernet3 <ce_to_pe_facing_interface>  
  no shutdown  
  ip address 192.168.12.1 255.255.255.0 <ce_to_pe_facing_interface_ipaddress>  
!  
router eigrp 100  
  network 1.0.0.0 <ce_device_lo0_ipaddress>  
  network 192.168.12.0 <ce_to_pe_facing_interface_ipaddress>  
!  
<?xml version="1.0" encoding="UTF-8"?>  
<devices xmlns="http://tail-f.com/ns/ncs">  
  <device>  
    <name>HQ1</name>  
    <config>  
      <interface xmlns="urn:ios">  
        <Loopback>  
          <name>0</name>  
          <ip>  
            <address>  
              <primary>  
                <address>9.9.9.9</address>  
                <mask>255.255.255.0</mask>  
              </primary>  
            </address>  
          </ip>  
        </Loopback>  
        <GigabitEthernet>  
          <name>1</name>  
          <ip>  
            <no-address>  
              <address xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"  
nc:operation="delete"/>  
            </no-address>  
            <address>  
              <primary>  
                <address>192.168.100.1</address>  
                <mask>255.255.255.0</mask>  
              </primary>  
            </address>  
          </ip>  
            <shutdown xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"  
nc:operation="delete"/>  
          </shutdown>  
        </GigabitEthernet>  
      </interface>  
      <router xmlns="urn:ios">  
        <eigrp>  
          <as-no>100</as-no>  
          <network-ip>  
            <network>  
              <ip>9.9.9.9</ip>  
            </network>  
            <network>  
              <ip>192.168.100.1</ip>  
            </network>  
          </network-ip>  
        </eigrp>  
      </router>  
    </config>  
  </device>  
</devices>
```

```

        </router>
    </config>
</device>
</devices>

```

< An XML Template Dedicated to PE Router - Side A >

PE Device - Side A

```

!
ip vrf CUSTOMER-A <customer_name>
  rd 100:1 <value_x:value_y>
  route-target both <value_y:value_x>
!
!
interface GigabitEthernet4 <pe_to_ce_facing_interface>
  no shutdown
  ip vrf forwarding CUSTOMER-A <customer_name>
  ip address 192.168.100.4 255.255.255.0 <ce_to_pe_facing_interface_ipaddress>
!
!
router eigrp 100
!
  address-family ipv4 vrf CUSTOMER-A <customer_name>
    redistribute bgp 1 metric 1500 400 20 20 1500
    network 192.168.100.4 <ce_to_pe_facing_interface_ipaddress>
    autonomous-system 100
  exit-address-family
!
router bgp 1
  bgp log-neighbor-changes
  neighbor 4.4.4.4 remote-as 1 <side_b_loopback0_ip_address>
  neighbor 4.4.4.4 update-source Loopback0 <side_b_loopback0_ip_address>
!
  address-family vpnv4
    neighbor 4.4.4.4 activate <side_b_loopback0_ip_address>
    neighbor 4.4.4.4 send-community both <side_b_loopback0_ip_address>
  exit-address-family
!
  address-family ipv4 vrf CUSTOMER-A <customer_name>
    redistribute eigrp 100
  exit-address-family
!
<?xml version="1.0" encoding="UTF-8"?>
<devices xmlns="http://tail-f.com/ns/ncs">
  <device>
    <name>SP1</name>
    <config>
      <interface xmlns="urn:ios">
        <GigabitEthernet>
          <name>4</name>
          <ip-vrf>
            <ip>
              <vrf>
                <forwarding>CUSTOMER-A</forwarding>
              </vrf>
            </ip>
          </ip-vrf>
          <ip>
            <no-address>

```

```

        <address xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
nc:operation="delete"/>
    </no-address>
    <address>
        <primary>
            <address>192.168.100.4</address>
            <mask>255.255.255.0</mask>
        </primary>
    </address>
</ip>
    <shutdown xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
nc:operation="delete"/>
</GigabitEthernet>
</interface>
<router xmlns="urn:ios">
    <bgp>
        <as-no>1</as-no>
        <neighbor-tag>
            <neighbor>
                <id>10.10.10.10.</id>
                <remote-as>1</remote-as>
            </neighbor>
        </neighbor-tag>
        <neighbor>
            <id>10.10.10.10</id>
            <update-source>
                <Loopback>0</Loopback>
            </update-source>
        </neighbor>
        <address-family>
            <vpnv4>
                <af>unicast</af>
                <neighbor>
                    <id>10.10.10.10</id>
                    <activate/>
                    <send-community>
                        <send-community-where>both</send-community-where>
                    </send-community>
                </neighbor>
            </vpnv4>
        </address-family>
    </bgp>
    <eigrp>
        <as-no>100</as-no>
        <address-family>
            <ipv4>
                <vrf>
                    <name>CUSTOMER-A</name>
                    <network-ip>
                        <network>
                            <ip>192.168.100.4</ip>
                        </network>
                    </network-ip>
                </vrf>
            </ipv4>
        </address-family>
    </eigrp>
</router>
</config>
</device>
</devices>

```

< An XML Template Dedicated to PE Router - Side B >

PE Device - Side B

```
!  
ip vrf CUSTOMER-A <customer_name>  
  rd 100:1 <value_x:value_y>  
  route-target both <value_y:value_x>  
!  
interface GigabitEthernet3 <pe_to_ce_facing_interface>  
  no shutdown  
  ip vrf forwarding CUSTOMER-A <customer_name>  
  ip address 192.168.45.4 255.255.255.0 <ce_to_pe_facing_interface_ipaddress>  
!  
router eigrp 100  
  !  
  address-family ipv4 vrf CUSTOMER-A <customer_name>  
    redistribute bgp 1 metric 1500 4000 200 10 1500  
    network 192.168.45.0 <ce_to_pe_facing_interface_ipaddress>  
    autonomous-system 100  
  exit-address-family  
!  
router bgp 1  
  bgp log-neighbor-changes  
  neighbor 2.2.2.2 remote-as 1 <side_a_loopback0_ip_address>  
  neighbor 2.2.2.2 update-source Loopback0 <side_a_loopback0_ip_address>  
  !  
  address-family vpnv4  
    neighbor 2.2.2.2 activate <side_a_loopback0_ip_address>  
    neighbor 2.2.2.2 send-community both <side_a_loopback0_ip_address>  
  exit-address-family  
  !  
  address-family ipv4 vrf CUSTOMER-A <customer_name>  
    redistribute eigrp 100  
  exit-address-family  
  !  
  !  
  !
```

#NSO 300 - CISCO NETWORK SERVICES ORCHESTRATOR (NSO) ADVANCED DESIGN - PYTHON NOTES

In the Cisco NSO CLI, examine the operational data for the l3mplsvpn package with the show packages package l3mplsvpn command. **Verify that the Python component for the l3mplsvpn package is successfully loaded.**

```
admin@ncs# show packages package l3mplsvpn  
packages package l3mplsvpn  
  package-version 1.0  
  description      "Generated Python package"  
  ncs-min-version [ 4.6 ]  
  python-package vm-name l3mplsvpn  
  directory        ./state/packages-in-use/1/l3mplsvpn  
  templates        [ l3mplsvpn-template ]  
  component main  
    `application python-class-name l3mplsvpn.main.Main  
    application start-phase phase2`  
  oper-status up
```

The LOG directory is here:

```
/opt/ncs/ncs-run/logs/ncs-python-vm-l3mplsvpn.log
```

6 Cisco NSO and Ansible together

Cisco NSO and Ansible have some similarities and can be used for similar actions, and they also work well together.

Ansible can provide workflows and version-controlled configurations in YAML. It is easy to use. You can use **Ansible Tower** for scheduling and verifying jobs.

Cisco NSO provides a single API to your entire hybrid and multi-vendor network. It has built-in mechanisms to calculate the minimal differences needed to achieve a desired configuration. It offers a full CRUD (create, read, update, and delete) interface for all configurations. It adds network-wide transactions. Configuration is model-driven with YANG, which means that Cisco NSO validates input parameters automatically against provided models to bring even more reliable network operation.

6.1 What is Ansible?

Ansible is an open-source tool that is typically used to automate procedures for configuration deployment, testing, or verification. It is a stateless tool, where no additional software is needed on managed nodes.

Ansible executes a playbook, which is a set of instructions. These instructions are executed by Ansible modules. Modules are abstractions that are typically written to perform a particular task.

Playbooks are executed on a set of nodes that are typically defined in inventory file. Ansible can also use the **Ansible Tower add-on, which includes dashboards, job status pages, scheduling, role-based access control, REST API, and more**. The advantage of using Ansible, is probably also that you can do everything you can do through NSO CLI, but you can use more readable and easy to be understood YAML configuration files. Continue reading for a few examples.

6.2 Cisco NSO Ansible modules

There are five Cisco NSO Ansible modules that you can use in your Ansible playbooks:

- `nso_action`: Executes Cisco NSO actions.
- `nso_config`: Manages Cisco NSO configuration.
- `nso_query`: Queries Cisco NSO using XPath.
- `nso_show`: Displays data from Cisco NSO.
- `nso_verify`: Verifies Cisco NSO configuration.

All modules are executed locally on an Ansible control machine. The modules use JSON-RPC API to talk to Cisco NSO and perform actions. The following parameters that are common to all modules:

- `url`: NSO JSON-RPC URL, for example: `http://localhost:8080/jsonrpc`
- `username`: NSO username
- `password`: NSO password

In most use cases, the three most commonly used modules are: `nso_config`, `nso_show`, and `nso_action`.

6.2.1 The `nso_config` module

The `nso_config` module manages Cisco NSO configuration. You can update any part of the configuration with the module.

The configuration is defined under the `data` parameter. The format of the configuration is YAML, which can be directly translated from Cisco NSO configuration in JSON, as shown in the following example:

```

developer@ncs# show running-config devices device dist-sw01 config vlan 1001 |
display json
{
  "data": {
    "tailf-ncs:devices": {
      "device": [
        {
          "name": "dist-sw01",
          "config": {
            "tailf-ned-cisco-nx:vlan": {
              "vlan-list": [
                {
                  "id": 1001,
                  "name": "cust1"
                }
              ]
            }
          }
        }
      ]
    }
  }
}

```

This configuration translates as the following YAML file (YAML is very close and similar to JSON or XML files):

```

- name: "Create VLAN"
  nso_config:
    url: "http://localhost:8080/jsonrpc"
    username: "username"
    password: "password"
    data:
      tailf-ncs:devices:
        device:
          - name: "dist-sw01"
            config:
              tailf-ned-cisco-nx:vlan:
                vlan-list:
                  - id: 1001
                    name: "cust1"

```

6.2.2 The “Nso_Show” module

The `nso_show` module displays data from Cisco NSO, and can be used to retrieve configuration and operational data. To retrieve required data, you need to specify the `path` parameter. The `path` parameter is a `keypath` expression, which specifies a path to a particular resource. You can find the correct `keypath` syntax in the Cisco NSO CLI with the `display keypath` option, as shown in the following example:

```

developer@ncs# show running-config devices device dist-sw01 config vlan 1001 |
display keypath
/devices/device{dist-sw01}/config/nx:vlan/vlan-list{1001}/name cust1
developer@ncs#

```

The module returns the output in JSON format. You can see the similar output in the Cisco NSO CLI, when displaying the configuration in JSON format.

```

- name: "Get VLAN"
  nso_show:
    url: "http://localhost:8080/jsonrpc"

```

```

username: "username"
password: "password"
path: "/devices/device{dist-sw01}/config/nx:vlan/vlan-list{1001}"
register: "output"

```

```

- name: "Display the output"
  debug:
    var: "output"

```

6.2.3 The `nso_action` module

The `nso_action` module executes Cisco NSO actions and then verifies that the output is as expected. You can use this module for all sorts of actions, including configuration sync, check synchronization, service re-deploy and more. The module accepts the `path` parameter, similar to the `nso_show` module. The parameter is a `keypath` expression to the action. The module also accepts optional `input` parameter, which accepts action input parameters.

The following is an example of an action that is executed with Ansible:

```

- name: "Sync device configuration to Cisco NSO"
  nso_action:
    url: "http://localhost:8080/jsonrpc"
    username: "username"
    password: "password"
    path: "/ncs:devices/device{dist-sw01}/sync-from"

```

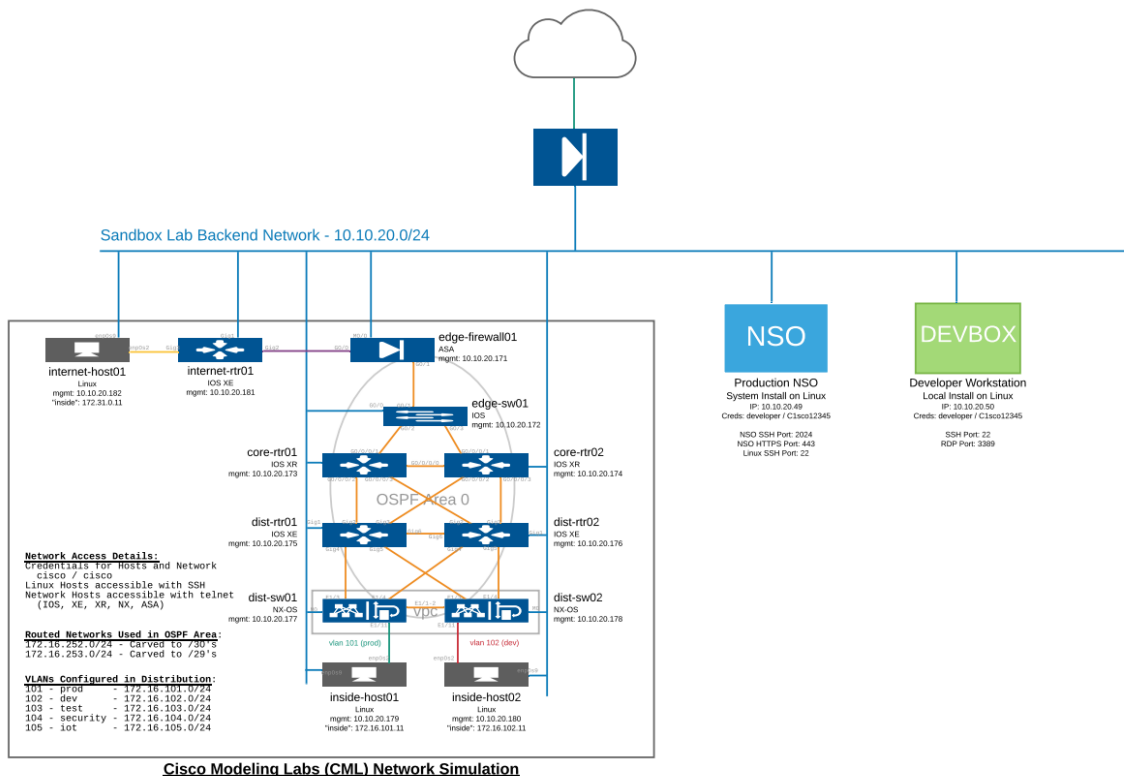
6.3 Lab setup

Before you begin, you will need to set up the environment on a DevNet NSO Sandbox.

6.3.1 Reserving the Sandbox

You will need to reserve an instance of the [DevNet NSO Sandbox](#). The following diagram shows the topology of the NSO sandbox.

DevNet Sandbox Topology: Cisco Network Services Orchestrator (NSO)



The sandbox will take approximately 15 minutes to provision. When the sandbox is done provisioning, you will receive a "Lab is Ready" email that includes access instructions and credentials.

You will be performing the tasks in this lab from the Developer Workstation. Once connected to the Cisco Anyconnect VPN, initiate an SSH session to the Developer Workstation 10.10.20.50 using username `developer` and password `Cisco12345`.

6.3.2 Setting up Cisco NSO and local environment

After you log into the Developer Workstation, you will run the setup script to prepare Cisco NSO and local environment.

```
(py3venv) [developer@devbox ~]$ git clone https://github.com/CiscoDevNet/lab-ansible-nso.git
(py3venv) [developer@devbox ~]$ cd lab-ansible-nso
(py3venv) [developer@devbox lab-ansible-nso]$ sh setup.sh
```

The setup process takes takes about a minute to complete.

6.3.3 Examine Cisco NSO

Cisco NSO is available at 10.10.20.49. You can initiate an SSH session directly to the Cisco NSO CLI (`developer/Cisco12345`):

```
(py3venv) [developer@devbox lab-ansible-nso]$ ssh developer@10.10.20.49 -p 2024
Warning: Permanently added '[10.10.20.49]:2024' (ED25519) to the list of known hosts.
developer@10.10.20.49's password:

User developer last logged in 2021-03-12T03:07:40.043871-08:00, to nso, from 10.10.20.50 using cli-ssh
developer connected from 10.10.20.50 using ssh on nso
developer@ncs#
```

There should be no devices deployed on Cisco NSO. Even though they are in the sandbox when it is spun up, we remove them as part of this lab to demonstrate adding devices to NSO:

```
developer@ncs# show devices list
NAME ADDRESS DESCRIPTION NED ID ADMIN STATE
-----
developer@ncs#
```

However you should see service packages and device driver packages (NEDs):

```
developer@ncs# show packages package oper-status
```

PACKAGE		PROGRAM						
META	FILE	CODE		JAVA	PYTHON		BAD NCS	
PACKAGE	PACKAGE	CIRCULAR	DATA	LOAD	ERROR			
NAME		UP	ERROR	UNINITIALIZED	UNINITIALIZED	VERSION	NAME	
VERSION	DEPENDENCY	ERROR	ERROR	INFO				
cisco-asa-cli-6.12		X	-	-	-	-	-	-
-	-	-	-	-				
cisco-ios-cli-6.67		X	-	-	-	-	-	-
-	-	-	-	-				


```

cisco-iosxr-cli-7.32  X  -  -  -  -  -  -
-  -  -  -  -  -  -  -
cisco-nx-cli-5.20    X  -  -  -  -  -  -
-  -  -  -  -  -  -  -
resource-manager     X  -  -  -  -  -  -
-  -  -  -  -  -  -  -
selftest             X  -  -  -  -  -  -
-  -  -  -  -  -  -  -
svi_verify_example   X  -  -  -  -  -  -
-  -  -  -  -  -  -  -

```

```

developer@ncs# exit
Connection to 10.10.20.49 closed.
(py3venv) [developer@devbox lab-ansible-nso]$

```

6.3.4 Examine the local environment

Playbooks that you will use during the lab are already prepared. You can find these in `/home/developer/ansible`.

```

(py3venv) [developer@devbox lab-ansible-nso]$ cd /home/developer/ansible/
(py3venv) [developer@devbox ansible]$ tree

```

```

.
├── add_devices.yml
├── ansible.cfg
├── export_configuration.yml
├── get_version.yml
├── host_vars
│   ├── dist-sw01.yml
│   └── dist-sw02.yml
├── import_configuration.yml
├── inventory
├── manage_loopbacks.yml
├── manage_svi_service.yml
├── remove_devices.yml
└── services
    ├── service1.yml
    └── service2.yml

```

```

2 directories, 13 files
(py3venv) [developer@devbox ansible]$

```

You will examine and run these playbooks during the lab.

6.4 Step 1: Manage Cisco NSO devices with Ansible

In this step you will learn how to use Ansible to add or remove devices in Cisco NSO configuration. Devices are defined in the Ansible inventory file, called `inventory`. You will use the following Ansible playbooks:

- `add_devices.yml`: Adds devices from the inventory to the Cisco NSO configuration.
- `remove_devices.yml`: Removes devices from the Cisco NSO configuration.

6.4.1 Explore the inventory

The inventory is defined in the `inventory` file. In this example, the inventory is defined in `INI` format. Consult the Ansible documentation to explore different formats that you can use for the inventory. Explore the content of the file. You can use the `cat inventory` command to display the content. Devices are specified as a list. The hostname is automatically stored into Ansible variable `inventory_hostname` during execution. Each device has an IP address, defined with the `ansible_host` variable.

Each device belongs to a group, which is indicated with the `[GROUP]` syntax. For example, IOS-XR routers are members of the group called `iosxr`. There are user defined groups, like `iosxr`, `ios`, `nxos`, and `asa`. There is also the default group called `all`. All devices are members of this group.

You can assign one or many variables to each host. You can assign variables on group or host level. To specify a variables on a group level, Ansible uses the following syntax: `[<GROUP>:vars]`.

You can find a couple of variables that are already defined in each group:

- `ned`: NED ID that will be used for devices. The value of the variable is different for each device type.
- `protocol`: Protocol that will be used by NED to manage a device.
- `authgroup`: The Cisco NSO authentication group.
- `nso_url`: The URL which is used by Cisco NSO Ansible modules (Ansible modules use JSON RPC API for communication with Cisco NSO).
- `nso_username`: The Cisco NSO username.
- `nso_password`: The Cisco NSO password.
- `ansible_connection`: Cisco NSO Ansible modules uses HTTP-based API for communication. The connection is established from Ansible control machine, which means that you can use the `local` connection.

Note: A NED ID may change as the sandbox or your environment is updated. You can find the current NED ID in use with Cisco NSO. You will need to SSH to NSO and use the command `show packages package oper-status`. Open the following section to see an example of how to get the correct NED ID.

6.4.2 Explore the `add_devices.yml` playbook

Use the `cat add_devices.yml` command to display the file. Explore the contents of the playbook.

```
(py3venv) [developer@devbox ansible]$ cat add_devices.yml
---
- name: "Add devices"
  hosts: all
  gather_facts: no

  vars:
    device_path: "/devices/device{% raw %}{{% endraw %}}{{ inventory_hostname
}}{% raw %}{{% endraw %}}/"

  tasks:

    - name: "Add devices to NSO configuration"
      nso_config:
        url: "{{ nso_url }}"
        username: "{{ nso_username }}"
        password: "{{ nso_password }}"
        data:
          devices:
            device:
              - name: "{{ inventory_hostname }}"
                address: "{{ ansible_host }}"
                authgroup: "{{ authgroup }}"
                device-type:
                  cli:
                    ned-id: "{{ ned }}"
                    protocol: "{{ protocol }}"
                state:
                  admin-state: "unlocked"

    - name: "Fetch SSH keys"
      nso_action:
```

```

    url: "{{ nso_url }}"
    username: "{{ nso_username }}"
    password: "{{ nso_password }}"
    path: "{{ device_path }}/ssh/fetch-host-keys"
    when: "protocol == 'ssh'"

- name: "Sync configuration"
  nso_action:
    url: "{{ nso_url }}"
    username: "{{ nso_username }}"
    password: "{{ nso_password }}"
    path: "{{ device_path }}/sync-from"

```

The playbook includes three tasks, (and remember all of the variables are defined outside the playbook to make the playbook versatile and simple):

- the first task adds devices to the Cisco NSO configuration with the `nso_config` module. The configuration is defined under the `data` parameter. In this particular task, you need to define device name, IP address, an authentication group, device type, and state.
- the second task fetches SSH keys from devices with the `nso_action` module. The task is executed only for devices that use `ssh` as a connection protocol. You need to specify the path to the action. The prefix `device_path` is already defined in the `vars` section in the header. As this will be used multiple times, it makes sense to put into a variable. Note that the variable parameter uses double quotation marks " " around each parameter. This is required, because otherwise Jinja2 syntax misinterprets the `{` and `}` as part of the expression.
- the third task syncs a configuration from a device. The task uses the `nso_action` module in a similar way to the second task.

The playbook includes a playbook level variable defined at the top of the playbook, rather than in the inventory:

```

vars:
  device_path: "/devices/device{% raw %}{{ endraw %}}{{ inventory_hostname }}{% raw %}{{ endraw %}}/"

```

The `device_path` variable uses some additional Jinja2 syntax help with the `{% raw %}` and `{% endraw %}` statements to "escape" the character `{` and `}` which are in the path. Since Jinja2 uses the characters `{}`, we need to tell Jinja2 in this case to not interpret them as Jinja2 characters, but just as normal curly brackets.

Ansible will execute all tasks in parallel for each device in the inventory.

6.4.3 Run the playbook

Now that you have explored both inventory and playbook, you are ready to execute the playbook. The command starts the playbook and executes the defined tasks

Use the following command:

```
ansible-playbook -i inventory add_devices.yml
```

The output shows the tasks that were executed and their statuses. At the end, you can see the recap for each device. Since all tasks are executed successfully, you can connect to Cisco NSO and verify the configuration to see that devices are present.

6.4.4 Connect to Cisco NSO and verify that devices are added

To verify that devices are present, connect to Cisco NSO. Initiate an SSH connection to Cisco NSO CLI with the `ssh developer@10.10.20.49 -p 2024` command and verify that you can see devices. Remember, use the password `Cisco12345`.

```
(py3venv) [developer@devbox ansible]$ ssh developer@10.10.20.49 -p 2024
Warning: Permanently added '[10.10.20.49]:2024' (ED25519) to the list of known
hosts.
developer@10.10.20.49s password:

User developer last logged in 2021-03-09T04:33:58.743446-08:00, to nso, from
10.10.20.50 using webui-https
developer connected from 10.10.20.50 using ssh on nso
developer@ncs#
```

After you are connected, you can use the `show devices list` command, which displays all configured devices. You should see your devices in the output.

```
developer@ncs# show devices list
NAME                ADDRESS          DESCRIPTION      NED ID              ADMIN STATE
-----
core-rtr01          10.10.20.173    -               cisco-iosxr-cli-7.32 unlocked
core-rtr02          10.10.20.174    -               cisco-iosxr-cli-7.32 unlocked
dist-rtr01           10.10.20.175    -               cisco-ios-cli-6.67  unlocked
dist-rtr02           10.10.20.175    -               cisco-ios-cli-6.67  unlocked
dist-sw01            10.10.20.177    -               cisco-nx-cli-5.20   unlocked
dist-sw02            10.10.20.178    -               cisco-nx-cli-5.20   unlocked
edge-firewall01     10.10.20.171    -               cisco-asa-cli-6.12  unlocked
developer@ncs# exit
Connection to 10.10.20.49 closed.
(py3venv) [developer@devbox ansible]$
```

6.4.5 Explore the `remove_devices.yml` playbook

Sometimes, you want to remove devices from the configuration as well. Explore the content of the `remove_devices.yml` playbook, which is used to remove devices from Cisco NSO configuration.

```
(py3venv) [developer@devbox ansible]$ cat remove_devices.yml
---
- name: "Remove devices"
  hosts: all
  gather_facts: no

  tasks:

    - name: "Remove devices from the NSO configuration"
      nso_config:
        url: "{{ nso_url }}"
        username: "{{ nso_username }}"
        password: "{{ nso_password }}"
        data:
          devices:
            device:
              - name: "{{ inventory_hostname }}"
                __state: "absent"
```

There is only a single task in the playbook. The task uses the `nso_config` module, with the same structure as you would use in the Cisco NSO CLI. The `nso_config` module uses `__state` parameter on list

elements, to define the required state of an element. If you put `__state: "absent"`, this tells the `nso_config` module to remove the list element if exists.

6.4.6 Run the playbook

To test the playbook, execute the playbook, which will remove devices from Cisco NSO configuration. Use the following command:

```
ansible-playbook -i inventory remove_devices.yml
```

You should notice the status of the task as `changed`, which means that devices are removed.

6.4.7 Connect to Cisco NSO and verify that devices are actually removed

To make sure that the playbook works correctly, verify that devices are no longer present in the configuration. Initiate an SSH connection to Cisco NSO.

```
(py3venv) [developer@devbox ansible]$ ssh developer@10.10.20.49 -p 2024
Warning: Permanently added '[10.10.20.49]:2024' (ED25519) to the list of known
hosts.
developer@10.10.20.49s password:

User developer last logged in 2021-03-09T04:44:29.869605-08:00, to nso, from
10.10.20.50 using webui-https
developer connected from 10.10.20.50 using ssh on nso
developer@ncs#
```

Use the `show devices list` command to see configured devices.

```
developer@ncs# show devices list
NAME  ADDRESS  DESCRIPTION  NED ID  ADMIN STATE
-----
developer@ncs# exit
Connection to 10.10.20.49 closed.
(py3venv) [developer@devbox ansible]$
```

As all devices were removed, you can only see an empty list.

6.4.8 Put devices back to the configuration

As you will need devices in the next steps in the lab, you need to add devices back to the configuration.

Execute the `add_devices.yml` playbook again.

Run the following command:

```
ansible-playbook -i inventory add_devices.yml
```

After the playbook is executed, devices are added back to the Cisco NSO configuration.

6.5 Step 2: Manage device configurations

You can use Cisco NSO to create, read, update, or delete (CRUD) device configurations. Cisco NSO can serve as a single API to access your network device configurations. In this step, you will leverage Ansible to manage device configurations with Cisco NSO. You will work on NX-OS devices in this task. In particular, you will configure, read, update, and delete loopback interfaces, but this example could be extended to any part of a device configuration. There is only a single playbook that is used for all operations. It is called `manage_loopbacks.yml`.

6.5.1 Explore the `manage_loopbacks.yml` playbook

Before you manage any configuration on Cisco NSO, review the content of the `manage_loopbacks.yml` playbook.

```
(py3venv) [developer@devbox ansible]$ cat manage_loopbacks.yml
```

```
---
- name: "Manage loopback interfaces"
  hosts: nxos
  gather_facts: no

  vars:
    device_path: "/devices/device{% raw %}{{% endraw %}}{{ inventory_hostname
}}{% raw %}{{% endraw %}}/"

  tasks:

    - name: "Add loopback interfaces"
      nso_config:
        url: "{{ nso_url }}"
        username: "{{ nso_username }}"
        password: "{{ nso_password }}"
        data:
          devices:
            device:
              - name: "{{ inventory_hostname }}"
                config:
                  interface:
                    loopback:
                      - name: "{{ item.id }}"
                        description: "Configured by Ansible"
      loop: "{{ loopbacks }}"
      when: "item.state|default('present') == 'present'"

    - name: "Configure IP address on loopback interfaces"
      nso_config:
        url: "{{ nso_url }}"
        username: "{{ nso_username }}"
        password: "{{ nso_password }}"
        data:
          devices:
            device:
              - name: "{{ inventory_hostname }}"
                config:
                  interface:
                    loopback:
                      - name: "{{ item.id }}"
                        ip:
                          address:
                            ipaddr: "{{ item.ip_address }}/32"
      loop: "{{ loopbacks }}"
      when: "item.state|default('present') == 'present'"

    - name: "Remove loopback interfaces"
      nso_config:
        url: "{{ nso_url }}"
        username: "{{ nso_username }}"
        password: "{{ nso_password }}"
        data:
          devices:
```

```

        device:
            - name: "{{ inventory_hostname }}"
              config:
                interface:
                  loopback:
                    - name: "{{ item.id }}"
                      __state: "absent"
loop: "{{ loopbacks }}"
when: "item.state|default('present') == 'absent'"

- name: "Read configuration for loopback interfaces"
  nso_show:
    url: "{{ nso_url }}"
    username: "{{ nso_username }}"
    password: "{{ nso_password }}"
    path: "{{ device_path }}/config/interface/loopback"
    register: "loopback_config"

- name: "Print configuration"
  debug:
    var: "loopback_config"

```

Let's break down the playbook:

- The first two tasks configure a loopback interface and configure IP address on the interface. In the first task, you create all required loopback interfaces.
- In the second task you configure IP address on the loopback interfaces. Two tasks are required, because a loopback interface is a list element in the NED Yang, which you need to create before you can configure it. Both tasks loop over the `loopbacks` variable, which is a list of loopbacks with several variables, as you will see later.
- The third task is used to remove a loopback interface. The interface is removed only when the `state` variable is set to the value `absent`. Notice that the `__state` parameter is used to remove the list element.
- The final two tasks read Cisco NSO configuration and print it out. To get the configuration, the task uses the `nso_show` module. To print the output, the task uses `debug` module.

6.5.2 Verify the configuration for interfaces

To configure interfaces, you have to provide the configuration. Some basic configuration is already prepared for you under the `host_vars` folder.

```

(py3venv) [developer@devbox ansible]$ ls host_vars/
dist-sw01.yml  dist-sw02.yml
(py3venv) [developer@devbox ansible]$

```

There is one file for each device. Display the configuration for the first device.

```

(py3venv) [developer@devbox ansible]$ cat host_vars/dist-sw01.yml
---
loopbacks:
  - id: 101
    ip_address: "192.168.101.1"
    state: "present"
  - id: 102
    ip_address: "192.168.102.1"
    state: "present"

```

You can see the variable called `loopbacks`, which is a list of dictionaries. Each dictionary has the following parameters:

- `id`, which defines a loopback interface ID
- `ip_address`, which defines an IP address for an interface
- `state`, which defines if an interface should be present or absent

These variables are used to control how to configure devices.

6.5.3 Execute the playbook

Now, that you are familiar with the playbook and the configuration, you can execute the playbook. Use the following command:

```
ansible-playbook -i inventory manage_loopbacks.yml
```

You should see that the playbook is successfully executed. The first part of the output shows the configuration tasks. The last part of the output shows the configuration of loopback interfaces on a device. The configuration is displayed in `JSON` format. Since `JSON` is directly convertible to a Python dictionary, you can access any parameter without parsing. This is a huge benefit, because the configuration is presented as structured data.

6.5.4 Modify the configuration

You can re-run the playbook multiple times with the different configurations.

To update the configuration, edit the file `host_vars/dist-sw01.yml`. Edit the configuration file for the `dist-sw01` switch by changing the `state` variable to `absent` for the loopback interface with the ID 102. This will remove the interface from the configuration.

Configuration before:

```
---
loopbacks:
  - id: 101
    ip_address: "192.168.101.1"
    state: "present"
  - id: 102
    ip_address: "192.168.102.1"
    state: "present"
```

Configuration after:

```
---
loopbacks:
  - id: 101
    ip_address: "192.168.101.1"
    state: "present"
  - id: 102
    ip_address: "192.168.102.1"
    state: "absent"
```

The state of the interface with the ID 102 is changed to `absent`.

6.5.5 Execute the playbook

Now, that the configuration is updated, execute the playbook again. This will remove the interface from the device. Use the following command:

```
ansible-playbook -i inventory manage_loopbacks.yml
```


When you look at the output, you can see that the line for the interface 102 in the third task has state changed, which means that configuration was updated. The configuration output in the last task shows that the interface `loopback 102` is no longer present in the configuration.

6.6 Step 4: Gather live status from devices

Cisco NSO enables you to execute `show` commands and more on devices from Cisco NSO. In this step, you will use Ansible to trigger the `show version` command on your devices from Cisco NSO. The output will be saved to a file. You will use the `get_version.yml` playbook.

6.6.1 Explore the `get_version.yml` playbook

To demonstrate how you can execute `show` commands from Cisco NSO, you will execute the `show version` command, which is accepted by all devices in the sandbox. But before you retrieve the output of the `show version` command from devices, observe the content of the `get_version.yml` playbook.

```
(py3venv) [developer@devbox ansible]$ cat get_version.yml
---
- name: "Get software version from devices"
  hosts: all
  gather_facts: no

  vars:
    device_path: "/devices/device{% raw %}{{% endraw %}}{{ inventory_hostname
}}{% raw %}{{% endraw %}}/"

  tasks:

    - name: "Get software version from devices using live status"
      nso_action:
        url: "{{ nso_url }}"
        username: "{{ nso_username }}"
        password: "{{ nso_password }}"
        path: "{{ device_path }}/live-status/exec/show"
        input:
          args: "version"
        register: "show_version"

    - name: "Create directory for outputs"
      file:
        path: "./outputs"
        state: "directory"

    - name: "Save the output to a file"
      copy:
        content: "{{ show_version.output.result }}"
        dest: "outputs/version_{{ inventory_hostname }}.txt"
```

The playbook has three tasks:

- the first task uses the `nso_action` module, which executes the `show version` command on devices. You have to specify a path and input arguments. The input argument should be a command without the `show` keyword. In this case, the input argument is `version`. The task also stores the output to a variable called `show_version`.
- the second task creates a directory called `outputs` if this directory does not exist.
- the last task saves the output of the command to a file. The file includes device's name.

6.6.2 Execute the playbook

Execute the playbook to save the output of the `show version` command. Use the following command:

```
ansible-playbook -i inventory get_version.yml

(py3venv) [developer@devbox ansible]$ ansible-playbook -i inventory
get_version.yml
PLAY [Get software version from devices] *

TASK [Get software version from devices using live status] ***
changed: [edge-firewall101]
changed: [dist-rtr01]
changed: [dist-rtr02]
changed: [core-rtr02]
changed: [core-rtr01]
changed: [dist-sw01]
changed: [dist-sw02]

TASK [Create directory for outputs] **
ok: [core-rtr01]
changed: [dist-rtr01]
ok: [core-rtr02]
ok: [edge-firewall101]
ok: [dist-rtr02]
ok: [dist-sw01]
ok: [dist-sw02]

TASK [Save the output to a file] *
changed: [dist-rtr02]
changed: [edge-firewall101]
changed: [core-rtr01]
changed: [core-rtr02]
changed: [dist-rtr01]
changed: [dist-sw02]
changed: [dist-sw01]

PLAY RECAP ***
core-rtr01 : ok=3    changed=2    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
core-rtr02 : ok=3    changed=2    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
dist-rtr01 : ok=3    changed=3    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
dist-rtr02 : ok=3    changed=2    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
dist-sw01  : ok=3    changed=2    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
dist-sw02  : ok=3    changed=2    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
edge-firewall101 : ok=3    changed=2    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

As you can see from the output, the playbook retrieves the output of the `show version` command, creates the directory for outputs, and save the output to a file.

6.6.3 Verify the outputs

After the playbook is executed, the outputs are saved to the `outputs` directory.

```
(py3venv) [developer@devbox ansible]$ ls outputs/
```

```
version_core-rtr01.txt  version_dist-rtr01.txt  version_dist-sw01.txt
version_edge-firewall01.txt
version_core-rtr02.txt  version_dist-rtr02.txt  version_dist-sw02.txt
```

Verify the outputs for different device types. Because you have different device types, the outputs are different, but still the result of the same command (`show version`).

6.7 Step 5: Manage service instances with Ansible

Since service instances are also a part of the configuration in Cisco NSO, you can manage them with the same techniques you used for devices.

In this step you will work with the `svi_verify_example` package, which is pre-installed on Cisco NSO in this sandbox. It is a sample service package, which creates a VLAN and an SVI interface on a device.

The playbook that you will use is called `manage_svi_service.yml`. The playbook is used to create, update, and remove service instances.

6.7.1 Explore the `manage_svi_service.yml` playbook

Review the content of the `manage_svi_service.yml` playbook.

```
(py3venv) [developer@devbox ansible]$ cat manage_svi_service.yml
---
- name: "Manage SVI service instances"
  hosts: localhost
  gather_facts: no

  tasks:

    - name: "Load service configuration files"
      include_vars:
        dir: "./services"
        name: "services"

    - name: "Create service instances"
      nso_config:
        url: "{{ nso_url }}"
        username: "{{ nso_username }}"
        password: "{{ nso_password }}"
        data:
          svi_verify_example:
            - name: "{{ item.key }}"
              vlan-id: "{{ item.value.vlan.id }}"
              vlan-name: "{{ item.value.vlan.name }}"
              switches: "{{ item.value.switches }}"
      loop: "{{ services | dict2items }}"
      when: "item.value.status == 'present'"

    - name: "Delete service instances"
      nso_config:
        url: "{{ nso_url }}"
        username: "{{ nso_username }}"
        password: "{{ nso_password }}"
        data:
          svi_verify_example:
            - name: "{{ item.key }}"
              __state: "absent"
      loop: "{{ services | dict2items }}"
      when: "item.value.status == 'absent'"

```

The playbook has three tasks:

- the first task loads configuration files. It uses the `include_vars` module. The task loads YAML files from the `./services` directory and saves the content as a variable called `services`.
- the second task configures a service instance. It loops over all instances from configuration files and configures them one by one. A service instance is configured only if the `status` variable is set to `present`. The loop parameter uses the `dict2items` Jinja2 filter, which transforms a dictionary into a list of items suitable for looping. There are a couple of parameters that need to be configured:
 - `name`: Service instance name
 - `vlan-id`: VLAN ID
 - `vlan-name`: VLAN name
 - `switches`: A list of switches where VLAN and SVI should be configured
- the third task deletes the service instance when the `status` variable is set to `absent`.

6.7.2 Explore service instance configuration files

Configuration files are already included. You can find these in the `services` directory. There are two configuration files, one for each service instance.

```
(py3venv) [developer@devbox ansible]$ ls services/  
service1.yml  service2.yml
```

Verify the configuration files.

```
services/service1.yml:  
---  
service-vlan201:  
  status: "present"  
  vlan:  
    id: 201  
    name: "test-vlan201"  
  switches:  
    - id: "sw01"  
      switch-device: "dist-sw01"  
services/service2.yml:  
---  
service-vlan202:  
  status: "present"  
  vlan:  
    id: 202  
    name: "test-vlan202"  
  switches:  
    - id: "sw02"  
      switch-device: "dist-sw02"
```

Each service instance has a name and other required parameters. Take a moment and check how these parameters map to the second task in the `manage_svi_service.yml` playbook.

6.7.3 Configure service instances

To apply the service configuration, run the playbook `manage_svi_service.yml`. The playbook will load the configurations and apply these configurations to Cisco NSO. Cisco NSO will then apply the required configuration to devices based on the mapping logic.

Use the following command:

```
ansible-playbook -i inventory manage_svi_service.yml
```

Since the `status` for both services are `present`, the playbook has only executed the first task. You will see later how to delete service instances.

6.7.4 Verify the configuration on Cisco NSO

Verify the configuration on Cisco NSO. First, initiate an SSH connection to Cisco NSO.

```
(py3venv) [developer@devbox ansible]$ ssh developer@10.10.20.49 -p 2024
Warning: Permanently added '[10.10.20.49]:2024' (ED25519) to the list of known
hosts.
developer@10.10.20.49's password:
```

```
User developer last logged in 2021-03-12T00:58:55.49533-08:00, to nso, from
192.168.254.15 using cli-ssh
developer connected from 10.10.20.50 using ssh on nso
developer@ncs#
```

Use the `show running-config svi_verify_example` command to display the service configuration.

```
developer@ncs# show running-config svi_verify_example
svi_verify_example service-vlan201
  switches sw01
    switch-device dist-sw01
  !
  vlan-id 201
  vlan-name test-vlan201
!
svi_verify_example service-vlan202
  switches sw02
    switch-device dist-sw02
  !
  vlan-id 202
  vlan-name test-vlan202
!
```

You can see two instances, each with the configuration as specified in your configuration files. Exit the Cisco NSO CLI with the `exit` command.

```
developer@ncs# exit
Connection to 10.10.20.49 closed.
(py3venv) [developer@devbox ansible]$
```

6.7.5 Modify the service instance configuration

You can also use the same playbook to delete service instances. To demonstrate that, modify the configuration in the `services/service1.yml` file. Change the `status` parameter to `absent`. Before the change:

```
---
service-vlan201:
  status: "present"
  vlan:
    id: 201
    name: "test-vlan201"
  switches:
    - id: "sw01"
      switch-device: "dist-sw01"
```

After the change:

```
---
```

```
service-vlan201:
  status: "absent"
  vlan:
    id: 201
    name: "test-vlan201"
  switches:
    - id: "sw01"
      switch-device: "dist-sw01"
```

You can notice that the `status` variable was changed from `present` to `absent`.

6.7.6 Remove the service instance

Now, that configuration was updated, you can execute the playbook `manage_svi_service.yml` again. Use the following command:

```
ansible-playbook -i inventory manage_svi_service.yml
```

If you focus on the output, you can see the `changed` status in the second task, which indicates that the service instance was removed.

6.7.7 Verify the configuration on Cisco NSO

To verify the service configuration, initiate an SSH connection to Cisco NSO.

```
(py3venv) [developer@devbox ansible]$ ssh developer@10.10.20.49 -p 2024
Warning: Permanently added '[10.10.20.49]:2024' (ED25519) to the list of known
hosts.
developer@10.10.20.49's password:

User developer last logged in 2021-03-12T00:58:55.49533-08:00, to nso, from
192.168.254.15 using cli-ssh
developer connected from 10.10.20.50 using ssh on nso
developer@ncs#
```

Use the `show running-config svi_verify_example` command to display the service configuration on Cisco NSO.

```
developer@ncs# show running-config svi_verify_example
svi_verify_example service-vlan202
  switches sw02
    switch-device dist-sw02
  !
  vlan-id 202
  vlan-name test-vlan202
  !
```

There is now only one service instance in the Cisco NSO configuration as you would expect. Exit the Cisco NSO CLI with the `exit` command.

```
developer@ncs# exit
Connection to 10.10.20.49 closed.
(py3venv) [developer@devbox ansible]$
```

7 Making changes

Once you've learned how to navigate the data, RESTCONF makes it easy to change. In line with the REST API principles, you will use different HTTP request methods to tell NSO what to do.

At this point you will need the [NSO reservable Sandbox](#) instance if you want the following requests to succeed. The always-on instance you've been using so far is read-only and disallows changes.

Note: The NSO server address will change from here on and examples will require an established VPN connection to work.

In addition, you will need to change the configured credentials in Postman. The new server requires the username 'developer' and the password 'C1sco12345', which you should set on the authorization tab of the collection.

7.1 Setting configuration with PUT

In the same way the YANG model defines data paths for navigation, it also defines the field names and constraints on the content of the RESTCONF messages. Therefore, all you really need in the way of RESTCONF API specification is the YANG model. But not everyone knows how to read YANG, so below you'll take a bit different approach.

In RESTCONF, data conforms to the YANG model for both, the requests you send to the server and the responses you get back. In other words, what you read from NSO is already according to the model, so you can just use that same data for making changes.

The one thing you have to consider, though, is to not send operational data as part of your change.

Remember how you used the `?content=config` query string to filter out the non-configuration data before? That's a good way to construct the data you'll need.

Next, you'll configure a device group with this method on your private lab server at <https://10.10.20.49>.

The data from a previous GET request for device group *IOS-DEVICES* will serve as the basis for the new one.

```
GET /restconf/data/tailf-ncs:devices/device-group=IOS-DEVICES?content=config
{
  "tailf-ncs:device-group": [
    {
      "name": "IOS-DEVICES",
      "device-name": [
        "dist-rtr01",
        "dist-rtr02",
        "internet-rtr01"
      ]
    }
  ]
}
```

Create a new request in Postman, only this time select the PUT method, instead of GET. Set the address to <https://10.10.20.49/restconf/data/tailf-ncs:devices/device-group=IOS-DEVICES> for the new server.

Copy the above response body into the body of the PUT request, using the *raw* type instead of *none* in Postman. If you did not duplicate an existing request, do not forget to set the Accept and Content-Type headers as well. The full HTTP request should resemble the following:

```
PUT https://10.10.20.49/restconf/data/tailf-ncs:devices/device-group=IOS-DEVICES
Accept: application/yang-data+json
Content-Type: application/yang-data+json

{
  "tailf-ncs:device-group": [
    {
      "name": "IOS-DEVICES",
      "device-name": [
        "dist-rtr01",
        "dist-rtr02",
        "internet-rtr01"
      ]
    }
  ]
}
```

```
}  
]
```

When sending the request, it is likely Postman will issue a warning about the server certificate. This is expected, as the server uses a self-signed certificate. Please click the **Disable SSL Verification** button to disable verification for the duration of this lab.

Look for a *201 Created* response. If you try to send this request multiple times, you may see a *204 No Content* response instead, since the group already exists. In case of errors, please check you are using the right, updated credentials.

You can use this same request later on, if you want to make changes to this device group. To add or remove devices from the group, simply add or remove them from the `device-name` field and re-send the request. Try it now by removing the *internet-rtr01* router.

```
PUT https://10.10.20.49/restconf/data/tailf-ncs:devices/device-group=IOS-DEVICES  
Accept: application/yang-data+json  
Content-Type: application/yang-data+json
```

```
{  
  "tailf-ncs:device-group": [  
    {  
      "name": "IOS-DEVICES",  
      "device-name": [  
        "dist-rtr01",  
        "dist-rtr02"  
      ]  
    }  
  ]  
}
```

If the request fails, please make sure you have removed the comma after "dist-rtr02", too, to pass JSON validation.

According to the REST principles, the PUT method creates or replaces a resource. It allows you to remove as well as add devices to the *IOS-DEVICES* device group. In addition to PUT, NSO and RESTCONF also support the POST, PATCH, and DELETE methods for manipulating data.

As the name suggests, you would use the DELETE method to remove configuration at the specified URL, just like using a **no ...** command on the NSO CLI.

PATCH is similar to PUT. PUT will always set an element to the provided value, removing existing data if there is any. PATCH, on the other hand, will merge the provided and existing data.

POST is perhaps the most complex. It has multiple uses. For example, it is used to create new list items that must not yet exist. If the item exists already, a POST request will fail. Another common use-case is requesting execution of various actions and operations.

8 Invoking actions

Actions are separate from configuration and are typically used for things such as initiating a device restart (reboot) or running commands on a remote device. In NSO, they are used to perform different maintenance tasks.

NSO keeps a copy of device configurations locally to speed up processing. This copy has to be kept in sync with the actual configuration on managed devices and NSO implements a few actions for this purpose. The **check-sync** action connects to a remote device and verifies if the configuration still matches. For example, to check configurations for all devices in the *IOS-DEVICES* group, you would run the following command on the NSO CLI:


```
admin@ncs# devices device-group IOS-DEVICES check-sync
```

To do the same through RESTCONF, you follow a similar process to what you did for reading data. You construct the target URL in the same way. Then you send a POST request and the RESTCONF server returns the output of the action. Perhaps you are wondering why not just use a GET request like before? The reason for the POST method is that the action might take some parameters and you would add those in the request body. The simple form of **check-sync** doesn't take any parameters, making it the ideal candidate for your first action call.

In Postman, create a new request with the right Accept headers, set the method to POST and use <https://10.10.20.49/restconf/data/taif-ncs:devices/device-group=IOS-DEVICES/check-sync> as the URL. Send the request to the server and wait a bit for a response. It takes longer because NSO has to connect to remote devices and check the configuration there. In the end, you should see the output similar to the following:

```
POST /restconf/data/taif-ncs:devices/device-group=IOS-DEVICES/check-sync
{
  "taif-ncs:output": {
    "sync-result": [
      {
        "device": "dist-rtr01",
        "result": "in-sync"
      },
      {
        "device": "dist-rtr02",
        "result": "in-sync"
      }
    ]
  }
}
```

You can also try and add { "taif-ncs:suppress-positive-result": [null] } as the request body to execute the action with `suppress-positive-result` parameter. The response will then contain only out-of-sync devices.

9 Netsim for NSO

9.1 Purpose

Netsim-tool enables you to easily add new netsim networks, multiple devices to existing networks and most of other commands otherwise available with *ncs-netsim* command. List of available commands below.

- create-network
- create-device
- delete-network
- add-device
- start
- stop
- is-alive
- list
- load
- update-network

9.2 Documentation

Apart from this README, please refer to the python code and associated YANG file.

9.3 Dependencies

- NSO 4.3.6+ Local installation (Probably works on older versions aswell but not tested)
- Python 2.7+ or 3+
- Python Popen module

9.4 Build instructions

```
make -C /packages/netsim-tool/src clean all
```

9.5 Usage examples

9.5.1 config

User can configure default ports of running netsim processes and netsim-dir, which represents the folder where netsim network and devices are created. In this way it's easy to switch between multiple netsim networks. Users should specify absolute path for a directory where they want to create the network. Default value is */netsim-lab*.

```
admin@ncs(config)# netsim config netsim-dir /ios-netsim-lab
```

9.5.2 create-device

New networks can be created by using *create-network* or *create-device* actions. Both will create new network and load devices to NSO cdb.

```
admin@ncs# netsim create-device device-name R1 ned-id cisco-ios
```

9.5.3 add-device

Users can add one or multiple devices to existing networks. This action also creates devices and loads them to NSO cdb.

```
netsim add-device device-name [ R2 R3 R4 ] ned-id cisco-ios
```

9.5.4 start

Netsim devices can be started or stopped using start action. If user provides list of devices then only those will be started. If he skips the list all of netsim devices will be started.

```
admin@ncs(config)# netsim start device-name [ R1 R3 ]
```

9.5.5 load

Using netsim-tool load users can load existing netsim networks into new NSO running instance. Issuing this command will load devices in netsim network specified in *config->netsim-dir*.

```
admin@ncs(config)# netsim load
```

9.5.6 update-network

When changing NSO versions, existing netsim networks might stop working. With update-network action users can change schema files (.fxs) in the network with the ones from current NSO, which enables netsim devices to work with current version.

```
netsim update-network ncs-run /home/cisco/ncs-run
```

9.6 Creating and managing devices

```
ncs-netsim create-device cisco-ios PE11  
ncs-netsim add-device cisco-ios PE12
```

```

ncs-netsim add-device cisco-iosxr PE21
ncs-netsim add-device cisco-ios PE31
ncs-netsim add-device cisco-ios CE11
ncs-netsim add-device cisco-ios CE12
ncs-netsim add-device cisco-ios CE21
ncs-netsim add-device cisco-ios CE31

```

```

# Bulk Export for easy import
ncs-netsim ncs-xml-init > devices.xml

```

```

# And then Bulk Import to NCS
ncs_load -l -m devices.xml

```

```

admin@ncs# show devices brief
NAME  ADDRESS      DESCRIPTION  NED ID
-----
CE11   127.0.0.1    -           cisco-ios
CE12   127.0.0.1    -           cisco-ios
CE21   127.0.0.1    -           cisco-ios
CE31   127.0.0.1    -           cisco-ios
PE11   127.0.0.1    -           cisco-ios
PE12   127.0.0.1    -           cisco-ios
PE21   127.0.0.1    -           cisco-ios-xr
PE31   127.0.0.1    -           cisco-ios
R1     127.0.0.1    -           cisco-ios

```

You can actually connect to a Virtual Emulated router by the `ncs-netsim`:

```

root@cisco-virtual-machine:/opt/ncs/ncs-run/packages/l2vpn/templates# ssh
admin@127.0.0.1 -p 10022
admin@127.0.0.1's password:admin

admin connected from 127.0.0.1 using ssh on cisco-virtual-machine
PE11> en
PE11#

```

Or you can connect to virtual device like this:

```

$ ncs-netsim cli-i c1
c1> enable
c1# show running-config

```

Now connect to the your NCS CLI `ncs_cli -C -u admin` and add the device to the NCS

```

config
  devices device R1
    address 127.0.0.1 port 10022
    device-type cli ned-id cisco-ios protocol ssh
    authgroup default
    state admin-state unlocked
commit

```

Now connect to the device and fetch the configuration from them:

```

devices fetch-host-keys
devices sync-from
end

```

10 Netsim wrapper for NSO

Ncs-netsim is a great tool, but it lacks the following features which are developed as part of netsim-wrapper

- netsim-wrapper features
 - delete-devices <device-names>
 - create-network-from [yaml | json] <filename>
 - create-device-from [yaml | json] <filename>
 - create-network-template [yaml | json]
 - create-device-template [yaml | json]

Netsim-wrapper is a wrapper on top of ncs-netsim with added features. It's written in python and we opened the space to add more features to it.

10.1 Introduction

Ncs-netsim, It's a powerful tool to build a simulated network environment for Network Service Orchestrator (NSO) it's also called as NCS - NSO. In these network topologies we can test the network configurations based on the need as per the use case. netsim-wrapper, An open space to automate the ncs-netsim.

10.2 Pre-requisites

- ncs-netsim command must be recognised by the terminal.
- netsim-wrapper supports both trains of **python** 2.7+ and 3.1+, the OS should not matter.

10.3 Installation and Downloads

The best way to get netsim-wrapper is with setuptools or pip. If you already have setuptools, you can install as usual:

```
python -m pip install netsim-wrapper
pip install netsim-wrapper
```

Otherwise download it from PyPi, extract it and run the `setup.py` script

```
python setup.py install
```

If you're Interested in the source, you can always pull from the github repo:

- From github `git clone https://github.com/kirankotari/netsim-wrapper.git`

10.4 Features

10.4.1 Delete a device(s) from topology

existing device list:

```
x> ~/k/i/netsim-wrapper on master ◦ netsim-wrapper list
ncs-netsim list for /Users/kkotari/idea/netsim-wrapper/netsim

name=xr0 netconf=12022 snmp=11022 ipc=5010 cli=10022
dir=/Users/kkotari/idea/netsim-wrapper/netsim/xr/xr0
name=xr1 netconf=12023 snmp=11023 ipc=5011 cli=10023
dir=/Users/kkotari/idea/netsim-wrapper/netsim/xr/xr1
```

```
name=xr2 netconf=12024 snmp=11024 ipc=5012 cli=10024
dir=/Users/kkotari/idea/netsim-wrapper/netsim/xr/xr2
name=xr3 netconf=12025 snmp=11025 ipc=5013 cli=10025
dir=/Users/kkotari/idea/netsim-wrapper/netsim/xr/xr3
X> ~/k/i/netsim-wrapper on master ◦
```

deleting devices

```
X> ~/k/i/netsim-wrapper on master ◦ netsim-wrapper delete-devices xr1 xr3
[ INFO ] :: [ ncs-netsim ] :: deleting device: xr1
[ INFO ] :: [ ncs-netsim ] :: deleting device: xr3
X> ~/k/i/netsim-wrapper on master ◦
```

latest device list

```
X> ~/k/i/netsim-wrapper on master ◦ netsim-wrapper list
ncs-netsim list for /Users/kkotari/idea/netsim-wrapper/netsim

name=xr0 netconf=12022 snmp=11022 ipc=5010 cli=10022
dir=/Users/kkotari/idea/netsim-wrapper/netsim/xr/xr0
name=xr2 netconf=12024 snmp=11024 ipc=5012 cli=10024
dir=/Users/kkotari/idea/netsim-wrapper/netsim/xr/xr2
X> ~/k/i/netsim-wrapper on master ◦
```

10.4.2 Create Network/Device Template

Template to automate the Network/Device creation process.

```
X> ~/k/i/netsim-wrapper on master ◦ netsim-wrapper create-network-template [yaml
| json]
X> ~/k/i/netsim-wrapper on master ◦ netsim-wrapper create-device-template [yaml
| json]
```

which gives template.json/yaml file where you can update the files based on your need/requirement.

10.4.3 Create Network/Device From Template

We are using the templates which are updated based on your requirement:

```
X> ~/k/i/netsim-wrapper on master ◦ netsim-wrapper create-network-from [yaml |
json] <filename>
X> ~/k/i/netsim-wrapper on master ◦ netsim-wrapper create-device-from [yaml |
json] <filename>
```

10.4.4 How to choose the Templates and How they look (Template options)

These templates follows the same process of ncs-netsim, format is your choice

1. prefix based creation - netsim-wrapper create-network-template yaml or json <file>
2. name based creation - netsim-wrapper create-device-template yaml or json <file>

Note:- If you need combinations of network/device template, we suggest to create 2 template files and run the command for each type. You can find the example all supported templates under it's folder.

10.4.4.1 prefix-based template example

```
nso-packages-path: <path-to>/nso-local-lab/nso-run-5.2.1.2/packages
compile-neds: true
```

```

start-devices: true
add-to-nso: true
add-authgroup-to-nso: true
authgroup:
  type: system
device-mode:
  prefix-based:
    cisco-ios-cli-6.56:
      count: 2
      prefix: ios-56-
    cisco-ios-cli-6.55:
      count: 2
      prefix: ios-55-
load-day0-config: true
config-path: <path-to>/nso-local-lab/nso-run-5.2.1.2/preconfig
config-files:
- devices_ios_56.xml
- devices_ios_55.xml

```

10.4.4.2 name-based template example

```

nso-packages-path: <path-to>/nso-local-lab/nso-run-5.2.1.2/packages
compile-neds: true
start-devices: true
add-to-nso: true
add-authgroup-to-nso: true
authgroup:
  type: system
device-mode:
  name-based:
    cisco-ios-cli-6.56:
      - ios-56-name-100
      - ios-56-name-150
    cisco-ios-cli-6.55:
      - ios-55-name-200
      - ios-55-name-250
load-day0-config: true
config-path: <path-to>/nso-local-lab/nso-run-5.2.1.2/preconfig
config-files:
- devices_ios_56.xml
- devices_ios_55.xml

```

10.4.4.3 Template options in detail

```

nso-packages-path:
  info: nso package path

compile-neds:
  options: true or false
  info: which run _make clean all_ for each ned

start-devices:
  options: true or false
  info: starts devices using _ncs-netsim start_, it's intellegent enought to
start only stopped devices

add-to-nso:
  options: true or false
  info: adds day0 devices config to nso

add-authgroup-to-nso:
  options:

```

type: local or system or custom
path: authgroup config file path this option is only for custom
info: configuring authgroup

device-mode:

- prefix-based: ncs-netsim create-network/add-to-network
ned-name: ned name
options:
count: number of devices
prefix: prefix text for device names
- name-based: ncs-netsim create-device/add-device
ned-name: ned name
options: device names

load-day0-config:

options: true or false
info: to add day0 config if the value is true

config-path:

info: configuratin folder path

config-files:

info: loads each configuration file from given config-path

10.5 Help

❯ ~/k/i/netsim-wrapper on master ◦ netsim-wrapper --help

```
Usage netsim-wrapper [--dir <NetsimDir>]
    create-network-template [yaml | json]
    create-network-from [yaml | json] <fileName>
    create-network <NcsPackage> <NumDevices> <Prefix>
    create-device-template [yaml | json]
    create-device-from [yaml | json] <fileName>
    create-device <NcsPackage> <DeviceName>
    add-to-network <NcsPackage> <NumDevices> <Prefix>
    add-device <NcsPackage> <DeviceName>
    delete-devices <DeviceNames>
    delete-network
    [-a | --async] start [devname]
    [-a | --async] stop [devname]
    [-a | --async] reset [devname]
    [-a | --async] restart [devname]
    list
    is-alive [devname]
    status [devname]
    whichdir
    ncs-xml-init [devname]
    ncs-xml-init-remote <RemoteNodeName> [devname]
    [--force-generic]
    packages
    netconf-console devname [XpathFilter]
    [-w | --window] [cli | cli-c | cli-i] devname
    get-port devname [ipc | netconf | cli | snmp]
    -v | --version
    -h | --help
```

See manpage for ncs-netsim for more info. NetsimDir is optional and defaults to ./netsim, any netsim directory above in the path, or \$NETSIM_DIR if set.

10.6 FAQ

Question: Do I need to install ncs-netsim too?

Answer: Not really, ncs-netsim tool comes along with NSO. If you are working with NSO it's won't be a problem.

Question: Is python mandatory for netsim-wrapper?

Answer: Yes, the library is written in python and we wanted not to be dependend on NSO versions.

Question: Is netsim-wrapper backward compatable?

Answer: We recommend to use netsim-wrapper commands instead of ncs-netsim. However couple of commands are still backward compatable ie. `ncs-netsim list`, etc.

Question: I am seeing following error `./env.sh: line 12: export:`

``Fusion.app/Contents/Public:/Applications/Wireshark.app/Contents/MacOS': not a valid identifier`

Answer: We recommend to check your env path as recommended in following [link](#)

11 Creating Network Configurations with Jinja

<https://routebythescript.com/creating-network-configurations-with-jinja/>

by **SETH BEAUCHAMP**

What is jinja?

Jinja is a templating engine for python. Commonly it is used alongside Django or Flask, both of which provide a way to create websites using python. As network engineers, we often use and create templates for network configurations. Typically for us that consists of a text document with placeholder text for variable items such as IP addresses or VLAN IDs. So of course, jinja is a natural fit for a network engineer as well.

So why use jinja over our usual text file?

One thing jinja can provide is the ability create a single configuration template and use it to iterate over a list to create a block of code for each item. A simple example would be VLAN creation. Perhaps you are configuring a new switch and need to create several vlans. Would you want to copy/paste your code block multiple times and manually change your placeholder text to the correct vlan ID each time? With jinja, you can simply pass in a list of vlan numbers. Heres an example.

Note: These examples assume you have basic knowledge of python. Check [this post](#) for recommendations on learning basic python. Syntax is written for python 3.

```
#start by importing the Template function from jinja2
from jinja2 import Template
```

```
#Set up your jinja template
jtemplate = Template("vlan {{ vlan }}\n name VLAN{{ vlan }}_byJinja\n")
```

```
#Create the list of VLANs you want to generate configuration for
vlans = [10,20,30,40,50,60,70,80,90,100]
```



```
#Iterate over the list of vlans and print a useable configuration
for vlan in vlans:
    output = jtemplate.render(vlan=vlan)
    print(output)
```

Above is a full ready to go python script that will generate configuration necessary for creating vlans 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100. Running this code will give the following output:

```
vlan 10
  name VLAN10_byJinja
vlan 20
  name VLAN20_byJinja
vlan 30
  name VLAN30_byJinja
vlan 40
  name VLAN40_byJinja
vlan 50
  name VLAN50_byJinja
vlan 60
  name VLAN60_byJinja
vlan 70
  name VLAN70_byJinja
vlan 80
  name VLAN80_byJinja
vlan 90
  name VLAN90_byJinja
vlan 100
  name VLAN100_byJinja
```

We have our vlans with names ready to deploy to our switch. Pretty simple example, so lets break it down.

```
jtemplate = Template("vlan {{ vlan }}\n name VLAN{{ vlan }}_byJinja\n")
```

In this example, we first set up a jinja template by creating an object named jtemplate and assinging Jinja's Template() function to it. We must also pass in our actual template into this function. Jinja uses double curly brackets to identify a variable within a block of configuration. In the above example, you will see {{ vlan }} placed in our template as a placeholder for our VLAN ID. In fact, its used twice. Once to configure the vlan itself, and a second time to help give a name or description to the vlan. You will also notice "\n" in the template, this is used to create a new line to help with the formatting of our output and is not printed literally.

```
vlans = [10,20,30,40,50,60,70,80,90,100]
```

```
for vlan in vlans:
    output = jtemplate.render(vlan=vlan)
    print(output)
```

Next, we need a list of vlans and then some code to actually generate the configuration. You can create this a variety of ways, however for this example it's a simple list of vlan ids. We will iterate over this list of vlans using a simple for loop to generate a configuration for each vlan in our list. Within the for loop we create a variable named output to hold our configuration, which is created by using an attribute of our jtemplate object called render(). You need to pass in your variables using keyword/named arguments. The only variable we have in this example is vlan, so we will pass it in using output = jtemplate.render(vlan=vlan). In other words, for each iteration of the for loop we will be telling jinja that vlan is equal to the current item in

our list. Then you can print the results like I have done in my example or retrieve the data in any way you wish, such as saving it to a text file.

While this is a very simple example, you can imagine how quickly you can create a configuration for a huge list of vlans. If you have a contiguous list of VLAN IDs, you could simplify even further with something like “vans = range(2,1000)” to quickly create a list of VLANs 2 through 999.

The previous example is single file python script, which works just fine for a simple vlan creation. You may be wondering how a much larger configuration template would work. Trying to work a large configuration template into python code isn't very practical. Jinja allows you to use a separate file containing your template to be used by your main python code. Heres an example of a slightly more complex template using jinja.

```
interface {{ intf }}
description {{ intdscr }}
ip address {{ ip }} {{ mask }}
service-policy output {{ qospol }}

router bgp {{ bgpasn|default("655123") }}
neighbor {{ bgpnip }} remote-as {{ remasn }}

ip route 0.0.0.0 0.0.0.0 {{ bgpnip }}
```

For this, you simply start a new text document and save it. It is recommended to use the file extension “.j2” to clearly identify it as a jinja template. For example “wan.j2”. This looks very similar to something most network engineers already use, we can format it to look like our typical network configuration. You just need to use the jinja double curly bracket format in place of your variables. You can also set default values directly in the jinja template. Here I am using {{ bgpasn|default(“655123”) }}. If the user doesn't pass in a BGP ASN value, jinja will use the default of 655123.

```
with open("wan.j2") as file:
    jtemplate = Template(file.read())
```

Once again you need to set up your jinja template. This time, we are reading our jinja template from a file named “wan.j2” and storing it in a variable named jtemplate. Remember, unless you specify a full path for the file in open(), python will only look in the directory from which you are executing the code. So here, our wan.j2 file needs to be in the same directory where we have the python interpreter running or where our .py file is located if we are executing it that way.

```
output =
jtemplate.render(intf="Ethernet1/0",intdscr="WAN_byJinja",ip="10.1.1.1",
                 mask="255.255.255.252",
qospol="200MB_SHAPE",bgpasn="65456",
                 bgpnip="10.1.1.2",remasn="65499")
print(output)
```

Now we just need to create the output variable to hold our configuration. In this case we are just passing the values in directly as keyword/named arguments instead of iterating over a list. Then we can print output which gives us this:

```
interface Ethernet1/0
description WAN_byJinja
ip address 10.1.1.1 255.255.255.252
service-policy output 200MB_SHAPE

router bgp 65456
neighbor 10.1.1.2 remote-as 65499
```

```
ip route 0.0.0.0 0.0.0.0 10.1.1.2
```

Now to put it all in one place:

```
#Separate jinja template file named wan.j2

interface {{ intf }}
  description {{ intdscr }}
  ip address {{ ip }} {{ mask }}
  service-policy output {{ qospol }}

router bgp {{ bgpasn|default("655123") }}
  neighbor {{ bgpnip }} remote-as {{ remasn }}

ip route 0.0.0.0 0.0.0.0 {{ bgpnip }}
#start by importing the Template function from jinja2
from jinja2 import Template

#Set up the jinja template
with open("wan.j2") as file:
    jtemplate = Template(file.read())

#Create a variable along with some logic to generate the configuration
output =
jtemplate.render(intf="Ethernet1/0",intdscr="WAN_byJinja",ip="10.1.1.1",
                 mask="255.255.255.252",
qospol="200MB_SHAPE",bgpasn="65456",
                 bgpnip="10.1.1.2",remasn="65499")
print(output)
```

Since this post is getting pretty long, I believe I will cut it off here and post a follow up with some more advanced uses for jinja. While jinja its self is pretty simple to use, the examples here just scratch the surface on ways you can use python and jinja to simplify creation of configuration. To boil it down, jinja allows you to create templates in a very similar way to how you may already be doing it. The real power comes from the ability to create a template one time and use creative python code to create hundreds of configurations near instantly.

In these very simple examples, I only showed how to use basic python to feed information to the template. You can also pair this with something like REST, YAML, JSON, etc. to obtain your information. Imagine feeding in a YAML file containing WAN information for thousands of devices, you would be able to create a WAN configuration for each device within a matter of seconds.

With this basic knowledge of jinja, I hope this sparks your imagination on the possibilities on how to turn the mundane task of creating the same network configurations over and over again into something simple that only takes seconds.

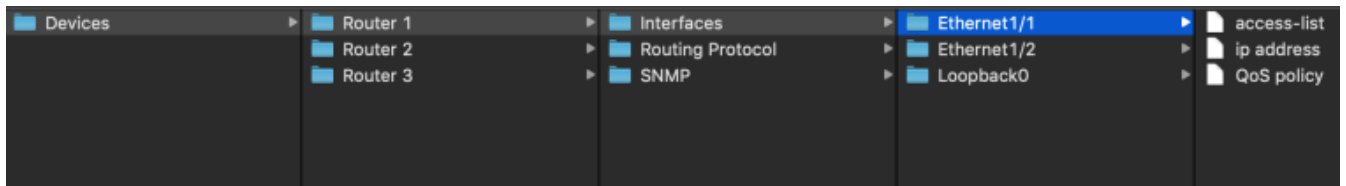
12 Using YAML and Jinja to Create Network Configurations

In a [previous post](#), I went over the basics of using Jinja to create templates for network configurations. This time I will be showing some examples of using YAML to store information about our network devices and using it to feed information to jinja.

What is YAML?

YAML is a data serialization language that is also very easy for humans to read. Often used to store data or information, such as settings for a program. Of course, it also happens to be quite good for storing network information, specifically configuration items for a device. You can think of it like a folder structure in an OS like windows or macOS. You may have a folder for documents, which contains another folder for photos which may contain another folder for vacation photos specifically. For us as network engineers this would translate to a container labeled “devices”, which has another container labeled “R1”, “R2”, “R3” and so on. Even further each of those containers may have yet another container for specific information about “interfaces”, “snmp”, “routing protocol” etc.

A visual aid using folders on my macbook might look like this:



So you can see how we start very broad at devices, then move into specific devices by name such as R1, then further into interfaces belonging to R1, then once again into a specific interface Ethernet1/1, in which we finally find the properties, attributes, configuration, or whatever you want to call it for Ethernet 1/1. Now here’s what this looks like in YAML:

```
devices:
  - name: "R1"
    interfaces:
      - name: "Ethernet1/1"
        state: "enabled"
        ip: "10.0.0.1"
        mask: "255.255.255.0"
        qos_policy: "SHAPE_200MB"
      - name: "Loopback0"
        state: "enabled"
        ip: "1.1.1.1"
        mask: "255.255.255.255"
    bgpasn: "65000"
    bgp_neighbors:
      - ip: "10.0.0.2"
        remote_as: "65001"
  - name: "R2"
    interfaces:
      - name: "Ethernet1/1"
        state: "enabled"
        ip: "10.0.0.2"
        mask: "255.255.255.0"
        qos_policy: "SHAPE_200MB"
    bgpasn: "65001"
    bgp_neighbors:
      - ip: "10.0.0.1"
        remote_as: "65000"
```

Still pretty easy to read isn’t it? Not only is it easy for us to read, but its also really easy for python to parse. If you’ve tinkered with automation, you have probably noticed it can be pretty difficult to parse CLI output from a show command in python, even though it’s really easy for us as humans to read. Having data structured in such a way that we can understand and use programmatically is quite useful.

In this example we have created an object to hold all of our devices by writing “devices:” with a colon. The next line leads with two spaces and a “-” to indicate we are beginning a new list which will represent a

single device with the name "R1". This list will then contain an object to hold this devices interfaces, which we indicate with "interfaces:", the name of the object following by just a colon. Then, we are creating another list within this object to describe "Ethernet1/1" and all of its configuration items, as well as a second list for another interface "Loopback 0". Its difficult to coherently describe nested objects with multiple layers, hence giving multiple examples in hopes one of them will stick. Formatting is pretty important here, as the indentation level indicates a sort of parent child relationship. So, the individual interfaces are children of object "interfaces", which is part of a list describing a particular device, which is a child of the overall object "devices".

Now, theres one more way I want to present YAML, and thats how it looks after python has consumed it. To use YAML in python you will need to install the PyYAML module using pip. Once thats done import the module, open the file containing your YAML, and use the `yaml.safe_load()` function to read it in like this:

```
import yaml
from pprint import pprint as p

with open("devices.yaml") as file:
    device_info = yaml.safe_load(file)

p(device_info)
```

I used prettyprint to get a better formatted output to display here, but its not required of course. Heres the output:

```
{'devices': [{'bgp_neighbors': [{'ip': '10.0.0.2', 'remote_as': '65001'}],
  'bgpasn': '65000',
  'interfaces': [{'ip': '10.0.0.1',
    'mask': '255.255.255.0',
    'name': 'Ethernet1/1',
    'state': 'enabled'},
    {'ip': '1.1.1.1',
    'mask': '255.255.255.255',
    'name': 'Loopback0',
    'state': 'enabled'}],
  'name': 'R1'},
  {'bgp_neighbors': [{'ip': '10.0.0.1', 'remote_as': '65000'}],
  'bgpasn': '65001',
  'interfaces': [{'ip': '10.0.0.2',
    'mask': '255.255.255.0',
    'name': 'Ethernet1/1',
    'qos_policy': '"SHAPE_200MB"',
    'state': 'enabled'}],
  'name': 'R2'}]}
```

If you are pretty familiar with python, you will probably now realize how using YAML provides python with good data to parse. Its your basic python data structure: lists and dictionaries. You can easily loop through any lists and use specific values from the dictionary key value pairs to plug in variables into a config. Also remember that some data types in python are unordered, so its not necessarily going to come out in the same way it went in. But that doesn't really matter, the data is still accessible all the same.

There are many ways to you could format your data with YAML, what I've written here is just one way. You can use it to describe a single device or many devices. There are even more tools within YAML available than what ive shown here, I encourage you to experiment with YAML on your own to find what works best for you.

Lets quickly go through an example of using YAML to feed a jinja template. Check out this jinja template below:

```

{{ device }}

{% for int in interfaces -%}
interface {{ int.name }}
  ip address {{ int.ip }} {{ int.mask }}
  service-policy output {{ int.qos_policy }}
  {% if int.state == "enabled" -%}
  no shutdown
  {% else -%}
  shutdown
  {% endif -%}
!
{% endfor -%}
router bgp {{ bgpasn }}
{% for neighbor in bgp_neighbors -%}
  neighbor {{ neighbor.ip }} remote-as {{ neighbor.remote_as }}
{% endfor -%}

```

This template uses a few new tricks I didn't show in my previous post. In Jinja, you can actually use some common python procedures such as for loops and if statements. These are indicated by opening with {% and ending with %}. You may also notice a "-" at the end of the for loops and if statements. This is to prevent that line from showing a blank line in your output and messing up the formatting. Jinja does require you indicate the end of your for loops and if statements, which python its self does not require.

The other thing thats a little different here is our variables. For example we are using "int.name". It will become a little more clear when I show the python code below, but this is because we are passing lists and dictionaries into jinja instead of plain string variables. In the first for loop, we are iterating over "interfaces" which is a list of dictionaries. We can then extract the values we need to complete our configuration by access the data stored in each key of the dictionary. So "int.ip" will access the ip address for the interface. In python code, this would look something like "print(int['ip'])".

So now, we have our data in a YAML file named "devices.yaml" and we have our jinja template in a file named "template.j2". Lets look at how to use these in python:

```

#import yaml and jinja
import yaml
from jinja2 import Template

#read your yaml file
with open("devices.yaml") as file:
    devices = yaml.safe_load(file)

#read your jinja template file
with open("template.j2") as file:
    template = Template(file.read())

#iterate over the devices described in your yaml file
#and use jinja to render your configuration
for device in devices["devices"]:
    print(template.render(device=device["name"], interfaces=device["interfaces"],
        bgpasn=device["bgpasn"], bgp_neighbors=device["bgp_neighbors"]))

```

The YAML object "devices" begins as a dictionary. When you iterate over it, you get a list. That list contains dictionaries. So when we pass our variables into jinja using template.render(), we need to be mindful of the data type we are passing in and what our jinja template is expecting. In our first iteration, our list contains a dictionary with they key "name". So we can pass the value of that key in directly. Same for bgpasn. For

interfaces, our jinja template is expecting an iterable object, so passing in devices["interfaces"] will give it a list of dictionaries to iterate over. The same is true for the bgp_neighbors variable. Here's the final output:

```
R1

interface Ethernet1/1
 ip address 10.0.0.1 255.255.255.0
 service-policy output SHAPE_200MB
 no shutdown
!
interface Loopback0
 ip address 1.1.1.1 255.255.255.255
 service-policy output
 no shutdown
!
router bgp 65000
 neighbor 10.0.0.2 remote-as 65001
```

```
R2

interface Ethernet1/1
 ip address 10.0.0.2 255.255.255.0
 service-policy output SHAPE_200MB
 no shutdown
!
router bgp 65001
 neighbor 10.0.0.1 remote-as 65000
```

YAML is one of the many ways we can store information about our network devices in such a way python can easily parse. You can write YAML directly yourself, or tie to a provisioning system or perhaps a web gui for users to input data while python takes the user input and creates a YAML file in the background. Here we used it to feed a jinja template, however you might even want to write YAML and convert it to JSON or XML for use directly with some devices.

As always, there's more to discover but this is a blog post and not a text book. So I hope this has been a useful demonstration to get you started with YAML.

13 Automating Your Network Operations, Part 1 – Ansible Basics

<https://blogs.cisco.com/developer/automating-network-operations-part1>

by **"Steven Carter"** (all the three parts are here, chapters 13, 14 and 15).

In this blog series we'll take you beyond the hype and dive more deeply into how and why to automate your network operations.

I've spent the last couple of years at Red Hat helping customers automate their networks with Ansible. If there is one thing that I've learned during that time, it is that network automation is not as easy as many would have you believe. That is not to say that tools like Ansible are not good tools for automation or that anyone is trying to sell you snake oil, but I believe that there is a fundamental impedance mismatch in translating the success Ansible has had with automating systems to automate networks. Part of this disconnect stems from a fundamental mis-understanding of the capabilities that Ansible provides. According to Red Hat, Ansible is a "common language to describe your infrastructure." In practice,

however, Ansible is more of a framework that brings an inventory of things together with a set of modules, plugins, and Jinja2 capabilities that perform operations on those things. The language, rendered in YAML or JSON, just passes key/value pairs between the modules, plugins, and Jinja2 capabilities. (Yes, that's a simple description of a complex tool, but one that is accurate to illustrate the point of this and subsequent blogs.) That is not to say that Ansible is not a powerful framework, but it has no native linguistic ability to describe a network. When I say an "inventory of things," it is because Ansible really does not care what that thing is. Because of its agentless approach, it can talk to many things: systems, network devices, clouds, lightbulbs, etc. This is a great capability and part of why Ansible is so popular, but Ansible truly does not know one thing from another. It has no innate prowess for automating networks. It is simply a tool for automating what an operator does task by task. You cannot "describe" what you want OSPF to look like on your network. You simply provide a bunch of key/value pairs that get passed to the devices on your network through modules in hopes of yielding the OSPF configuration that you want.

13.1 Configuring settings on an IOS device

To illustrate this, let's look at configuring two simple settings on an IOS device: hostname and NTP servers. Using Ansible parlance, we'll describe the desired end state of the hostname of a particular device. Hostname is a great use case because it is a scalar (i.e. a single value). To change the hostname, the Ansible `ios_config` module does a simple textual compare of the configuration. If 'hostname newname' is not present, it sends that line to the device. Since hostname is a scalar, the old hostname gets replaced by the desired hostname. A list of NTP servers, however, is more difficult. Say you've set the NTP server to 1.1.1.1 with:

```
- ios_config:
  lines:
    - ntp server 1.1.1.1
```

Now you want to change your NTP server to 2.2.2.2, so you do:

```
- ios_config:
  lines:
    - ntp server 2.2.2.2
```

Simple, right? But the problem is that you would end up with 2 NTP servers in the configuration:

```
ntp server 1.1.1.1
ntp server 2.2.2.2
```

This is because the Ansible `ios_config` module does not see `ntp server 2.2.2.2` present in the configuration, so it sends the line. Since `ntp server` is a list, however, it adds a new NTP server instead of replacing the existing one, giving you 2 NTP servers (one that you do not want). To end up with just 2.2.2.2 as your NTP server, you would have to know that 1.1.1.1 was already defined as an NTP server and explicitly remove it... exactly what an operator would do. This is also the case with ACLs, IP prefix-lists, and any other list in IOS. Ansible does not have a native way to describe the desired end state of something simple like NTP servers on a network device, much less something more complex like OSPF, QoS, or Multicast.

Does that mean that Ansible is not a great tool for network automation? No, but like any tool, it needs to be used for the right task and can only complete a complex task when used in concert with other tools. As a framework, it is not a complete solution.

The intent of this blog series is to go beyond the hype and simple demonstrations prevalent in network automation conversations today and to dive more deeply into how and why to automate your network operations. In the next installment, I'll talk about data models and why they are a critical piece of any automation framework.

Until then, please visit the DevNet [Networking Dev Center](#) to see the wide range of resources and learning opportunities that are available. And please drop me a comment on this blog if you have questions, or topics you'd like this series to cover.

14 Automating Your Network Operations, Part 2 – Data Models

14.1 Keeping your IT infrastructure operational

Before I get into data models, I want to take a slight diversion to incorporate some of the feedback that I received from the first blog. It was pointed out that my use of the `ios_config` module was “naïve.” I contend that it is more accurate to say that my use of Ansible in general was “naïve,” since this was a pretty straight forward use of the module. In any case, it was by design. Why? Well, if you are like the majority of the network operators that I've worked with over the years, you are not a programmer (or you are a “naïve” programmer). You spend 110% of your time busting your chops making sure that the network that underpins your company's IT infrastructure is operational. I certainly do not want to discourage network operators going to classes, workshops, etc. to learn new skills if they have the time and the motivation, but it should not be necessary to *consume* network automation. We can do better!

The NTP example that I used in the first blog was also chosen for specific reasons. First, NTP and AAA were the questions that I got most from customers on this issue. They were able to configure these things initially, but ran into problems when they moved to an operational posture. Second, it is representative of the “naïve” approach that you'll see in the examples and workshops that are being delivered in the Ansible network automation genre. These “naïve” examples and workshops perpetuate the first reason. Lastly, there is no way to use the `ios_config` module, the most used Ansible module for Cisco IOS, to fix this. Nor is the issue addressed by an NTP-specific Ansible network module. Yes, there might be some other module in some git repo written by someone that addresses your corner case, but do you want to use it? Is writing a module for every single corner case for network automation a scalable approach? I say no. Partially because it is intractable, partially because it is a support nightmare.

14.2 Where do you get help for the modules that underpin the automation?

Red Hat supports a specific set of modules (generally the ones they write), other vendors support their modules, other modules are completely unsupported. Where do you get help for the modules that underpin the automation that makes your network work? Incidentally, we'll cover a technique to augment the `ios_config` module with a parser to address this problem in a later blog, but it is not intuitively obvious to the casual observer.

I believe that we need to focus on a smaller number of more capable modules and couple that with a more sophisticated, or more realistic, approach to network automation. Subsequent blogs will focus on different aspects of this more realistic approach, but this one will start with the biggest: Data Models.

14.3 So, what is a data model?

I know what you are thinking: “Wait a minute! I didn't need data models for automating my servers!” Well, building a system is a well-defined procedure with relatively few permutations or interdependencies on other systems. Also, provisioning a system generally consists of configuring values like hostname, IPs, DNS, AAA, and packages. Each of these are key/value pairs (e.g. `nameservers = 8.8.8.8, 8.8.4.4`) that define the operation of that system and there are relatively few of them for a system.

This is not the case for a network element. If we take a standard 48-port Top-of-Rack Switch, each port could have a description, a state, a VLAN, an MTU, etc. A single ToR could have hundreds of key/value pairs that dictate its operation. Multiply that across hundreds or even thousands of switches, and the number of key/value pairs grows rapidly. Collectively, all of these key/value pairs make up the Source of Truth (SoT) of

your network and there can be a lot of them. In fact, automating networks is really more of a data management and manipulation problem than it is a programming problem.

So, what is a data model? Generally speaking, a data model is a structure in which the meaning of a key/value pair is defined by its relative position in that structure. As an example, let's start with the de-facto standard in the networking space: YANG. According to RFC 7950, "YANG is a data modeling language used to model configuration data, state data, Remote Procedure Calls, and notifications for network management protocols."

If we look at the way we setup a simple BGP peering on Cisco IOS and Juniper JunOS, we basically have a bunch of values accompanied by a bunch of words using a particular grammar that describe those values:

Cisco IOS

```
router bgp 65082
no synchronization
bgp log-neighbor-changes
neighbor 10.11.12.2 remote-as 65086
no auto-summary
```

Juniper JunOS

```
bgp {
  local-as 65082;
  group TST {
    peer-as 65086;
    neighbor 10.11.12.2;
  }
}
```

But the values, two ASNs and an IP address, are the only things that really matter and they are the same in each.

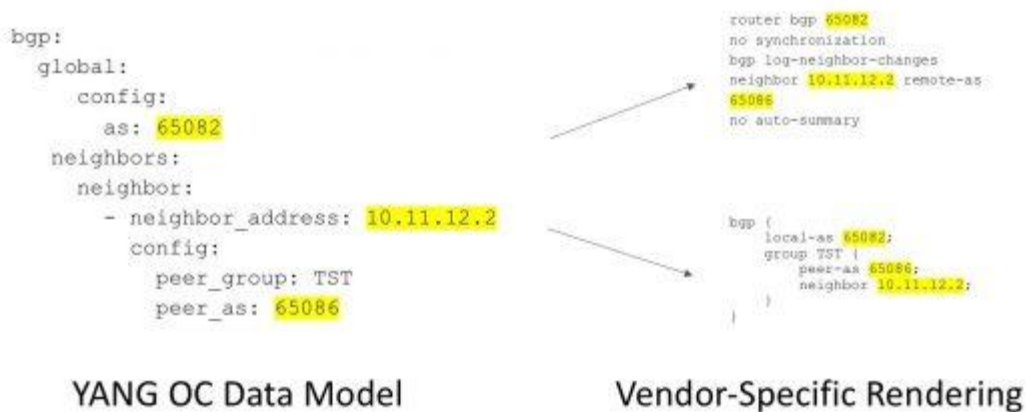
Cisco IOS

```
router bgp 65082
no synchronization
bgp log-neighbor-changes
neighbor 10.11.12.2 remote-as 65086
no auto-summary
```

Juniper JunOS

```
bgp {
  local-as 65082;
  group TST {
    peer-as 65086;
    neighbor 10.11.12.2;
  }
}
```

In fact, the switch hardware does not care about the words that describe those values since they get stored in a config DB anyway. The words are what the engineers gave to the humans to communicate the meaning of those values to the hardware. After all, we can't just specify 2 ASNs because we need to know which is the local and which is the remote. We could, however, communicate their meaning by order: e.g. <Local ASN>, <Peer ASN>, <Peer IP>. This is basically a small data model. Well, BGP gets A LOT more complicated, so we'll need a more capable data model. Here we have an example of the same data in the OpenConfig data model rendered in YAML:



The data in the model on the left contains the information needed to deliver either of the syntax specific versions... just add words. Yes, we still have words as tags in the model, but it normalizes those tags across vendors and gets rid of the grammar needed to specify how values relate to each other. We do not want to add words back if we can avoid it, so the next step is to encode all of this data into XML and shove it into the device via NETCONF.

NETCONF/YANG gives us programmability, but we still need automation since the two are not the same. This is where Ansible enters back into the story. In my opinion, it is the best of the open-source IaC tools for delivering data models to devices. I'll explain why and dig deeper into the power of data models in my next blog.

Want to learn more about data models before we start to use them in the next few blogs? Check out [Model Driven Programmability](#) at Cisco DevNet. As always, please drop me a comment on this blog if you have questions, or topics you'd like this series to cover.

15 Automating Your Network Operations, Part 3 – Data-Driven Ansible

How encoding and transport of the data-model gives you power and flexibility

In the [first blog](#), I attempted to make the point that the declarative approach of using function-specific modules in Ansible is not scalable. In the [second blog](#), I introduced data models into the conversation to help organize all of the key/value pairs that define your network. In this blog, I'll explain how the encoding and transport of the data-model give us quite a bit of power and flexibility when paired with Ansible. Data Models help organize the key/value pairs that define the network and YANG gives us a way to describe the meaning of the key/value pairs in the data structure, but we still need to communicate that information to the devices. This is where NETCONF comes in. NETCONF gives us several operational advantages over CLI, including:

- multiple configuration data stores (e.g. candidate, running, startup)
- configuration validation and testing
- differentiation between configuration and state data

For the purposes of this blog, however, the combination of YANG and NETCONF give us structure and determinism to enable programmability. As an example, let's consider the problem from the first blog: maintaining NTP server lists. This is what the ntp server configuration looks like in IOS CLI syntax:

```

line vty 2 4
  transport input ssh
!
ntp server 1.1.1.1
ntp server 2.2.2.2
!

```

I left the 'line vty' block to illustrate that the ntp configuration is just thrown in there at the root level. After all, the CLI was created for the humans and the humans are messy.

To illustrate the advantages of the model-driven method, let's use netconf-console to get and set the NTP servers on an IOS-XE device. First, let's see what our NTP servers are currently set to:

```

# netconf-console --host <device IP> --port 830 --user admin --password admin --
db running --get-config --xpath /native/ntp

```

```

<data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
    <ntp>
      <server xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-ntp">
        <server-list>
          <ip-address>1.1.1.1</ip-address>
        </server-list>
        <server-list>
          <ip-address>2.2.2.2</ip-address>
        </server-list>
      </server>
    </ntp>
  </native>
</data>

```

While this is a wordier rendering of the same configuration, it is deterministic. All of the ntp servers and their associated configuration are organized into one section of the tree. We can deal with it as a separate entity. Also, note that we were able to ask the device for just the ntp configuration information. No parsing required.

Now let's change our NTP servers. First, we take the previous output, change the IP address of the second NTP server, and specify that this operation should replace the server section. All of this goes into a file named ntp.xml:

```

<native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
  <ntp>
    <server xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-ntp"
operation='replace'>
      <server-list>
        <ip-address>1.1.1.1</ip-address>
      </server-list>
      <server-list>
        <ip-address>3.3.3.3</ip-address>
      </server-list>
    </server>
  </ntp>
</native>

```

Next, we push the XML payload to the device:

```

# netconf-console --host <device IP> --port 830 --user admin --password admin --
db running --edit-config ntp.xml

```

```
<ok xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"/>
```

The device accepted the change, so let's look at the result:

```
# netconf-console --host <device IP> --port 830 --user admin --password admin --
db running --get-config --xpath /native/ntp
<data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
    <ntp>
      <server xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-ntp">
        <server-list>
          <ip-address>1.1.1.1</ip-address>
        </server-list>
        <server-list>
          <ip-address>3.3.3.3</ip-address>
        </server-list>
      </server>
    </ntp>
  </native>
</data>
```

Huzzah! We've taken a change that normally takes less than a minute to do by hand and turned it into a 10-minute excursion. Welcome to programmability. You see, programmability by itself is useless to the humans. It's great that we can use netconf-console or gin up some python code to push and pull data to a single device, but it provides us no operational value. That is why this blog series is entitled "Automating Your Network Operations" and not "How to Make your Network Operations 10 Times Slower with Programmability".

This is where Ansible comes back in. Ansible is an automation tool. There are many automation tools that we could use, but Ansible is my preferred tool because it is more simple to use than other tools. Now simplicity is a double-edged sword. To achieve this simplicity, Ansible had to make compromises over more capable languages. Nested loops, for example, are a real pain in Ansible. That is why the model-driven approach is best for Ansible. It simplifies the use case such that it fits nicely within Ansible's capabilities. Let's look at how we'd do this same task using the netconf_rpc module in Ansible.

```
- hosts: routers
  connection: netconf
  gather_facts: no
  tasks:
    - netconf_rpc:
        rpc: edit-config
        content: |
          <target>
            <running/>
          </target>
          <config>
            <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
              <ntp>
                <server xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-ntp"
operation='replace'>
                  <server-list>
                    <ip-address>1.1.1.1</ip-address>
                  </server-list>
                  <server-list>
                    <ip-address>3.3.3.3</ip-address>
                  </server-list>
                </server>
              </native>
            </config>
          </target>
        
```

```

    </ntp>
  </native>
</config>

```

Great! We are using Ansible, so we are automating, right? We are one step closer. Ansible's inventory capabilities provide the mechanism to run this playbook over multiple devices. However, this playbook is neither portable nor reusable. We really want to separate out the key/value pairs embedded in the playbook from the tasks. We'll go deeper into Ansible's inventory capabilities in the next blog, but let's start by putting the NTP servers into a simple data structure:

```

ntp_servers:
- 1.1.1.1
- 3.3.3.3

```

To use this data structure, we create a Jinja2 template to create the XML payload for the NETCONF call and put it into a file named ntp.j2:

```

<target>
  <running/>
</target>
<config>
  <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
    <ntp>
      <server xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-ntp"
operation='replace'>
{% for server in ntp_servers %}
      <server-list>
        <ip-address>{{ server }}</ip-address>
      </server-list>
{% endfor %}
    </server>
  </ntp>
</native>
</config>

```

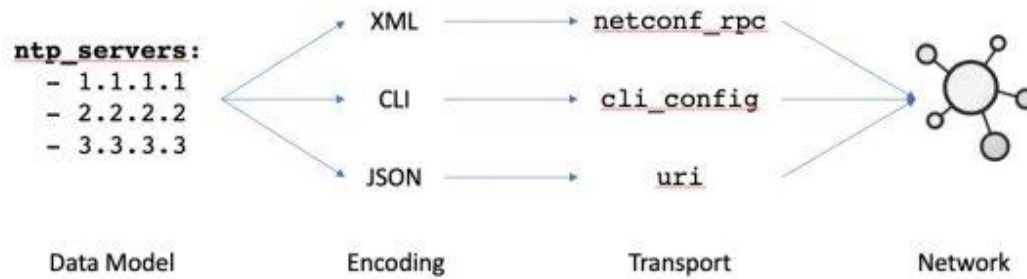
This template statically specifies the XML from above but dynamically adds the servers from the ntp_servers list, allowing code re-use by simply changing the data instead of the playbook. Then we use the lookup plug-in to process the template before we pass it into the netconf_rpc module:

```

- hosts: routers
  connection: netconf
  gather_facts: no
  tasks:
    - netconf_rpc:
      rpc: edit-config
      content: "{{ lookup('template', 'ntp-netconf.j2') }}"

```

The addition of Jinja2 gives us a powerful templating language to dynamically create XML payloads for NETCONF. It also adds more programmatic tools (like nested loops) for processing our data models to reduce the consumption of adult beverages needed to write a given playbook. Essentially, we take the model, encode it into an XML payload with a Jinja2 template, then send it via NETCONF with the netconf_rpc module. We can use this same technique, changing the encoding and the transport, to deliver our data model in a variety of ways to support nearly any device:



As an example of how we can change the encoding and transport to deliver the same data model in a different way, here is a Jinja2 template that encodes it into IOS-XE CLI:

```
{% for server in ntp_servers %}
ntp server {{ server }}
{% endfor %}
```

Then we use the cli_command module to deliver the payload via SSH:

```
- hosts: routers
  connection: network_cli
  gather_facts: no
  tasks:
    - cli_config:
        config: "{{ lookup('template', 'ntp-cli.j2') }}"
```

This is the framework that we'll use in this blog series for taking models and delivering them to devices. NTP is a simple example, but this framework will work with any deployment, as we'll see as this blog series progresses. In the next blog, I'll dig into the specifics of how we store our inventory and data for Ansible to reference and how to make our playbooks reusable, portable, and testable with Ansible Roles. In the meantime, I encourage you to check out more [NETCONF](#) training at [Cisco DevNet](#) and test NETCONF with your devices using [netconf-console](#).