NSO 4.4.2.3 Getting Started Guide

First Published: May 17, 2010

Last Modified: September 1, 2017

Americas Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 527-0883

CONTENTS

CHAPTER 1

NSO Introduction

Cisco Network Service Orchestrator enabled by Tail-f (NSO) version 4.4.2.3 is an evolution of the Tail-f Network Control System (NCS). Tail-f was acquired by Cisco in 2014. The product has been enhanced and forms the base for Cisco NSO. Note that the terms 'ncs' and 'tail-f' are used extensively in file names, command-line command names, YANG models, application programming interfaces (API), etc. Throughout this document we will use NSO to mean the product, which consists of a number of modules and executable components. These executable components will be referred to by their command line name, e.g. **ncs**, **ncs-netsim**, **ncs_cli**, etc. **ncs** is also used to refer to the executable, the running daemon.

Purpose of the Document

The purpose of this User Guide is two-fold:

- To get an understanding of NSO as a service orchestration and network configuration tool.
- To be used as a tool to get started using NSO.

This document only covers the most essential parts of using NSO to manage a network. Features are only covered at an introductory level.

NSO Overview

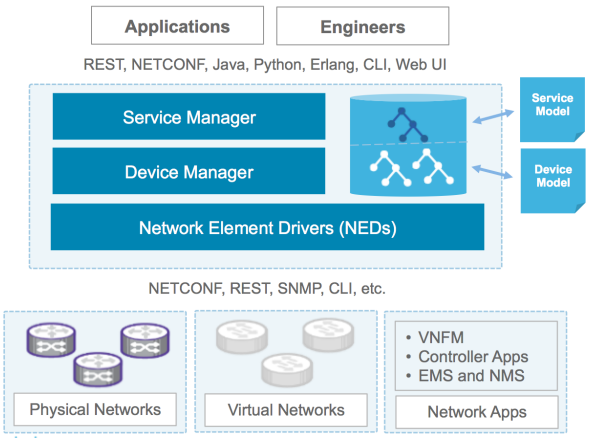
Creating and configuring network services is a complex task that often requires multiple integrated configuration changes to every device in the service chain. Additionally changes need to be made concurrently across all devices with service implementation being either completely successful or completely removed from the network. All configurations need to be maintained completely synchronized and up to date. NSO solves these problems by acting as interface between the configurators (network operators and automated systems) and the underlying devices in the network.

NSO does this by providing the following key functions:

1. Representation of the services.
2. Multi-vendor device configuration modification in the native language of the network devices.
3. Configuration Database (CDB) with current synchronized configurations for all devices and services in the network domain.

4 Northbound interfaces that can be accessed via WebUI or with automated systems using REST, Python, NETCONF, Java or other tools.

Figure 1. NSO



Network engineers use NSO as a central point of management for the entire network, using a "network CLI" or a web-based user interface. While this guide will illustrate the use cases with the CLI it is important to realize that any northbound interface can be used to achieve the same functionality.

All devices and services in the network can be accessed and configured using this CLI, making it a powerful tool for network engineers. It is also easy to define roles limiting the authority of engineers to the devices under their control. Policies and integrity constraints can also be defined making sure the configuration adhere to site standards.

The typical work flow when using the network CLI in NSO is as follows:

1 All changes are initially made to a (logical) copy of the NSO database of configurations.
2 Changes are validated against network policies and integrity constraints using the "validate" command. Changes can be viewed and verified by the network operator prior to committing them.
3 The changes are committed, meaning that the changes are copied to the NSO database and pushed out to the network. Changes that violate integrity constraints or network policies will not be committed. The changes to the devices are done in a holistic distributed atomic transaction, across all devices in parallell.
4 Changes either succeed and remain committed or fail and are rolled back as a whole returning the entire network to the uncommitted state.

Main Features

The main features of NSO are:

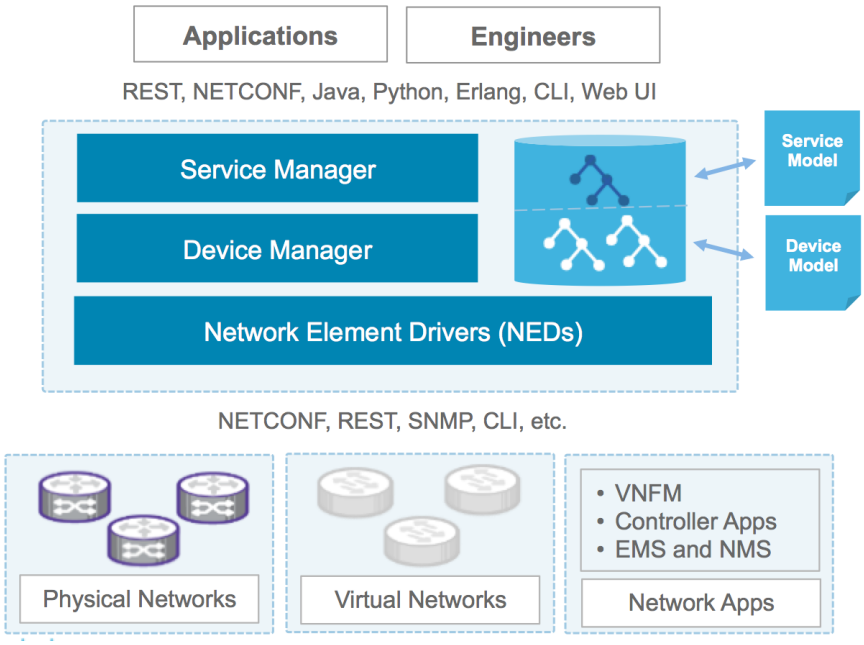
- Intuitive to the methods currently used for configuring network services using either Web UI or network CLI.
- Device changes are whole and either completely succeed or are completed discarded, no partial configurations are left.
- Fine-grained deep control of all elements in the service.
- Centralized configuration database that is synchronized with the network. Reconciliation can done in any direction: to network, from network.

- Rich set of user interfaces (WebUI, CLI) and scripting interfaces (REST, Python, NETCONF, JavaScript).

Architecture Overview

This section provides a broad overview of the NSO architecture and functionality.

Figure 2. NSO Architecture



NSO has two main layers: the *Device Manager* and the *Service Manager*. They serve different purposes but are tightly integrated into one transactional engine and database.

The purpose of the Device Manager is to manage device configurations in a transactional manner. It supports features like fine-grained configuration commands, bidirectional device configuration synchronization, device groups and templates, and compliance reporting.

The Service Manager makes it possible for an operator to manage high-level aspects of the network that are not supported by the devices directly, or is supported in a cumbersome way. With the appropriate service definition running in the Service Manager, an operator could for example configure the VLANs that should exist in the network in a single place, and the Service Manager compute the specific configuration changes required for each device in the network and push them out. This covers the whole life-cycle for a service: creation, modification and deletion. NSO has an intelligent easy to use mapping layer so that network engineers can define how a service should be deployed in the network.

NSO uses a dedicated built-in storage Configuration Database (CDB) for all configuration data. NSO keeps the CDB in sync with the real network device configurations. Audit, to ensure configuration consistency, and reconciliation, to synchronize configuration with the devices, functions are supported. It also maintains the runtime relationships between service instances and the corresponding device configurations.

NSO comes with Network Element Drivers, NEDs, that communicates with the devices. NEDs are not closed hard-coded adapters. Rather, the device interface is modeled in a data-model using YANG (RFC

6020). NSO can render the underlying commands from this model, including for example a Cisco IOS CLI. This means that the NEDs can easily be updated to support new commands just by adding these to the data-models. NSO in combination with the NEDs will apply device configuration changes as atomic change sets.

NSO *netsim* is used to simulate devices. Netsim can simulate management interfaces like Cisco CLI and NETCONF. It does not simulate the network behavior.

NSO supports a rich variety of APIs and User Interfaces. Independent of the device types NSO will render a northbound Cisco XR style network wide CLI. This will support all devices and services. The same is true for the NSO WebUI.

Scripts can be written using Python, REST, and NSO CLI or other tools to automate network activities.

CHAPTER 2

Running The NSO Examples

Overview

Throughout this Guide we will refer to examples included with the product. If you have a NSO system installed it is recommended to run the examples.

The procedure in general is:

Step 1 Make sure you **source** the `ncsrc` file in the NSO installation directory.
Step 2 **cd** to the example home directory.
Step 3 Prepare the simulated network using the **netsim** tools.
Step 4 Prepare the NSO example configuration.
Step 5 Start **netsim**.
Step 6 Start **ncs**.
Step 7 Run the scenario
Step 8 Stop **netsim**.
Step 9 Stop **ncs**.

For detailed instructions see below.

Instructions

Most of this User Guide will refer to some of the examples in `$NCS_DIR/examples.ncs`. Read and run the README files in parallel with this user guide.

Make sure NSO is installed according to the instructions in "Installing NSO". Source the `ncsrc` file in the NSO installation directory. For example:

```
$ source /opt/ncs/ncsrc
```

Go to the example:

```
$ cd $NCS_DIR/examples.ncs/getting-started/using-ncs/1-simulated-cisco-ios
```

Then follow the instructions in this user guide and the README that is located in the root of all examples.

Every example directory is a complete NSO *run-time* directory. The README file and the detailed instructions later in this guide show how to generate a simulated network and NSO configuration for running the specific examples. Basically, the following steps are done:

Step 1 Create a simulated network using the **ncs-netsim --create-network**. For example:

```
$ ncs-netsim create-network $NCS_DIR/packages/neds/cisco-ios 3 c
```

This creates 3 Cisco IOS devices called `c0`, `c1`, and `c2`.

Step 2 Create NSO run-time environment. For example:

```
$ ncs-setup --netsim-dir ./netsim --dest .
```

This does the following:

1. **--netsim-dir**: Reads netsim data (list of devices) from the `./netsim` directory. This was created by **ncs-netsim create-network** command.
2. **--dest**: creates local directories for logs, database files, and the NSO configuration file to the current directory (note that '.' refers to current directory).
3. Start netsim:

   ```
   $ ncs-netsim start
   ```
4. Start NSO

   ```
   $ ncs
   ```

It is important to make sure that you stop **ncs** and **ncs-netsim** when moving between examples.

```
$ cd $NCS_DIR/examples.ncs/getting-started/1-simulated-cisco-ios
$ ncs-netsim start
$ ncs
$ ncs-netsim stop
$ ncs --stop

$ cd $NCS_DIR/examples.ncs/data-center-qinq
$ ncs-netsim start
$ ncs
$ ncs-netsim stop
$ ncs --stop
```

Common mistakes

The three most common mistakes are:

1. You have not sourced the `ncsrc` file:

   ```
   $ ncs
   -bash: ncs: command not found
   ```
2. You are starting NSO from a directory, which is not a NSO runtime directory.

   ```
   $ ncs
   Bad configuration: /etc/ncs/ncs.conf:0: "./state/packages-in-use: \
   ```

```
   Failed to create symlink: no such file or directory"
Daemon died status=21
```

What happens above is that NSO did not find a `ncs.conf` in `./` so it uses the default in `/etc/ncs/ncs.conf`. That `ncs.conf` says there shall be directories at `./` such as `./state` which is not true.

Make sure you **cd** to the "root" of the example. Check that there is a `ncs.conf` file, and a `cdb-dir`.

3 You already have another NSO instance running (or same with netsim):

```
$ ncs
Cannot bind to internal socket 127.0.0.1:4569 : address already in use
Daemon died status=20
$ ncs-netsim start
DEVICE c0 Cannot bind to internal socket 127.0.0.1:5010 : \
  address already in use
Daemon died status=20
FAIL
```

In order to resolve the above, just stop the running NSO and netsim. (Note, again, remember that it does not matter where you started the running NSO and netsim, there is no need to cd back to the other example before stopping.)

Another problem that users run into sometimes is where the netsim devices are not loaded into NSO. This can happen if the order of commands are not followed. To resolve, remove the database files in the `ncs_cdb` directory (keep any files with .xml extension). In this way NSO will load the XML initialization files.

```
$ ncs --stop
$ cd ncs-cdb/
$ ls
A.cdb
C.cdb
O.cdb
S.cdb
netsim_devices_init.xml
$ rm *.cdb
$ ncs
```

CHAPTER 3

NSO Basics

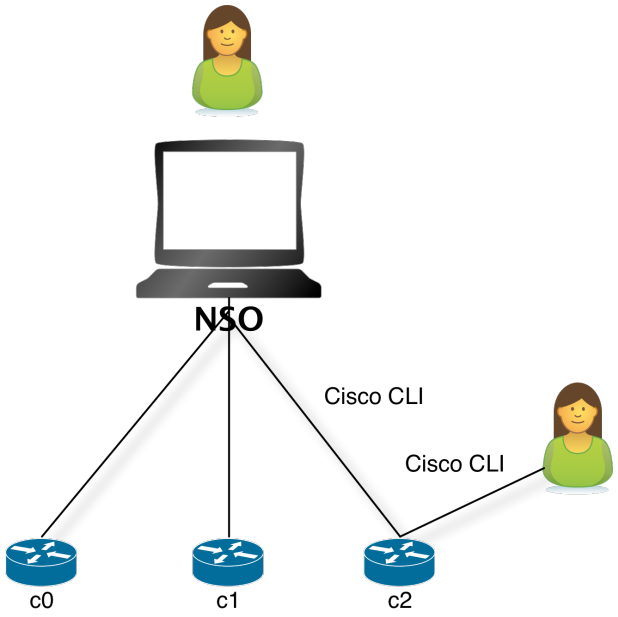
Getting started

The purpose of this section is to get started with NSO, learn the basic operational scenarios and get acquainted with the most common CLI commands.

Make sure you have installed NSO and that you have sourced the `ncsrc` file in `$NCS_DIR`. This sets up paths and environment variables in order to run NSO. So this must be done all times before running NSO, so it is recommended to put that in your profile.

We will use the NSO network simulator to simulate 3 Cisco IOS routers. NSO will talk Cisco CLI to those devices. You will use the NSO CLI and Web UI to perform the tasks. Sometimes you will use the native Cisco device CLI to inspect configuration or do out of band changes.

Figure 3. The first example



Note that both the NSO software (ncs)and the simulated network devices run on your local machine.

Starting the simulator

Go to `examples.ncs/getting-started/using-ncs/1-simulated-cisco-ios`.

Most of this section follows the procedure in the `README` file so it is useful to have that open as well. First of all we will generate a network simulator with 3 Cisco devices. They will be called `c0`, `c1`, and `c2`. Perform the following command:

```
$ ncs-netsim create-network $NCS_DIR/packages/neds/cisco-ios 3 c
```

This creates three simulated devices all running Cisco IOS and the will be named c0, c1, c2. Start the simulator:

```
$ ncs-netsim start
DEVICE c0 OK STARTED
DEVICE c1 OK STARTED
DEVICE c2 OK STARTED
```

Run the CLI towards one of the simulated devices

```
$ ncs-netsim cli-i c1
admin connected from 127.0.0.1 using console *

c1> enable
c1# show running-config
class-map m
match mpls experimental topmost 1
match packet length max 255
match packet length min 2
match qos-group 1
!
...
c1# exit
```

This shows that the device has some initial configurations.

Starting NSO and reading device configuration

The previous step started the simulated Cisco devices. It is now time to start NSO. The first action is to prepare directories needed for NSO to run and populate NSO with information of the simulated devices. This is all done with the **ncs-setup** command. Make sure you are in the `examples.ncs/getting-started/using-ncs/1-simulated-cisco-ios` directory. (Again ignore the details for time being).

```
$ ncs-setup --netsim-dir ./netsim --dest .
```

Note the "." at the end of the command referring to current directory. What the command does is to create directories needed for NSO in the current directory and populates NSO with devices that are running in netsim. We call this the "run-time" directory.

Start NSO;

```
$ ncs
```

Start the NSO CLI as user "admin" with a Cisco XR style CLI:

```
$ ncs_cli -C -u admin
```

NSO also supports a J-style CLI, that is started by using a -J modification to the command like this:

```
$ ncs_cli -J -u admin
```

Throughout this user guide we will show the commands in Cisco XR style.

At this point NSO only knows the address, port, and authentication information of the devices. This management information was loaded to NSO by the setup utility. It also tells NSO how to communicate with the devices by using NETCONF, SNMP, Cisco IOS CLI etc. Although at this point, the actual configuration of the individual devices is un-known.

```
admin@ncs# show running-config devices device
devices device c0
 address   127.0.0.1
 port      10022
...
 authgroup default
 device-type cli ned-id cisco-ios
 state admin-state unlocked
 config
  no ios:service pad
  no ios:ip domain-lookup
  no ios:ip http secure-server
  ios:ip source-route
 !
!  ...
```

Let us analyze the above CLI command. First of all, when you start the NSO CLI it starts in operational mode, so in order to show configuration data you have to explicitly say **show running-config**.

NSO manages a list of devices, each device is reached by the path **devices device "name"** . You can use standard tab completion in the CLI to learn this.

The `address` and `port` fields tells NSO where to connect to the device. For now they all live in local host with different ports. The `device-type` structure tells NSO it is a CLI device and the specific CLI is supported by the Network Element Driver (NED) `cisco-ios`. A more detailed explanation on how to configure the device-type structure and how to chose NEDs will be addressed later in this guide.

So now NSO can try to connect to the devices:

```
admin@ncs# devices connect
connect-result {
    device c0
    result true
    info (admin) Connected to c0 - 127.0.0.1:10022
}
connect-result {
    device c1
    result true
    info (admin) Connected to c1 - 127.0.0.1:10023
}
connect-result {
    device c2
    result true
    info (admin) Connected to c2 - 127.0.0.1:10024
}....
```

NSO does not need to have the connections "active" continuously, instead NSO will establish a connection when needed and connections are pooled to conserve resources. At this time NSO can read the configurations from the devices and populate the configuration database, CDB.

The following command will synchronize the configurations of the devices with the CDB and respond with "true" if successful:

```
admin@ncs# devices sync-from
sync-result {
    device c0
    result true
}....
```

The NSO data-store, CDB, will store configuration for every device at the path `devices device "name" config`, everything after this path is configuration in the device. NSO keeps this synchronized. The synchronization is managed with the following principles:

1. At initialization NSO can discover the configuration as shown above.
2. The modus operandi when using NSO to perform configuration changes is that the network engineer uses NSO (CLI, WebUI, REST,...) to modify the representation in NSO CDB. The changes are committed to the network as a transaction that includes the actual devices. Only if all changes happens on the actual devices will it be committed to the NSO data-store. The transaction also covers the devices so if any of the devices participating in the transaction fails, NSO will roll-back the configuration changes on the all modified devices. This works even in the case of devices that do not natively support roll-back like Cisco IOS CLI.
3. NSO can detect out of band changes and reconcile them by either updating the CDB or modifying the configuration on the devices to reflect the currently stored configuration.

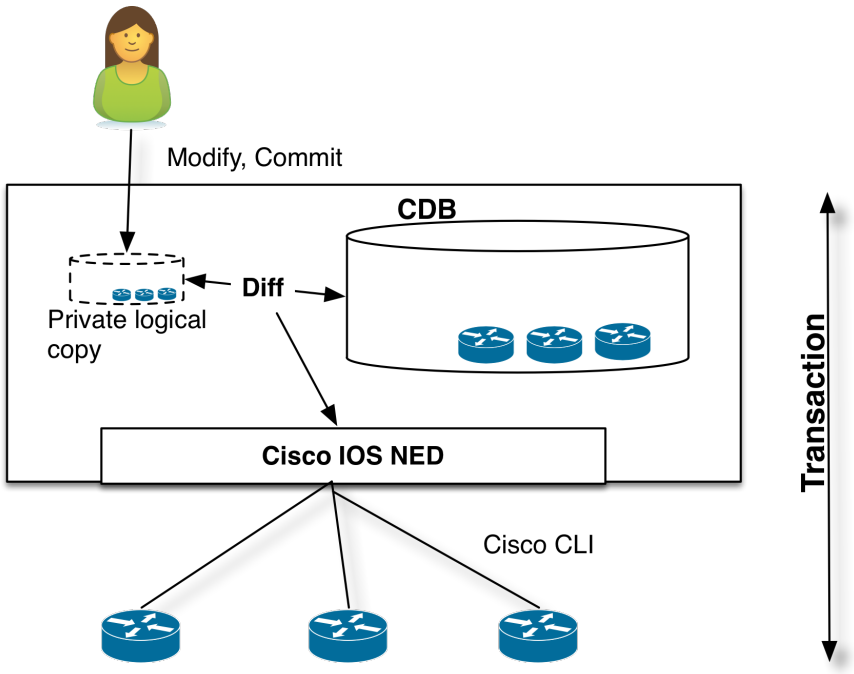
NSO only needs to be synchronized with the devices in the event of a change being made outside of NSO. Changes made using NSO will reflected in both the CDB and the devices. The following actions *do not* need to be taken:

1. Perform configuration change via NSO
2. Perform sync-from action

The above incorrect (or not necessary) sequence stems from the assumption that the NSO CLI speaks directly to the devices. This is not the case, the northbound interfaces in NSO modifies the configuration in the NSO data-store, NSO calculates a minimum difference between current configuration and the new

configuration, gives only the changes to the configuration to the NEDS that runs the commands to the devices. All this as one single change-set.

Figure 4. Device transaction



View the configuration of the "c0" device using the command:

```
admin@ncs# show running-config devices device c0 config
devices device c0
 config
  no ios:service pad
  ios:ip vrf my-forward
   bgp next-hop Loopback 1
  !
...
```

Or show a particular piece of configuration from several devices

```
admin@ncs# show running-config devices device c0..2 config ios:router
devices device c0
 config
  ios:router bgp 64512
   aggregate-address 10.10.10.1 255.255.255.251
   neighbor 1.2.3.4 remote-as 1
   neighbor 1.2.3.4 ebgp-multihop 3
   neighbor 2.3.4.5 remote-as 1
   neighbor 2.3.4.5 activate
   neighbor 2.3.4.5 capability orf prefix-list both
   neighbor 2.3.4.5 weight 300
  !
 !
!
devices device c1
```

```
 config
  ios:router bgp 64512
...
```

Or show a particular piece of configuration from all devices

```
admin@ncs# show running-config devices device config ios:router
```

The CLI can pipe commands, try TAB after "|" to see various pipe targets.

```
admin@ncs# show running-config devices device config ios:router \
                    | display xml | save router.xml
```

The above command shows the router config of all devices as xml and then saves it to a file `router.xml`

Writing device configuration

In order to change the configuration, enter configure mode:

```
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)#
```

Change or add some configuration across the devices, for example:

```
admin@ncs(config)# devices device c0..2 config ios:router bgp 64512
            neighbor 10.10.10.0 remote-as 64502
admin@ncs(config-router)#
```

It is important to understand how NSO applies configuration changes to the network. At this point the changes are local to NSO, no configurations have been sent to the devices yet. Since the NSO Configuration Database, CDB, is in sync with the network, NSO can calculate the minimum diff to apply the changes to the network. The command below compares the ongoing changes with the running database:

```
admin@ncs(config-router)# top
admin@ncs(config)# show configuration
devices device c0
 config
  ios:router bgp 64512
   neighbor 10.10.10.0 remote-as 64502
...
```

It is possible to dry-run the changes in order to see the native Cisco CLI output (in this case almost the same as above):

```
admin@ncs(config)# commit dry-run outformat native
native {
    device {
        name c0
        data router bgp 64512
              neighbor 10.10.10.0 remote-as 64502
             !
...
```

The changes can be committed to the devices and the NSO CDB simultaneously with a single commit. In the commit command below, we pipe to details to understand the actions being taken.

```
admin@ncs% commit | details
```

Changes are committed to the devices and the NSO database as one transaction. If any of the device configurations fail, all changes will be rolled back and the devices will be left in the state that they were

in prior to the commit and the NSO CDB will not be updated. There are numerous options to the commit command which will affect the behaviour of the atomic transactions.

```
admin@ncs(config)# commit TAB
Possible completions:
  and-quit               Exit configuration mode
  check                  Validate configuration
  comment                Add a commit comment
  commit-queue           Commit through commit queue
  label                  Add a commit label
  no-confirm             No confirm
  no-networking          Send nothing to the devices
  no-out-of-sync-check   Commit even if out of sync
  no-overwrite           Do not overwrite modified data on the device
  no-revision-drop       Fail if device has too old data model
  save-running           Save running to file
  ---
  dry-run                Show the diff but do not perform commit
```

As seen by the details output, NSO stores a roll-back file for every commit so that the whole transaction can be rolled back manually. The following is an example of a roll-back file:

```
admin@ncs(config)# do file show logs/rollback1000
Possible completions:
     rollback10001  rollback10002  rollback10003  \
                          rollback10004  rollback10005
admin@ncs(config)# do file show logs/rollback10005
# Created by: admin
# Date: 2014-09-03 14:35:10
# Via: cli
# Type: delta
# Label:
# Comment:
# No: 10005

ncs:devices {
    ncs:device c0 {
        ncs:config {
            ios:router {
                ios:bgp 64512 {
                    delete:
                    ios:neighbor 10.10.10.0;
                }
            }
        }
    }
```

(Viewing files as an operational command, prefixing a command in configuration mode with **do** executes in operational mode.) To perform a manual roll-back first load the rollback file:

```
admin@ncs(config)# rollback configuration 10005
```

Rollback "configuration" restores to that saved configuration, rollback "selective" just rollbacks the delta in that specific rollback file. Show the differences:

```
admin@ncs(config)# show configuration
devices device c0
 config
  ios:router bgp 64512
   no neighbor 10.10.10.0 remote-as 64502
  !
 !
```

```
!
devices device c1
 config
  ios:router bgp 64512
   no neighbor 10.10.10.0 remote-as 64502
  !
 !
!
devices device c2
 config
  ios:router bgp 64512
   no neighbor 10.10.10.0 remote-as 64502
  !
 !
!
```

Commit the rollback:

```
admin@ncs(config)# commit
Commit complete.
```

A trace log can be created to see what is going on between NSO and the device CLI enable trace. Use the following command to enable trace:

```
admin@ncs(config)# devices global-settings trace raw trace-dir logs
admin@ncs(config)# commit
Commit complete.
admin@ncs(config)# devices disconnect
```

Note: Trace settings only take effect for new connections so is important to disconnect the current connections. Make a change to for example c0:

```
admin@ncs(config)# devices device c0 config ios:interface FastEthernet
                                1/2 ip address  192.168.1.1 255.255.255.0
admin@ncs(config-if)# commit dry-run outformat native
admin@ncs(config-if)# commit
```

Note the use of the command **commit dry-run outformat native** . This will display the net result device commands that will be generated over the native interface without actually committing them to the CDB or the devices. Exit from the NSO CLI and return to the Unix Shell. Inspect the CLI trace:

```
 less logs/ned-cisco-ios-c0.trace
```

More on Device Management

Device Groups

As seen above, ranges can be used to send configuration commands towards several devices. Device groups can be created to allow for grouped actions that does not require naming conventions. A group can reference any number of devices. A device can be part of any number of groups, and groups can be hierarchical.

The command sequence below creates a group of core devices and a group with all devices. Note that you can use tab completion when adding the device names into the group. Also note that it requires configuration mode. (If you are still in the Unix Shell from the steps above, do **$ncs_cli -C -u admin**)

```
admin@ncs(config)# devices device-group core device-name [ c0 c1 ]
admin@ncs(config-device-group-core)# commit
```

```
admin@ncs(config)# devices device-group all device-name c2 device-group core
admin@ncs(config-device-group-all)# commit

admin@ncs(config)# show full-configuration devices device-group
devices device-group all
 device-name  [ c2 ]
 device-group [ core ]
!
devices device-group core
 device-name [ c0 c1 ]
!

admin@ncs(config)# do show devices device-group
NAME  MEMBER        INDETERMINATES  CRITICALS  MAJORS  MINORS  WARNINGS
-------------------------------------------------------------------------
all   [ c0 c1 c2 ]  0               0          0       0       0
core  [ c0 c1 ]     0               0          0       0       0
```

Note well the "do show" which shows the operational data for the groups. Device groups has a member attribute that shows all member devices, flattening any group members.

Device groups can contain different devices as well as devices from different vendors. Configuration changes will be committed to each device in its native language without needing to be adjusted in NSO.

You can for example at this point use the group to check if all core are in sync:

```
admin@ncs# devices device-group core check-sync
sync-result {
    device c0
    result in-sync
}
sync-result {
    device c1
    result in-sync
}
```

Device Templates

Assume we would like to manage permit lists across devices. This can be achieved by defining templates and apply them to device groups. The following CLI sequence defines a tiny template, called `community-list`:

```
admin@ncs(config)# devices template community-list config ios:ip \
                                community-list standard test1 \
                                permit permit-list 64000:40
admin@ncs(config-permit-list-64000:40)# commit
Commit complete.
admin@ncs(config-permit-list-64000:40)# top

admin@ncs(config)# show full-configuration devices template
devices template community-list
 config
  ios:ip community-list standard test1
   permit permit-list 64000:40
   !
  !
 !
!
[ok][2013-08-09 11:27:28]
```

This can now be applied to a device group:

```
admin@ncs(config)# devices device-group core apply-template \
                                 template-name community-list
admin@ncs(config)# show configuration
devices device c0
 config
  ios:ip community-list standard test1 permit 64000:40
 !
!
devices device c1
 config
  ios:ip community-list standard test1 permit 64000:40
 !
!
admin@ncs(config)# commit dry-run outformat native
native {
    device {
        name c0
        data ip community-list standard test1 permit 64000:40
    }
    device {
        name c1
        data ip community-list standard test1 permit 64000:40
    }
}
admin@ncs(config)# commit
Commit complete.
```

What if the device group `core` contained different vendors? Since the configuration is written in IOS the above template would not work Juniper devices. Templates can be used on different device types (read NEDs) by using a prefix for the device model. The template would then look like:

```
template community-list {
  config {
    junos:configuration {
    ...
    }
    ios:ip {
    ...
    }
```

The above indicates how NSO manages different models for different device-types. When NSO connects to the devices the NEDs checks the device type and revision and returns that to NSO. This can be inspected (note, in operational mode):

```
admin@ncs# show devices device module
NAME  NAME                       REVISION    FEATURES  DEVIATIONS
-------------------------------------------------------------------
c0    tailf-ned-cisco-ios        2014-02-12  -         -
      tailf-ned-cisco-ios-stats  2014-02-12  -         -
c1    tailf-ned-cisco-ios        2014-02-12  -         -
      tailf-ned-cisco-ios-stats  2014-02-12  -         -
c2    tailf-ned-cisco-ios        2014-02-12  -         -
      tailf-ned-cisco-ios-stats  2014-02-12  -         -
```

So here we see that `c0` uses a `tailf-ned-cisco-ios` module which tells NSO which data-model to use for the device. Every NED comes with a YANG data-model for the device. This renders the NSO data-store (CDB) schema, the NSO CLI, WebUI and southbound commands.

The model introduces namespace prefixes for every configuration item. This also resolves issues around different vendors using the same configuration command for different configuration elements. Note that every item is prefixed with ios:

```
admin@ncs# show running-config devices device c0 config ios:ip community-list
devices device c0
 config
  ios:ip community-list 1 permit
  ios:ip community-list 2 deny
  ios:ip community-list standard s permit
  ios:ip community-list standard test1 permit 64000:40
 !
!
```

Another important question is how to control if the template shall merge the list or replace the list. This is managed via "tags". The default behavior of templates is to merge the configuration. Tags can be inserted at any point in the template. Tag values are `merge`, `replace`, `delete`, `create` and `nocreate`.

Assume that `c0` has the following configuration:

```
admin@ncs# show running-config devices device c0 config ios:ip community-list
devices device c0
 config
  ios:ip community-list 1 permit
  ios:ip community-list 2 deny
  ios:ip community-list standard s permit}
```

If we apply the template the default result would be:

```
admin@ncs# show running-config devices device c0 config ios:ip community-list
devices device c0
 config
  ios:ip community-list 1 permit
  ios:ip community-list 2 deny
  ios:ip community-list standard s permit
  ios:ip community-list standard test1 permit 64000:40
 !
!
```

We could change the template in the following way to get a result where the permit list would be *replaced* rather than *merged*. When working with tags in templates it is often helpful to view the template as a tree rather then a command view. The CLI has a display option for showing a curly-braces tree view that corresponds to the data-model structure rather then the command set. This makes it easier to see where to add tags.

```
admin@ncs(config)# show full-configuration devices template
devices template community-list
 config
  ios:ip community-list standard test1
   permit permit-list 64000:40
   !
  !
 !
!
admin@ncs(config)# show full-configuration devices \
                                template | display curly-braces
template community-list {
    config {
        ios:ip {
            community-list {
                standard test1 {
                    permit {
                        permit-list 64000:40;
                    }
                }
            }
```

```
        }
    }
}


admin@ncs(config)# tag add devices template community-list
                                config ios:ip community-list replace
admin@ncs(config)# commit
Commit complete.
admin@ncs(config)# show full-configuration devices
                                template | display curly-braces
template community-list {
    config {
        ios:ip {
            /* Tags: replace */
            community-list {
                standard test1 {
                    permit {
                        permit-list 64000:40;
                    }
                }
            }
        }
    }
}
```

Different tags can be added across the template tree. If we now apply the template to device `c0` which already have community lists the following happens:

```
admin@ncs(config)# show full-configuration devices device c0 \
                                config ios:ip community-list
devices device c0
 config
  ios:ip community-list 1 permit
  ios:ip community-list 2 deny
  ios:ip community-list standard s permit
  ios:ip community-list standard test1 permit 64000:40
 !
!
admin@ncs(config)# devices device c0 apply-template \
                                template-name community-list
admin@ncs(config)# show configuration
devices device c0
 config
  no ios:ip community-list 1 permit
  no ios:ip community-list 2 deny
  no ios:ip community-list standard s permit
 !
!
```

Any existing values in the list are replaced in this case. The following tags are available:

- *merge* (default): the template changes will be merged with the existing template
- *replace*: the template configuration will be replaced by the new configuration
- *create*: the template will create those nodes which does not exist. If a node already exists this will result in an error.
- *nocreate*: the merge will only affect configuration items that already exist in the template. It will never create the configuration with this tag, or any associated commands inside it. It will only modify existing configuration structures.
- *delete*: delete anything from this point

Note that a template can have different tags along the tree nodes.

A "problem" with the above template is that every value is hard-coded. What if you wanted a template where the community-list name and permit-list value are variables passed to the template when applied? Any part of a template can be a variable, (or actually an XPATH expression). We can modify the template to use variables in the following way:

```
admin@ncs(config)# no devices template community-list config ios:ip \
                              community-list standard test1
admin@ncs(config)# devices template community-list config ios:ip \
                              community-list standard \
                              {$LIST-NAME} permit permit-list {$AS}

admin@ncs(config-permit-list-{$AS})# commit
Commit complete.

admin@ncs(config-permit-list-{$AS})# top
admin@ncs(config)# show full-configuration devices template
devices template community-list
 config
  ios:ip community-list standard {$LIST-NAME}
   permit permit-list {$AS}
   !
  !
 !
!
```

The template now requires two parameters when applied (tab completion will prompt for the variable):

```
admin@ncs(config)# devices device-group all apply-template
template-name community-list variable { name LIST-NAME value 'test2' }
variable { name AS value '60000:30' }

admin@ncs(config)# commit
```

Note, that the `replace` tag was still part of the template and it would delete any existing community lists, which is probably not the desired outcome in the general case.

The template mechanism described so far is "fire-and-forget". The templates does not have any memory of what happened to the network, which devices they touched. A user can modify the templates without anything happening to the network until an explicit `apply-template` action is performed. (Templates are of course as all configuration changes done as a transaction). NSO also supports service templates that are more "advanced" in many ways, more information on this will be presented later in this guide.

Policies

In order to make sure that configuration is applied according to site or corporate rules you can use policies. Policies are validated at every commit, they can be of type *error* that implies that the change cannot go through or a *warning* which means that you have to confirm a configuration that gives a warning.

A policy is composed of:

1. Policy *name*
2. *Iterator*: loop over a path in the model, for example all devices, all services of a specific type.
3. *Expression*: a boolean expression that must be true for every node returned from the iterator, for example, snmp must be turned on.
4. *Warning* or *error*: a message displayed to the user. If it is of type warning the user can still commit the change, if of type error the change cannot be made.

An example is shown below:

```
admin@ncs(config)# policy rule class-map
Possible completions:
  error-message     Error message to print on expression failure
  expr              XPath 1.0 expression that returns a boolean
  foreach           XPath 1.0 expression that returns a node set
  warning-message   Warning message to print on expression failure

admin@ncs(config)# policy rule class-map foreach /devices/device \
       expr config/ios:class-map[name='a'] \
       warning-message "Device {name} must have a class-map a"

admin@ncs(config-rule-class-map)# top

admin@ncs(config)# commit
Commit complete.

admin@ncs(config)# show full-configuration policy
policy rule class-map
 foreach         /devices/device
 expr            config/ios:class-map[ios:name='a']
 warning-message "Device {name} must have a class-map a"
!
```

Now if we try to delete a class-map 'a' we will get a policy violation.

```
admin@ncs(config)# no devices device c2 config ios:class-map match-all a
admin@ncs(config)# validate
Validation completed with warnings:
  Device c2 must have a class-map a

admin@ncs(config)# commit
The following warnings were generated:
  Device c2 must have a class-map a
Proceed? [yes,no] yes
Commit complete.

admin@ncs(config)# validate
Validation completed with warnings:
  Device c2 must have a class-map a
```

The **{name}** variable refers to the node-set from the iterator. This node-set will be the list of devices in NSO and the devices have an attribute called 'name'.

In order to understand the syntax for the expressions a pipe-target in the CLI can be used:

```
admin@ncs(config)# show full-configuration devices device c2 config \
                                ios:class-map | display xpath
/ncs:devices/ncs:device[ncs:name='c2']/ncs:config/ \
ios:class-map[ios:name='cmap1']/ios:prematch match-all
...
```

In order to debug policies look at the end of `logs/xpath.trace`. This file will show all validated XPATH expressions and any errors.

```
4-Sep-2014::11:05:30.103 Evaluating XPath for policy: class-map:
  /devices/device
get_next(/ncs:devices/device) = {c0}
XPath policy match: /ncs:devices/device{c0}
get_next(/ncs:devices/device{c0}) = {c1}
XPath policy match: /ncs:devices/device{c1}
```

```
get_next(/ncs:devices/device{c1}) = {c2}
XPath policy match: /ncs:devices/device{c2}
get_next(/ncs:devices/device{c2}) = false
exists("/ncs:devices/device{c2}/config/class-map{a}") = true
exists("/ncs:devices/device{c1}/config/class-map{a}") = true
exists("/ncs:devices/device{c0}/config/class-map{a}") = true
```

Validation scripts can also be defined in Python, see more about that in "Plug and Play scripts".

Out-of-band changes, transactions, pre-provisioning

In reality, network engineers will still modify configurations using other tools like out of band CLI or other management interfaces. It is important to understand how does NSO manage this. The NSO network simulator supports CLI towards the devices. For example we can use the IOS CLI on say `c0` and delete a permit-list. From the UNIX shell start a CLI session towards `c0`.

```
$ ncs-netsim cli-i c0

c0> enable
c0# configure
Enter configuration commands, one per line. End with CNTL/Z.

c0(config)# show full-configuration ip community-list
ip community-list standard test1 permit
ip community-list standard test2 permit 60000:30
c0(config)# no ip community-list standard test2
c0(config)#
c0# exit
$
```

Start the NSO CLI again:

```
$ ncs_cli -C -u admin
```

NSO detects if its configuration copy in CDB differs from the configuration in the device. Various strategies are used depending on device support; transaction-ids, time-stamps, configuration hash-sums. For example a NSO user can request a check-sync operation:

```
admin@ncs# devices check-sync
sync-result {
    device c0
    result out-of-sync
    info got: e54d27fe58fda990797d8061aa4d5325 expected: 36308bf08207e994a8a83af710effbf0

}
sync-result {
    device c1
    result in-sync
}
sync-result {
    device c2
    result in-sync
}

admin@ncs# devices device-group core check-sync
sync-result {
    device c0
    result out-of-sync
    info got: e54d27fe58fda990797d8061aa4d5325 expected: 36308bf08207e994a8a83af710effbf0

}
```

```
sync-result {
    device c1
    result in-sync
}
```

NSO can also compare the configurations with the CDB and show the difference:

```
admin@ncs# devices device c0 compare-config
diff
 devices {
     device c0 {
         config {
             ios:ip {
                 community-list {
+                    standard test1 {
+                        permit {
+                        }
+                    }
-                    standard test2 {
-                        permit {
-                            permit-list 60000:30;
-                        }
-                    }
                 }
             }
         }
     }
 }
```

At this point we can choose if we want to use the configuration stored in the CDB as the valid configuration or the configuration on the device:

```
admin@ncs# devices sync-
Possible completions:
  sync-from   Synchronize the config by pulling from the devices
  sync-to     Synchronize the config by pushing to the devices

admin@ncs# devices sync-to
```

In the above example we chose to overwrite the device configuration from NSO.

NSO will also detect out-of-sync when committing changes. In the following scenario a local `c0` CLI user adds an interface. Later a NSO user tries to add an interface:

```
$ ncs-netsim cli-i c0

c0> enable
c0# configure
Enter configuration commands, one per line. End with CNTL/Z.
c0(config)#  interface FastEthernet 1/0 ip address 192.168.1.1 255.255.255.0
c0(config-if)#
c0# exit

$ ncs_cli -C -u admin

admin@ncs# config
Entering configuration mode terminal

admin@ncs(config)# devices device c0 config ios:interface \
       FastEthernet1/1 ip address 192.168.1.1 255.255.255.0

admin@ncs(config-if)# commit
Aborted: Network Element Driver: device c0: out of sync
```

At this point we actually have two diffs:

1. The device and NSO CDB (**devices device compare-config**)
2. The on-going transaction and CDB (**show configuration**)

```
admin@ncs(config)# devices device c0 compare-config
diff
 devices {
     device c0 {
         config {
             ios:interface {
                 FastEthernet 1/0 {
                     ip {
                         address {
                             primary {
+                                mask 255.255.255.0;
+                                address 192.168.1.1;
                             }
                         }
                     }
                 }
             }
         }
     }
 }

admin@ncs(config)# show configuration
devices device c0
 config
  ios:interface FastEthernet1/1
   ip address 192.168.1.1 255.255.255.0
  exit
 !
!
```

To resolve this you can choose to synchronize the configuration between the devices and the CDB before committing. There is also an option to over-ride the out-of-sync check:

```
admin@ncs(config)# commit no-out-of-sync-check
```

or:

```
admin@ncs(config)# devices global-settings out-of-sync-commit-behaviour
Possible completions:
  accept  reject
```

As noted before, all changes are applied as complete transactions of all configurations on all of the devices. Either all configuration changes are completed successfully or all changes are removed entirely. Consider a simple case where one of the devices is not responding. For the transaction manager an error response from a device or a non-responding device are both errors and the transaction should automatically rollback to the state before the commit command was issued.

Stop c0:

```
$ ncs-netsim stop c0
DEVICE c0 STOPPED
```

Go back to the NSO CLI and perform a configuration change over c0 and c1:

```
admin@ncs(config)# devices device c0 config ios:ip community-list \
                                 standard test3 permit 50000:30
admin@ncs(config-config)# devices device c1 config ios:ip \
```

```
                                community-list standard test3 permit 50000:30

admin@ncs(config-config)# top
admin@ncs(config)# show configuration
devices device c0
 config
  ios:ip community-list standard test3 permit 50000:30
 !
!
devices device c1
 config
  ios:ip community-list standard test3 permit 50000:30
 !
!

admin@ncs(config)# commit
Aborted: Failed to connect to device c0: connection refused: Connection refused
admin@ncs(config)# *** ALARM connection-failure: Failed to connect to
device c0: connection refused: Connection refused
```

NSO sends commands to all devices in parallel, not sequentially. If any of the devices fails to accept the changes or reports an error, NSO will issue a rollback to the other devices. Note, this works also for non-transactional devices like IOS CLI and SNMP. This works even for non-symmetrical cases where the rollback command sequence is not just the reverse of the commands. NSO does this by treating the rollback as it would any other configuration change. NSO can use the current configuration and previous configuration and generate the commands needed to rollback from the configuration changes.

The diff configuration is still in the private CLI session, it can be restored, modified (if the error was due to something in the config), or in some cases, fix the device.

NSO is not a "best effort" configuration management system. The error reporting coupled with the ability to completely rollback failed changes to the devices, ensures that the configurations stored in the CDB and the configurations on the devices are always consistent and that no failed or "orphan" configurations are left on the devices.

First of all, if the above was not a multi-device transaction, meaning that the change should be applied independently device per device, then it is just a matter of performing the commit between the devices.

Second, NSO has a commit flag "commit-queue async" or "commit-queue sync". Commit queues should primarily be used for performance reasons when doing configuration changes in large networks. Atomic transactions comes with a cost, the critical section of the data-store is locked per in the final piece of commit in each transaction. So, in cases where there are northbound systems of NSO that generates many simultaneous large configuration changes these might get queued. Commit queues will send the device commands as separate transactions, so the lock is much shorter. If any device fails an alarm will be raised.

```
admin@ncs(config)# commit commit-queue async
commit-queue-id 2236633674
Commit complete.

admin@ncs(config)# do show devices commit-queue
                           KILO
                           BYTES             WAITING  TRANSIENT
ID          AGE  STATUS    SIZE   DEVICES    FOR      ERRORS     DONE
-------------------------------------------------------------------------
2236633674  11   executing 1      [ c1 c0 ]  -        [ c0 ]     [ c1
]
```

Go to the UNIX shell and start the device and monitor the commit queues.

```
$ncs-netsim start c0
```

```
DEVICE c0 OK STARTED

$ ncs_cli -C -u admin

admin@ncs# show devices commit-queue
                          KILO
                          BYTES             WAITING  TRANSIENT
ID          AGE  STATUS   SIZE   DEVICES    FOR      ERRORS     DONE
-------------------------------------------------------------------------
2236633674  92   blocking 1      [ c1 c0 ]  -        [ c0 ]     [ c1 ]

admin@ncs# show devices commit-queue
                          KILO
                          BYTES             WAITING  TRANSIENT
ID          AGE  STATUS   SIZE   DEVICES    FOR      ERRORS     DONE
-------------------------------------------------------------------------
2236633674  94   executing 1      [ c1 c0 ]  -        [ c0 ]     [ c1 ]

...

admin@ncs# show devices commit-queue
%  No entries found.
```

Devices can also be pre-provisioned, this means that the configuration can be prepared in NSO and pushed to the device when it is available. To illustrate this we can start by adding a new device to NSO that is *not* available in the network simulator:

```
admin@ncs(config)# devices device c3 address 127.0.0.1 port 10030 \
                              authgroup default device-type cli
                              ned-id cisco-ios
admin@ncs(config-device-c3)# state admin-state southbound-locked
admin@ncs(config-device-c3)# commit
```

Above we added a new device to NSO with IP address local host and port 10030. This device does not exist in the network simulator. We can tell NSO not to send any commands southbound by setting the **admin-state** to **southbound-locked** (actually the default). This means that all configuration changes will succeed, the result will be stored in CDB. At any point in time when the device is available in the network the state can be changed and the complete configuration pushed to the new device. The CLI sequence below also illustrates a powerful copy configuration command which can copy any configuration from one device to another. The from and to paths are separated by the keyword `to`.

```
admin@ncs(config)# copy cfg merge devices device c0 config \
                              ios:ip community-list to \
                              devices device c3 config ios:ip community-list
admin@ncs(config)# show configuration
devices device c3
 config
  ios:ip community-list standard test2 permit 60000:30
  ios:ip community-list standard test3 permit 50000:30
 !
!


admin@ncs(config)# commit

admin@ncs(config)# devices check-sync
...

sync-result {
    device c3
    result locked
```

```
}
```

As shown above `check-sync` operations will tell the user that the device is southbound locked. When the device is available in the network the device can be synchronized with the current configuration in the CDB using the sync-to action.

Conflict Resolution

Different users or management tools can of course run parallel sessions to NSO. All on-going sessions have a logical copy of CDB. An important case needs to be understood if there is a conflict when multiple users attempt to modify the same device configuration at the same time with different changes. First lets look at the CLI sequence below, user admin to the left, user joe to the right.

```
admin@ncs(config)# devices device c0 config \
           ios:snmp-server community fozbar

      joe@ncs(config)# devices device c0 config \
            ios:snmp-server community fezbar

admin@ncs(config-config)# commit

      System message at 2014-09-04 13:15:19...
      Commit performed by admin via console using cli.

      joe@ncs(config-config)# commit
      joe@ncs(config)# show full-configuration devices device c0 \
                                 config ios:snmp-server
      devices device c0
        config
          ios:snmp-server community fezbar
          ios:snmp-server community fozbar
        !
      !
```

There is no conflict in the above sequence, community is a list so both joe and admin can add items to the list. Note that user joe gets information about user admin committing.

On the other hand if two users modifies a single item that is not a list the following happens:

```
admin@ncs(config)# devices device c0 config ios:power redundancy-mode redundant


    joe@ncs(config)# devices device c0 config ios:power redundancy-mode combined

admin@ncs% commit

    System message at 2014-09-04 13:23:10...
    Commit performed by admin via console using cli.

      joe@ncs(config-config)# commit
      Aborted: there are conflicts.
      -------------------------------------------------------------------------
      Resolve needed before configuration can be committed. View conflicts with
      the command 'show configuration' and execute the command 'resolved' when done,
      or exit configuration mode to abort.
      Conflicting configuration items are indicated with a leading '!'
      Conflicting users: admin
      -------------------------------------------------------------------------
      joe@ncs(config)# show configuration
      devices device c0
        config
```

```
!  ios:power redundancy-mode combined
  !
!

joe@ncs(config)# resolved
joe@ncs(config)# commit
Commit complete.
```

In this case joe commits a change to `redundancy-mode` after admin and a conflict resolution process starts. Joe can see the value set by admin and his own value and choose resolution.

CHAPTER 4

Network Element Drivers and Adding Devices

Overview

Network Element Drivers, NEDs, provides the connectivity between NSO and the devices. NEDs are installed as NSO packages. (For information on how to add a package for a new device type, see the section called “Packages”)

To see the list of installed packages (you will not see the F5 BigIP):

```
admin@ncs# show packages
packages package cisco-ios
 package-version 3.0
 description     "NED package for Cisco IOS"
 ncs-min-version [ 3.0.2 ]
 directory       ./state/packages-in-use/1/cisco-ios
 component upgrade-ned-id
  upgrade java-class-name com.tailf.packages.ned.ios.UpgradeNedId
 component cisco-ios
  ned cli ned-id  cisco-ios
  ned cli java-class-name com.tailf.packages.ned.ios.IOSNedCli
  ned device vendor Cisco
NAME      VALUE
---------------------
show-tag  interface

 oper-status up
packages package f5-bigip
 package-version 1.3
 description     "NED package for the F5 BigIp FW/LB"
 ncs-min-version [ 3.0.1 ]
 directory       ./state/packages-in-use/1/bigip
 component f5-bigip
  ned generic java-class-name com.tailf.packages.ned.bigip.BigIpNedGeneric
  ned device vendor F5
 oper-status up
!
```

The core parts of a NED are:

1. *Data-Model* : independent of underlying device interface technology NEDs come with a data-model in YANG that specifies configuration data and operational data that is supported for the device. For native NETCONF devices the YANG comes from the device, for JunOS NSO generates the model from the JunOS XML schema, for SNMP devices NSO generates the model from the MIBs. For CLI devices the NED designer wrote the YANG to map the CLI.

   NSO only cares about the data that is in the model for the NED. The rest is ignored. See the NED documentation to learn more about what is covered for the NED.
2. *Code*: for NETCONF and SNMP devices there is no code. For CLI devices there is a minimum of code managing connecting over ssh/telnet and looking for version strings. The rest is auto-rendered from the data-model.

There are four categories of NEDs depending on the device interface:

1. NETCONF NED: the device supports NETCONF, for example Juniper.
2. CLI NED: any device with a CLI that resembles a Cisco CLI
3. Generic NED: proprietary protocols like REST, non-Cisco CLIs.
4. SNMP NED: a SNMP device.

Device Authentication

Every device needs an authgroup that tells NSO how to authenticate to the device:

```
admin@ncs(config)# show full-configuration devices authgroups
devices authgroups group default
 umap admin
  remote-name     admin
  remote-password $4$wIo7Yd068FRwhYYI0d4IDw==
 !
 umap oper
  remote-name     oper
  remote-password $4$zp4zerM68FRwhYYI0d4IDw==
 !
!
devices authgroups snmp-group default
 default-map community-name public
 umap admin
  usm remote-name admin
  usm security-level auth-priv
  usm auth md5 remote-password $4$wIo7Yd068FRwhYYI0d4IDw==
  usm priv des remote-password $4$wIo7Yd068FRwhYYI0d4IDw==
 !
!
```

The CLI snippet above shows that there is a mapping from NSO users admin and oper to the remote user and password to be used on the devices. There are two options, either a mapping from local user to remote user or to pass the credentials. Below is shown a CLI example to create a new `authgroup foobar` and map NSO user jim:

```
admin@ncs(config)# devices authgroups group foobar umap joe same-pass same-user
admin@ncs(config-umap-joe)# commit
```

This authgroup will pass on joes credentials to the device.

There is a similar structure for SNMP `devices authgroups snmp-group` that supports SNMPv1/v2c, and SNMPv3 authentication.

The SNMP authgroup above has a default authgroup for not mapped users.

Connecting devices for different NED Types

Make sure you know the authentication information and created authgroups as above. Also try all information like port numbers, authentication information and that you can read and set the configuration over for example CLI if it is a CLI NED. So if it is a CLI device try to ssh (or telnet) to the device and do show and set configuration first of all.

All devices have a `admin-state` with default value `southbound-locked`. This means that if you do not set this value to unlocked no commands will be sent to the device.

CLI NEDs

(See also `examples.ncs/getting-started/using-ncs/2-real-device-cisco-ios`). Straightforward, adding a new device on a specific address, standard ssh port:

```
admin@ncs(config)# devices device c7 address 1.2.3.4 port 22 \
                                device-type cli ned-id cisco-ios
admin@ncs(config-device-c7)# authgroup
Possible completions:
  default  foobar
admin@ncs(config-device-c7)# authgroup default
admin@ncs(config-device-c7)# state admin-state unlocked
admin@ncs(config-device-c7)# commit
```

.

NETCONF NEDs, JunOS

See also `/examples.ncs/getting-started/using-ncs/3-real-device-juniper`. Make sure that NETCONF over SSH is enabled on the JunOS device:

```
junos1% show system services
ftp;
ssh;
telnet;
netconf {
    ssh {
        port 22;
    }
}
```

Then you can create a NSO netconf device as:

```
admin@ncs(config)# devices device junos1 address junos1.lab port 22 \
                                authgroup foobar device-type netconf
admin@ncs(config-device-junos1)# state admin-state unlocked
admin@ncs(config-device-junos1)# commit
```

SNMP NEDs

(See also `examples.ncs/snmp-ned/basic/README` .) First of all lets explain SNMP NEDs a bit. By default all read-only objects are mapped to operational data in NSO and read-write objects are mapped to configuration data. This means that a sync-from operation will load read-write objects into NSO. How can you reach read-only objects? Note the following is true for all NED types that have modelled operational data. The device configuration exists at `devices device config` and has a copy in CDB. NSO can speak live to the device to fetch for example counters by using the path `devices device live-status`:

```
admin@ncs# show devices device r1 live-status SNMPv2-MIB
live-status SNMPv2-MIB system sysDescr "Tail-f ConfD agent - r1"
live-status SNMPv2-MIB system sysObjectID 1.3.6.1.4.1.24961
live-status SNMPv2-MIB system sysUpTime 4253
live-status SNMPv2-MIB system sysContact ""
live-status SNMPv2-MIB system sysName ""
live-status SNMPv2-MIB system sysLocation ""
live-status SNMPv2-MIB system sysServices 72
live-status SNMPv2-MIB system sysORLastChange 0
live-status SNMPv2-MIB snmp snmpInPkts 3
live-status SNMPv2-MIB snmp snmpInBadVersions 0
live-status SNMPv2-MIB snmp snmpInBadCommunityNames 0
live-status SNMPv2-MIB snmp snmpInBadCommunityUses 0
live-status SNMPv2-MIB snmp snmpInASNParseErrs 0
live-status SNMPv2-MIB snmp snmpEnableAuthenTraps disabled
live-status SNMPv2-MIB snmp snmpSilentDrops 0
live-status SNMPv2-MIB snmp snmpProxyDrops 0
live-status SNMPv2-MIB snmpSet snmpSetSerialNo 2161860
```

In many cases SNMP NEDs are used for reading operational data in parallel with a CLI NED for writing and reading configuration data. More on that later.

Before trying NSO use net-snmp command line tools or your favorite SNMP Browser to try that all settings are ok.

Adding a SNMP device assuming the NED is in place:

```
admin@ncs(config)# show full-configuration devices device r1
devices device r1
 address 127.0.0.1
 port    11023
 device-type snmp version v2c
 device-type snmp snmp-authgroup default
 state admin-state unlocked
!
admin@ncs(config)# show full-configuration devices device r2
devices device r2
 address 127.0.0.1
 port    11024
 device-type snmp version v3
 device-type snmp snmp-authgroup default
 device-type snmp mib-group [ basic snmp ]
 state admin-state unlocked
!
```

MIB Groups are important. A MIB group is just a named collection of SNMP MIB Modules. If you do not specify any MIB group for a device, NSO will try with all known MIBs. It is possible to create MIB groups with wild-cards such as CISCO*.

```
admin@ncs(config)# show full-configuration devices mib-group
devices mib-group basic
 mib-module [ BASIC-CONFIG-MIB ]
!
devices mib-group snmp
 mib-module [ SNMP* ]
!
```

Generic NEDs

Generic devices are typically configured like a CLI device. Make sure you set the right address, port, protocol and authentication information.

Below follows an example to setup NSO with F5 BigIP:

```
admin@ncs(config)# devices device bigip01 address 192.168.1.162 \
                                port 22 device-type generic ned-id f5-bigip
admin@ncs(config-device-bigip01)# state admin-state southbound-locked
admin@ncs(config-device-bigip01)# authgroup
Possible completions:
  default  foobar
admin@ncs(config-device-bigip01)# authgroup default
admin@ncs(config-device-bigip01)# commit
```

Multi-NED Devices

Assume you have a Cisco device that you would like NSO to configure over CLI but read statistics over SNMP. This can be achieved by adding settings for "live-device-protocol":

```
admin@ncs(config)# devices device c0 live-status-protocol snmp \
                              device-type snmp version v1 \
                              snmp-authgroup default mib-group [ snmp ]
admin@ncs(config-live-status-protocol-snmp)# commit


admin@ncs(config)# show full-configuration devices device c0
devices device c0
 address   127.0.0.1
 port      10022
 !
 authgroup default
 device-type cli ned-id cisco-ios
 live-status-protocol snmp
  device-type snmp version v1
  device-type snmp snmp-authgroup default
  device-type snmp mib-group [ snmp ]
 !
```

Device c0 have a config tree from the cli NED and a live-status tree (read-only) from the SNMP NED using all MIBs in group snmp.

Administrative State for Devices

Devices have an `admin-state` with following values:

- *unlocked*: the device can be modified and changes will be propagated to the real device.
- *southbound-locked*: the device can be modified but changes will be *not* be propagated to the real device. Can be used to prepare configurations before the device is available in the network.
- *locked*: the device can only be read.

The admin-state value *southbound-locked* is default. This means if you create a new device without explicitly setting this value configuration changes will not propagate to the network. To see default values use the pipe target `details`

```
admin@ncs(config)# show full-configuration devices device c0 | details
```

Trouble-shooting NEDs

In order to analyze NED problems, turn on the tracing for a device and look at the trace file contents.

```
admin@ncs(config)# show full-configuration devices global-settings
devices global-settings trace-dir ./logs
```

```
admin@ncs(config)# devices device c0 trace raw
admin@ncs(config-device-c0)# commit

admin@ncs(config)# devices device c0 disconnect
admin@ncs(config)# devices device c0 connect
```

NSO pools ssh connections and trace settings are only affecting new connections so therefore any open connection must be closed before the trace setting will take effect. Now you can inspect the raw communication between NSO and the device:

```
$ less logs/ned-c0.trace

admin connected from 127.0.0.1 using ssh on HOST-17
c0>
  *** output 8-Sep-2014::10:05:39.673 ***
enable

  *** input 8-Sep-2014::10:05:39.674 ***
 enable
c0#
  *** output 8-Sep-2014::10:05:39.713 ***
terminal length 0

  *** input 8-Sep-2014::10:05:39.714 ***
 terminal length 0
c0#
  *** output 8-Sep-2014::10:05:39.782 ***
terminal width 0

  *** input 8-Sep-2014::10:05:39.783 ***
 terminal width 0
0^M
c0#
  *** output 8-Sep-2014::10:05:39.839 ***
-- Requesting version string --
show version

  *** input 8-Sep-2014::10:05:39.839 ***
 show version
Cisco IOS Software, 7200 Software (C7200-JK9O3S-M), Version 12.4(7h), RELEASE SOFTWARE (fc1)^M
Technical Support: http://www.cisco.com/techsupport^M
Copyright (c) 1986-2007 by Cisco Systems, Inc.^M
...
```

If NSO fails in talking to the device the typical root causes are:

1. *Timeout problems*: some devices are slow to respond, latency on connections etc. Fine-tune the connect, read and write timeouts for the device:

   ```
   admin@ncs(config)# devices device c0
   Possible completions:
     ...
     connect-timeout           - Timeout in seconds for new connections
     ...
     read-timeout              - Timeout in seconds used when reading data
     ...
     write-timeout             - Timeout in seconds used when writing data
   ```

   These settings can be set in profiles shared by devices.

   ```
   admin@ncs(config)# devices profiles profile good-profile
   ```

```
Possible completions:
  connect-timeout   Timeout in seconds for new connections
  ned-settings      Control which device capabilities NCS uses
  read-timeout      Timeout in seconds used when reading data
  trace             Trace the southbound communication to devices
  write-timeout     Timeout in seconds used when writing data
```

2. *Device management interface problems*: examples, not enabled the NETCONF ssh subsystem on Juniper, not enabled the SNMP agent, using wrong port numbers etc. Use stand-alone tools to make sure you can connect, read configuration and write configuration over the device interface that NSO is using
3. *Access rights*: the NSO mapped user does not have access rights to do the operation on the device. Make sure the authgroups settings are ok, test them manually to read and write configuration with those credentials.
4. *NED data-model and device version problems*: if the device is upgraded and existing commands actually change in an incompatible way the NED has to be updated. This can be done by editing the YANG data-model for the device or by using Tail-f support.

CHAPTER 5

Managing Network Services

Overview

Up until this point, this user guide has described how to use NSO to configure devices. NSO can also manage the life-cycle for services like VPNs, BGP peers, ACLs. It is important to understand what is meant by service in this context.

1. NSO abstracts the device specific details. The user only needs to enter attributes relevant to the service.
2. The service instance has configuration data itself that can be represented and manipulated.
3. A service instance configuration change is applied to all affected devices.

These are the features NSO uses to support service configuration.

1. *Service Modeling*: network engineers can model the service attributes and the mapping to device configurations. For example, this means that a network engineer can specify at data-model for VPNs with router interfaces, VLAN id, VRF and route distinguisher.
2. *Service life-cycle*: while less sophisticated configuration management systems can only create an initial service instance in the network they do not support changing or deleting a service instance. With NSO you can at any point in time modify service elements like the VLAN id of a VPN and NSO can generate the corresponding changes to the network devices.
3. The NSO *service instance* has configuration data that can be represented and manipulated. The service model run-time updates all NSO northbound interfaces so a network engineer can view and manipulate the service instance over CLI, WebUI, REST etc.
4. NSO maintains *references between service instances and device configuration*. This means that a VPN instance knows exactly which device configurations it created/modified. Every configuration stored in the CDB is mapped to the service instance that created it.

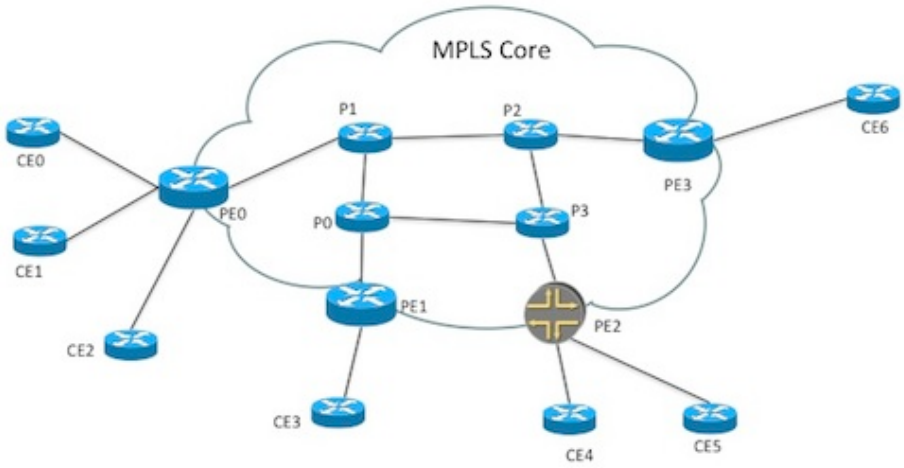
A Service Example

An example is the best method to illustrate how services are created and used in NSO. As described in the sections about devices and NEDs it was said that NEDs come in packages. The same is true for services, either if you do design the services yourself or use ready-made service applications it ends up in a package that is loaded into NSO.

Tip At the Tail-f website you can find a video presentation of this demo.

The example `examples.ncs/service-provider/mpls-vpn` will be used to explain NSO Service Management features. This example illustrates Layer3 VPNs in a service provider MPLS network. The example network consists of Cisco ASR 9k and Juniper core routers (P and PE) and Cisco IOS based CE routers. The Layer3 VPN service configures the CE/PE routers for all endpoints in the VPN with BGP as the CE/PE routing protocol. Layer2 connectivity between CE and PE routers are expected to be done through a Layer2 ethernet access network, which is out of scope for this example. The Layer3 VPN service includes VPN connectivity as well as bandwidth and QOS parameters.

Figure 5. A L3 VPN Example



The service configuration only has references to CE devices for the end-points in the VPN. The service mapping logic reads from a simple topology model that is configuration data in NSO, outside the actual service model, and derives what other network devices to configure. The topology information has two parts. The first part lists connections in the network and is used by the service mapping logic to find out which PE router to configure for an endpoint. The snippets below show the configuration output in the Cisco style NSO CLI.

```
topology connection c0
 endpoint-1 device ce0 interface GigabitEthernet0/8 ip-address 192.168.1.1/30
 endpoint-2 device pe0 interface GigabitEthernet0/0/0/3 ip-address 192.168.1.2/30
 link-vlan 88
!
topology connection c1
 endpoint-1 device ce1 interface GigabitEthernet0/1 ip-address 192.168.1.5/30
 endpoint-2 device pe1 interface GigabitEthernet0/0/0/3 ip-address 192.168.1.6/30
 link-vlan 77
```

```
!
```

The second part lists devices for each role in the network and is in this example only used to dynamically render a network map in the Web UI.

```
topology role ce
 device [ ce0 ce1 ce2 ce3 ce4 ce5 ]
!
topology role pe
 device [ pe0 pe1 pe2 pe3 ]
!
```

QOS configuration in service provider networks is complex, and often require a lot of different variations. It is also often desirable to be able to deliver different levels of QOS. This example shows how a QOS policy configuration can be stored in NSO and be referenced from VPN service instances. Three different levels of QOS policies are defined; GOLD, SILVER and BRONZE with different queuing parameters.

```
qos qos-policy GOLD
 class BUSINESS-CRITICAL
  bandwidth-percentage 20
 !
 class MISSION-CRITICAL
  bandwidth-percentage 20
 !
 class REALTIME
  bandwidth-percentage 20
  priority
 !
!
qos qos-policy SILVER
 class BUSINESS-CRITICAL
  bandwidth-percentage 25
 !
 class MISSION-CRITICAL
  bandwidth-percentage 25
 !
 class REALTIME
  bandwidth-percentage 10
 !
```

Three different traffic classes are also defined with a DSCP value that will be used inside the MPLS core network as well as default rules that will match traffic to a class.

```
qos qos-class BUSINESS-CRITICAL
 dscp-value af21
 match-traffic ssh
  source-ip      any
  destination-ip any
  port-start     22
  port-end       22
  protocol       tcp
 !
!
qos qos-class MISSION-CRITICAL
 dscp-value af31
 match-traffic call-signaling
  source-ip      any
  destination-ip any
  port-start     5060
  port-end       5061
  protocol       tcp
 !
```

```
!
```

Running the example

Make sure you start clean, i.e. no old configuration data is present. If you have been running this or some other example before, make sure to stop any NSO or simulated network nodes (ncs-netsim) that you may have running. Output like 'connection refused (stop)' means no previous NSO was running and 'DEVICE ce0 connection refused (stop)...' no simulated network was running, which is good.

```
$ make stop clean all start
$ ncs_cli -u admin -C
```

This will setup the environment and start the simulated network.

Before creating a new L3VPN service we must sync the configuration from all network devices and then enter config mode. (A hint for this complete section is to have the README file from the example and cut and paste the CLI commands).

```
ncs# devices sync-from
sync-result {
    device ce0
    result true
}
...
ncs# config
Entering configuration mode terminal

ncs(config)# vpn l3vpn volvo
ncs(config-l3vpn-volvo)# as-number 65101
ncs(config-l3vpn-volvo)# endpoint main-office
ncs(config-endpoint-main-office)# ce-device    ce0
ncs(config-endpoint-main-office)# ce-interface GigabitEthernet0/11
ncs(config-endpoint-main-office)# ip-network   10.10.1.0/24
ncs(config-endpoint-main-office)# bandwidth    12000000
ncs(config-endpoint-main-office)# !
ncs(config-endpoint-main-office)# endpoint branch-office1
ncs(config-endpoint-branch-office1)# ce-device    ce1
ncs(config-endpoint-branch-office1)# ce-interface GigabitEthernet0/11
ncs(config-endpoint-branch-office1)# ip-network   10.7.7.0/24
ncs(config-endpoint-branch-office1)# bandwidth    6000000
ncs(config-endpoint-branch-office1)# !
ncs(config-endpoint-branch-office1)# endpoint branch-office2
ncs(config-endpoint-branch-office2)# ce-device    ce4
ncs(config-endpoint-branch-office2)# ce-interface GigabitEthernet0/18
ncs(config-endpoint-branch-office2)# ip-network   10.8.8.0/24
ncs(config-endpoint-branch-office2)# bandwidth    300000
ncs(config-endpoint-branch-office2)# !

ncs(config-endpoint-branch-office2)# top
ncs(config)# show configuration
vpn l3vpn volvo
 as-number 65101
 endpoint branch-office1
  ce-device    ce1
  ce-interface GigabitEthernet0/11
  ip-network   10.7.7.0/24
  bandwidth    6000000
 !
 endpoint branch-office2
  ce-device    ce4
```

```
  ce-interface GigabitEthernet0/18
  ip-network   10.8.8.0/24
  bandwidth    300000
 !
 endpoint main-office
  ce-device    ce0
  ce-interface GigabitEthernet0/11
  ip-network   10.10.1.0/24
  bandwidth    12000000
 !
!

ncs(config)# commit dry-run outformat native
native {
    device {
        name ce0
        data interface GigabitEthernet0/11
              description volvo local network
              ip address 10.10.1.1 255.255.255.0
             exit
...

(config)# commit
```

Add another VPN (prompts ommitted):

```
top
!
vpn l3vpn ford
as-number 65200
endpoint main-office
ce-device    ce2
ce-interface GigabitEthernet0/5
ip-network   192.168.1.0/24
bandwidth    10000000
!
endpoint branch-office1
ce-device    ce3
ce-interface GigabitEthernet0/5
ip-network   192.168.2.0/24
bandwidth    5500000
!
endpoint branch-office2
ce-device    ce5
ce-interface GigabitEthernet0/5
ip-network   192.168.7.0/24
bandwidth    1500000
!

commit
```

The above sequence showed how NSO can be used to manipulate service abstractions on top of devices. Services can be defined for various purpose such as VPNs, Access Control Lists, firewall rules etc. Support for services is added to NSO via a corresponding service package.

A service package in NSO comprises two parts:

1. Service model: the attributes of the service, input parameters given when creating the service. In this example name, as-number, and end-points.
2. Mapping: what is the corresponding configuration of the devices when the service is applied. The result of the mapping can be inspected by the **commit dry-run outformat native** command.

We later in this guide show how to define this, for now assume that the job is done.

Service-Life Cycle Management

Service Changes

When NSO applies services to the network, NSO stores the service configuration along with resulting device configuration changes. This is used as a base for the Tail-f FASTMAP algorithm which automatically can derive device configuration changes from a service change. So going back to the example L3 VPN above any part of `volvo` VPN instance can be modified. A simple change like changing the `as-number` on the service results in many changes in the network. NSO does this automatically.

```
ncs(config)# vpn l3vpn volvo as-number 65102
ncs(config-l3vpn-volvo)# commit dry-run outformat native
native {
    device {
        name ce0
        data no router bgp 65101
             router bgp 65102
              neighbor 192.168.1.2 remote-as 100
              neighbor 192.168.1.2 activate
              network 10.10.1.0
             !
...
ncs(config-l3vpn-volvo)# commit
```

Let us look at a more challenging modification. A common use-case is of course to add a new CE device and add that as an end-point to an existing VPN. Below follows the sequence to add two new CE devices and add them to the VPN's. (In the CLI snippets below we omit the prompt to enhance readability). First we add them to the topology.

```
top
!
topology connection c7
endpoint-1 device ce7 interface GigabitEthernet0/1 ip-address 192.168.1.25/30
endpoint-2 device pe3 interface GigabitEthernet0/0/0/2 ip-address 192.168.1.26/30
link-vlan 103
!
topology connection c8
endpoint-1 device ce8 interface GigabitEthernet0/1 ip-address 192.168.1.29/30
endpoint-2 device pe3 interface GigabitEthernet0/0/0/2 ip-address 192.168.1.30/30
link-vlan 104
!
ncs(config)#commit
```

Note well that the above just updates NSO local information on topological links. It has no effect on the network. The mapping for the L3 VPN services does a look-up in the topology connections to find the corresponding pe router.

Then we add them to the VPN's

```
top
!
vpn l3vpn ford
endpoint new-branch-office
ce-device    ce7
ce-interface GigabitEthernet0/5
ip-network   192.168.9.0/24
```

```
bandwidth    4500000
!
vpn l3vpn volvo
endpoint new-branch-office
ce-device    ce8
ce-interface GigabitEthernet0/5
ip-network   10.8.9.0/24
bandwidth    4500000
!
```

Before we send anything to the network, lets see look at the device configuration using dry-run. As you can see, both new CE devices are connected to the same PE router, but for different VPN customers.

```
ncs(config)#commit dry-run outformat native
```

And commit the configuration to the network

```
(config)#commit
```

Service Impacting out-of-band changes

Next we will show how NSO can be used to check if the service configuration in the network is up to date. In a new terminal window we connect directly to the device ce0 that is a Cisco device emulated by the tool ncs-netsim.

```
$ncs-netsim cli-c ce0
```

We will now reconfigure an edge interface that we previously configured using NSO.

```
ce0> enable
ce0# configure
Enter configuration commands, one per line. End with CNTL/Z.
ce0(config)# no policy-map volvo
ce0(config)# exit
ce0# exit
```

Going back to the terminal with NSO, check the status of the network configuration:

```
ncs# devices check-sync
sync-result {
    device ce0
    result out-of-sync
    info got: c5c75ee593246f41eaa9c496ce1051ea expected: c5288cc0b45662b4af88288d29be8667
...

ncs# vpn l3vpn * check-sync
vpn l3vpn ford check-sync
    in-sync true
vpn l3vpn volvo check-sync
    in-sync true

ncs# vpn l3vpn * deep-check-sync
vpn l3vpn ford deep-check-sync
    in-sync true
vpn l3vpn volvo deep-check-sync
    in-sync false


}
```

The CLI sequence above performs 3 different comparisons:

- Real device configuration versus device configuration copy in NSO CDB
- Expected device configuration from service perspective and device configuration copy in CDB.
- Expected device configuration from service perspective and real device configuration.

Notice that the service 'volvo' is out of sync from the service configuration. Use the check-sync outformat cli to see what the problem is:

```
ncs# vpn l3vpn volvo deep-check-sync outformat cli
cli  devices {
         devices {
             device ce0 {
                 config {
    +                ios:policy-map volvo {
    +                    class class-default {
    +                        shape {
    +                            average {
    +                                bit-rate 12000000;
    +                            }
    +                        }
    +                    }
    +                }
                 }
             }
         }
     }
```

Assume that a network engineer considers the real device configuration to be the master:

```
ncs# devices device ce0 sync-from
result true
```

And then restore the service:

```
ncs# vpn l3vpn volvo re-deploy dry-run { outformat native }
native {
    device {
        name ce0
        data policy-map volvo
               class class-default
                shape average 12000000
               !
              !

    }
}
ncs# vpn l3vpn volvo re-deploy
```

Service Deletion

In the same way as NSO can calculate any service configuration change it can also automatically delete the device configurations that resulted from creating services:

```
ncs(config)# no vpn l3vpn ford
ncs(config)# commit dry-run
cli  devices {
         device ce7 {
             config {
    -            ios:policy-map ford {
    -                class class-default {
    -                    shape {
    -                        average {
```

```
    -                           bit-rate 4500000;
    -                       }
    -                   }
    -               }
    -           }
...
```

It is important to understand the two diffs shown above. The first diff as an output to `show configuration` shows the diff at service level. The second diff shows the output generated by NSO to clean up the device configurations.

Finally, we commit the changes to delete the service.

```
(config)# commit
```

Viewing service configurations

Service instances live in the NSO data-store as well as a copy of the device configurations. NSO will maintain relationships between these two.

Show the configuration for a service

```
ncs(config)# show full-configuration vpn l3vpn
vpn l3vpn volvo
 as-number 65102
 endpoint branch-office1
  ce-device    ce1
  ce-interface GigabitEthernet0/11
  ip-network   10.7.7.0/24
  bandwidth    6000000
 !
...
```

You can ask NSO to list all devices that are touched by a service and vice versa:

```
ncs# show vpn l3vpn device-list
NAME   DEVICE LIST
----------------------------------------
volvo  [ ce0 ce1 ce4 ce8 pe0 pe2 pe3 ]


ncs# show devices device service-list
NAME  SERVICE LIST
-------------------------------------
ce0   [ "/l3vpn:vpn/l3vpn{volvo}" ]
ce1   [ "/l3vpn:vpn/l3vpn{volvo}" ]
ce2   [  ]
ce3   [  ]
ce4   [ "/l3vpn:vpn/l3vpn{volvo}" ]
ce5   [  ]
ce6   [  ]
ce7   [  ]
ce8   [ "/l3vpn:vpn/l3vpn{volvo}" ]
p0    [  ]
p1    [  ]
p2    [  ]
p3    [  ]
pe0   [ "/l3vpn:vpn/l3vpn{volvo}" ]
pe1   [  ]
pe2   [ "/l3vpn:vpn/l3vpn{volvo}" ]
pe3   [ "/l3vpn:vpn/l3vpn{volvo}" ]
```

Note that operational mode in the CLI was used above. Every service instance has an operational attribute that is maintained by the transaction manager and shows which device configuration it created. Furthermore every device configuration has backwards pointers to the corresponding service instances:

```
ncs(config)# show full-configuration devices device ce3 \
                   config | display service-meta-data
devices device ce3
 config
  ...
  /* Refcount: 1 */
  /* Backpointer: [ /l3vpn:vpn/l3vpn:l3vpn[l3vpn:name='ford'] ] */
  ios:interface GigabitEthernet0/2.100
   /* Refcount: 1 */
   description Link to PE / pe1 - GigabitEthernet0/0/0/5
   /* Refcount: 1 */
   encapsulation dot1Q 100
   /* Refcount: 1 */
   ip address 192.168.1.13 255.255.255.252
   /* Refcount: 1 */
   service-policy output ford
  exit

ncs(config)# show full-configuration devices device ce3 config \
                    | display curly-braces | display service-meta-data
...
ios:interface {
    GigabitEthernet 0/1;
    GigabitEthernet 0/10;
    GigabitEthernet 0/11;
    GigabitEthernet 0/12;
    GigabitEthernet 0/13;
    GigabitEthernet 0/14;
    GigabitEthernet 0/15;
    GigabitEthernet 0/16;
    GigabitEthernet 0/17;
    GigabitEthernet 0/18;
    GigabitEthernet 0/19;
    GigabitEthernet 0/2;
    /* Refcount: 1 */
    /* Backpointer: [ /l3vpn:vpn/l3vpn:l3vpn[l3vpn:name='ford'] ] */
    GigabitEthernet 0/2.100 {
        /* Refcount: 1 */
        description "Link to PE / pe1 - GigabitEthernet0/0/0/5";
        encapsulation {
            dot1Q {
                /* Refcount: 1 */
                vlan-id 100;
            }
        }
        ip {
            address {
                primary {
                    /* Refcount: 1 */
                    address 192.168.1.13;
                    /* Refcount: 1 */
                    mask    255.255.255.252;
                }
            }
        }
        service-policy {
            /* Refcount: 1 */
            output ford;
```

```
        }
    }
```

```
ncs(config)# show full-configuration devices device ce3 config \
                  | display service-meta-data | context-match Backpointer
devices device ce3
  /* Refcount: 1 */
  /* Backpointer: [ /l3vpn:vpn/l3vpn:l3vpn[l3vpn:name='ford'] ] */
  ios:interface GigabitEthernet0/2.100
devices device ce3
  /* Refcount: 2 */
  /* Backpointer: [ /l3vpn:vpn/l3vpn:l3vpn[l3vpn:name='ford'] ] */
  ios:interface GigabitEthernet0/5
```

The reference counter above makes sure that NSO will not delete shared resources until the last service instance is deleted. The context-match search is helpful, it displays the path to all matching configuration items.

Defining your own services

Overview

In order to have NSO deploy services across devices, two pieces are needed:

1. A service model in YANG: the service model shall define the black-box view of a service; which are the input parameters given when creating the service? This YANG model will render an update of all NSO northbound interfaces, for example the CLI.
2. Mapping, given the service input parameters, what is the resulting device configuration? This mapping can be defined in templates, code or a combination of both.

Defining the service model

The first step is to generate a skeleton package for a service. (For details on packages see the section called “Packages”). In order to reuse an existing environment for NSO and netsim we will reuse the `examples.ncs/getting-started/using-ncs/1-simulated-cisco-ios/` example. Make sure you have stopped any running NSO and netsim. Navigate to the simulated ios directory and create a new package for the VLAN service model:

```
$cd examples.ncs/getting-started/using-ncs/1-simulated-cisco-ios/packages
```

The next step is to create the template skeleton by using the `ncs-make-package` utility:

```
$ ncs-make-package --service-skeleton template --augment /ncs::services vlan
```

This results in a directory structure:

```
vlan
   load-dir
   package-meta-data.xml
   src
   templates
```

For now lets focus on the `src/yang/vlan.yang` file.

```
module vlan {

  namespace "http://com/example/vlan";
```

```
  prefix vlan;

  import ietf-inet-types {
    prefix inet;
  }

  import tailf-ncs {
    prefix ncs;
  }

  augment /ncs:services {
    list vlan {
      key name;

      uses ncs:service-data;
      ncs:servicepoint "vlan";

      leaf name {
        type string;
      }

      // may replace this with other ways of refering to the devices.
      leaf-list device {
        type leafref {
          path "/ncs:devices/ncs:device/ncs:name";
        }
      }

      // replace with your own stuff here
      leaf dummy {
        type inet:ipv4-address;
      }
    }
  }
}
```

If this is your first exposure to YANG you can see that the modeling language is very straightforward and easy to understand. See RFC 6020 for more details and examples for YANG. The concepts to understand in the above generated skeleton are:

1. The vlan service list will be augmented into the services tree in NSO. This specifies the path to reach vlans in the CLI, REST etc. There are no requirements on where the service shall be added into NSO. If you want vlans to be in the top (in the CLI for example), just remove the augments statement.
2. The two lines of `uses ncs:service-data` and `ncs:servicepoint "vlan"` tells NSO that this is a service. This will be explained more later in the document.

So if a user wants to create a new VLAN in the network what should be the parameters? - A very simple service model would look like below (modify the `src/yang/vlan.yang` file):

```
  augment /ncs:services {
    list vlan {
      key name;

      uses ncs:service-data;
      ncs:servicepoint "vlan";
      leaf name {
        type string;
      }

      leaf vlan-id {
        type uint32 {
```

```
        range "1..4096";
      }
    }

    list device-if {
      key "device-name";
        leaf device-name {
          type leafref {
            path "/ncs:devices/ncs:device/ncs:name";
          }
        }
        leaf interface-type {
          type enumeration {
            enum FastEthernet;
            enum GigabitEthernet;
            enum TenGigabitEthernet;
          }
        }
        leaf interface {
          type string;
        }
    }
  }
}
```

This simple VLAN service model says:

1. We give a VLAN a name, for example net-1, this must also be unique, it is specified as "key".
2. The VLAN has an id from 1 to 4096
3. The VLAN is attached to a list of devices and interfaces. In order to make this example as simple as possible the interface reference is selected by picking the type and then the name as a plain string.

The good thing with NSO is that already at this point you could load the service model to NSO and try if it works well in the CLI etc. Nothing would happen to the devices since we have not defined the mapping, but this is normally the way to iterate a model, test the CLI towards the network engineers.

To build this service model **cd** to `$NCS_DIR/examples.ncs/getting-started/using-ncs/1-simulated-cisco-ios/packages/vlan/src` and type **make** (assuming you have the make build system installed).

```
$ make
```

Go to the root directory of the `simulated-ios` example:

```
$ cd $NCS_DIR/examples.ncs/getting-started/using-ncs/1-simulated-cisco-ios
```

Start netsim, NSO and the CLI:

```
$ncs-netsim start
$ncs --with-package-reload
$ncs_cli -C -u admin
```

When starting NSO above we give NSO a parameter to reload all packages so that our newly added vlan package is included. Packages can also be reloaded without restart. At this point we have a service model for VLANs, but no mapping of VLAN to device configurations. This is fine, we can try the service model and see if it makes sense. Create a VLAN service:

```
admin@ncs(config)# services vlan net-0 vlan-id 1234 \
```

```
                                device-if c0 interface-type FastEthernet interface 1/0
admin@ncs(config-device-if-c0)# top
admin@ncs(config)# show configuration
services vlan net-0
 vlan-id 1234
 device-if c0
  interface-type FastEthernet
  interface      1/0
 !
!
admin@ncs(config)# services vlan net-0 vlan-id 1234 \
                                 device-if c1 interface-type FastEthernet interface 1/0
admin@ncs(config-device-if-c1)# top
admin@ncs(config)# show configuration
services vlan net-0
 vlan-id 1234
 device-if c0
  interface-type FastEthernet
  interface      1/0
 !
 device-if c1
  interface-type FastEthernet
  interface      1/0
 !
!
admin@ncs(config)#
admin@ncs(config)# commit dry-run outformat native
admin@ncs(config)# commit
Commit complete.
admin@ncs(config)# no services vlan
admin@ncs(config)# commit
Commit complete.
```

Committing service changes has no effect on the devices since we have not defined the mapping. The service instance data will just be stored in NSO CDB.

Note that you get tab completion on the devices since they are leafrefs to device names in CDB, the same for interface-type since the types are enumerated in the model. However the interface name is just a string, and you have to type the correct interface-name. For service-models where there is only one device-type like in this simple example, we could have used a reference to the ios interface name according to the IOS model. However that makes the service model dependant on the underlying device types and if another type is added, the service model needs to be updated and this is most often not desired. There are techniques to get tab completion even when the data-type is string, but this is omitted here for simplicity.

Make sure you delete the vlan service instance as above before moving on with the example.

Defining the mapping

Now it is time to define the mapping from service configuration to actual device configuration. The first step is to understand the actual device configuration. Hard-wire the vlan towards a device as an example. This concrete device configuration is a boiler-plate for the mapping, it shows the expected result of applying the service.

```
admin@ncs(config)# devices device c0 config ios:vlan 1234
admin@ncs(config-vlan)# top
admin@ncs(config)# devices device c0 config ios:interface \
                   FastEthernet 10/10 switchport trunk allowed vlan 1234
admin@ncs(config-if)# top
admin@ncs(config)# show configuration
devices device c0
```

```
 config
  ios:vlan 1234
  !
  ios:interface FastEthernet10/10
   switchport trunk allowed vlan 1234
  exit
 !
!
admin@ncs(config)# commit
```

The concrete configuration above has the interface and VLAN hard-wired. This is what we now will make into a template instead. It is always recommended to start like above and create a concrete representation of the configuration the template shall create. Templates are device-configuration where parts of the config is represented as variables. These kind of templates are represented as XML files. Show the above as XML:

```
admin@ncs(config)# show full-configuration devices device c0 \
                                  config ios:vlan | display xml
<config xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
  <device>
    <name>c0</name>
      <config>
      <vlan xmlns="urn:ios">
        <vlan-list>
          <id>1234</id>
        </vlan-list>
      </vlan>
      </config>
  </device>
  </devices>
</config>

admin@ncs(config)# show full-configuration devices device c0 \
                                 config ios:interface FastEthernet 10/10 | display xml
<config xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
  <device>
    <name>c0</name>
      <config>
      <interface xmlns="urn:ios">
      <FastEthernet>
        <name>10/10</name>
        <switchport>
          <trunk>
            <allowed>
              <vlan>
                <vlans>1234</vlans>
              </vlan>
            </allowed>
          </trunk>
        </switchport>
      </FastEthernet>
      </interface>
      </config>
  </device>
  </devices>
</config>
admin@ncs(config)#
```

Now, we shall build that template. When the package was created a skeleton XML file was created in `packages/vlan/templates/vlan.xml`

```
<config-template xmlns="http://tail-f.com/ns/config/1.0"
                 servicepoint="vlan">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <!--
          Select the devices from some data structure in the service
          model. In this skeleton the devices are specified in a leaf-list.
          Select all devices in that leaf-list:
      -->
      <name>{/device}</name>
      <config>
        <!--
            Add device-specific parameters here.
            In this skeleton the service has a leaf "dummy"; use that
            to set something on the device e.g.:
            <ip-address-on-device>{/dummy}</ip-address-on-device>
        -->
      </config>
    </device>
  </devices>
</config-template>
```

We need to specify the right path to the devices. In our case the devices are identified by `/device-if/device-name` (see the YANG service model).

For each of those devices we need to add the VLAN and change the specified interface configuration. Copy the XML config from the CLI and replace with variables:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0"
                 servicepoint="vlan">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>{/device-if/device-name}</name>
      <config>
        <vlan xmlns="urn:ios">
          <vlan-list tags="merge">
            <id>{../vlan-id}</id>
          </vlan-list>
        </vlan>
        <interface xmlns="urn:ios">
          <FastEthernet tags="nocreate" \
             when="{interface-type='FastEthernet'}">
            <name>{interface}</name>
            <switchport>
              <trunk>
                <allowed>
                  <vlan tags="merge">
                    <vlans>{../vlan-id}</vlans>
                  </vlan>
                </allowed>
              </trunk>
            </switchport>
          </FastEthernet>
          <GigabitEthernet tags="nocreate" \
             when="{interface-type='GigabitEthernet'}">
            <name>{interface}</name>
            <switchport>
              <trunk>
                <allowed>
                  <vlan tags="merge">
                    <vlans>{../vlan-id}</vlans>
                  </vlan>
```

```
              </allowed>
            </trunk>
          </switchport>
        </GigabitEthernet>
        <TenGigabitEthernet tags="nocreate" \
           when="{interface-type='TenGigabitEthernet'}">
          <name>{interface}</name>
          <switchport>
            <trunk>
              <allowed>
                <vlan tags="merge">
                  <vlans>{../vlan-id}</vlans>
                </vlan>
              </allowed>
            </trunk>
          </switchport>
        </TenGigabitEthernet>
      </interface>
    </config>
  </device>
 </devices>
</config-template>
```

Walking through the template can give a better idea of how it works. For every `/device-if/device-name` from the service model do the following:

1. Add the vlan to the vlan-list, the tag merge tells the template to merge the data into an existing list (default is replace).
2. Fore every interface within that device, add the vlan to the allowed vlans and set mode to trunk. The tag "nocreate" tells the template to not create the named interface if it does not exist

It is important to understand that every path in the template above refers to paths from the service model in `vlan.yang`.

Request NSO to reload the packages:

```
admin@ncs# packages reload
reload-result {
    package cisco-ios
    result true
}
reload-result {
    package vlan
    result true
}
```

Previously we started ncs with a reload package option, the above shows how to do the same without starting and stopping NSO.

We can now create services that will make things happen in the network. (Delete any dummy service from the previous step first). Create a VLAN service:

```
admin@ncs(config)# services vlan net-0 vlan-id 1234 device-if c0 \
                                interface-type FastEthernet interface 1/0
admin@ncs(config-device-if-c0)# top
admin@ncs(config)# services vlan net-0 device-if c1 \
                                interface-type FastEthernet interface 1/0
admin@ncs(config-device-if-c1)# top
admin@ncs(config)# show configuration
services vlan net-0
```

```
 vlan-id 1234
 device-if c0
  interface-type FastEthernet
  interface      1/0
 !
 device-if c1
  interface-type FastEthernet
  interface      1/0
 !
!
admin@ncs(config)# commit dry-run outformat native
native {
    device {
        name c0
        data interface FastEthernet1/0
              switchport trunk allowed vlan 1234
             exit
    }
    device {
        name c1
        data vlan 1234
             !
             interface FastEthernet1/0
              switchport trunk allowed vlan 1234
             exit
    }
}
admin@ncs(config)# commit
Commit complete.
```

When working with services in templates there is a useful debug option for commit which will show the template and XPATH evaluation.

```
admin@ncs(config)# commit | debug
Possible completions:
 template   Display template debug info
 xpath      Display XPath debug info
admin@ncs(config)# commit | debug template
```

We can change the VLAN service:

```
admin@ncs(config)# services vlan net-0 vlan-id 1222
admin@ncs(config-vlan-net-0)# top
admin@ncs(config)# show configuration
services vlan net-0
 vlan-id 1222
!
admin@ncs(config)# commit dry-run outformat native
native {
    device {
        name c0
        data no vlan 1234
             vlan 1222
             !
             interface FastEthernet1/0
              switchport trunk allowed vlan 1222
             exit
    }
    device {
        name c1
        data no vlan 1234
             vlan 1222
             !
```

```
            interface FastEthernet1/0
             switchport trunk allowed vlan 1222
            exit
    }
}
```

It is important to understand what happens above. When the VLAN id is changed, NSO is able to calculate the minimal required changes to the configuration. The same situation holds true for changing elements in the configuration or even parameters of those elements. In this way NSO does not need to explicit mapping to define a VLAN change or deletion. NSO does not overwrite a new configuration on the old configuration. Adding an interface to the same service works the same:

```
admin@ncs(config)# services vlan net-0 device-if c2 interface-type FastEthernet interface 1/0
admin@ncs(config-device-if-c2)# top
admin@ncs(config)# commit dry-run outformat native
native {
    device {
        name c2
        data vlan 1222
             !
             interface FastEthernet1/0
              switchport trunk allowed vlan 1222
             exit
    }
}
admin@ncs(config)# commit
Commit complete.
```

To clean up the configuration on the devices, run the delete command as shown below:

```
admin@ncs(config)# no services vlan net-0
admin@ncs(config)# commit dry-run outformat native
native {
    device {
        name c0
        data no vlan 1222
             interface FastEthernet1/0
              no switchport trunk allowed vlan 1222
             exit
    }
    device {
        name c1
        data no vlan 1222
             interface FastEthernet1/0
              no switchport trunk allowed vlan 1222
             exit
    }
    device {
        name c2
        data no vlan 1222
             interface FastEthernet1/0
              no switchport trunk allowed vlan 1222
             exit
    }
}
admin@ncs(config)# commit
Commit complete.
```

To make the VLAN service package complete edit the package-meta-data.xml to reflect the service model purpose. This example showed how to use template-based mapping. NSO also allows for programmatic mapping and also a combination of the two approaches. The latter is very flexible, if some logic need to

be attached to the service provisioning that is expressed as templates and the logic applies device agnostic templates.

Reactive FASTMAP

FASTMAP is the NSO algorithm that renders any service change from the single definition of the create service. As seen above, the template or code only has to define how the service shall be created, NSO is then capable of defining *any* change from that single definition.

A limitation in the scenarios described so far is that the mapping definition could immediately do its work as a single atomic transaction. This is sometimes not possible. Typical examples are: external allocation of resource such as IP addresses from an IPAM, spinning up VMs, and sequencing in general.

Reactive FASTMAP addresses these scenarios. The general idea is that the create definition does not only write device configuration but also state data for the service. An external process subscribes to that state data and performs the side-effects. The external process will then call service redeploy which will run the create method again, checking available states like allocated IP address, running VM etc.

The example in `getting-started/developing-with-ncs/4-rfs-service` has a package called `vlan-reactive-fastmap` that implements external allocation of a unit and a vlan-id for the service to illustrate this concept.

Reconciling existing services

A very common situation when we wish to deploy NSO in an existing network is that the network already has existing services implemented in the network. These services may have been deployed manually or through an other provisioning system. The task is to introduce NSO, import the existing services into NSO. The goal is to use NSO to manage existing services, and to add additional instances of the same service type, using NSO. This is a non-trivial problem since existing services may have been introduced in various ways. Even if the service configuration has been done consistently it resembles the challenges of a general solution for rendering a corresponding C-program from assembler.

One of the prerequisites for this to work is that it is possible to construct a list of the already existing services. Maybe such a list exists in an inventory system, an external database, or maybe just an Excel spreadsheet. It may also be the case that we can:

1. Import all managed devices into NSO.
2. Execute a full sync-from on the entire network.
3. Write a program, using Python/Maapi or Java/Maapi that traverses the entire network configuration and computes the services list.

The first thing we must do when we wish to reconcile existing services is to define the service YANG model. The second thing is to implement the service mapping logic, and do it in such a way that given the service input parameters, when we run the service code, they would all result in configuration that is already there in the existing network.

The basic principles for reconciliation is:

1. Read the device configuration to NSO using the `sync-from` action. This will get the device configuration that is a result of any existing services as well.
2. Instantiate the services according to the principles above

Performing the above actions with the default behaviour would not render the correct reference counters since NSO did not actually create the original configuration. The service activation can be run with dedicated flags to take this into account. See the NSO User Guide for a detailed process.

Brown-field networks

In many cases a service activation solution like NSO is deployed in parallell with existing activation solutions. It is then desirable to make sure that NSO does not conflict with the device configuration rendered from the existing solution.

NSO has a commit-flag that will restrict the device configuration to not overwrite data that NSO did not create: `commit no-overwrite`

CHAPTER 6

Compliance Reporting

Overview

When the unexpected happens or things come to the worst and the network configuration is broken, there is a need to gather information and verify the configuration.

NSO has numerous functions to show different aspects of such a network config verification. However, to simplify this task, the compliance reporting can assemble information using a selection of these NSO functions and present the resulting information in one report. The aim for this report is to answer two fundamental questions:

- Who has done what?
- Is the network correctly configured?

What defines then a correctly configured network? Where is the master? NSO by nature, with the configurations stored in CDB, is the master. Checking the live devices against the NSO stored device configuration is a fundamental part of compliance reporting. But it does not stop here. Compliance reporting can also be based on one or a number of stored device templates which the live devices are compared against. The compliance reports can also be a combination of both approaches.

Compliance verification can be defined to check the current situation or checking historic events, or both. To assemble historic events, rollback files and audit logs are used. Therefore these functionalities must have been enabled for the time period of interest, or else no history view can be presented.

The reports can be formatted either as text, html or docbook xml format. The intention of the docbook format is that the user can take the report and by further post-processing create reports formatted by own choice, for instance PDF using Apache FOP.

Creating compliance report definitions

It is possible to create several named compliance report definitions. Each named report defines which devices, services and/or templates that the configuration should be checked for.

The CLI command below shows a very simple definition (if you start the `examples.ncs/service-provider/mpls-vpn` you will see this example):

```
ncs(config)# show full-configuration compliance
compliance reports report Compliance-Audit
 device-check all-devices
 device-check current-out-of-sync true
 device-check historic-changes true
 device-check historic-out-of-sync true
 service-check all-services
 service-check current-out-of-sync true
 service-check historic-changes true
 service-check historic-out-of-sync true
 compare-template snmp1 P
  variable COMMUNITY
   value 'public'
  !
 !
 compare-template snmp1 PE
  variable COMMUNITY
   value 'public'
  !
 !
!
```

This definition says that the "Compliance-Audit" audit will do a check-sync for all devices and services when the report is executed. It also includes historical entries from the NSO audit log indicating that the same devices and services have been out of sync.

The report also specifies that the template named `snmp1` shall be compared with all devices in the device-group called `P` using the template variable value `"COMMUNITY=public"`.

Running reports

A named report can be executed and takes the following parameters when run:

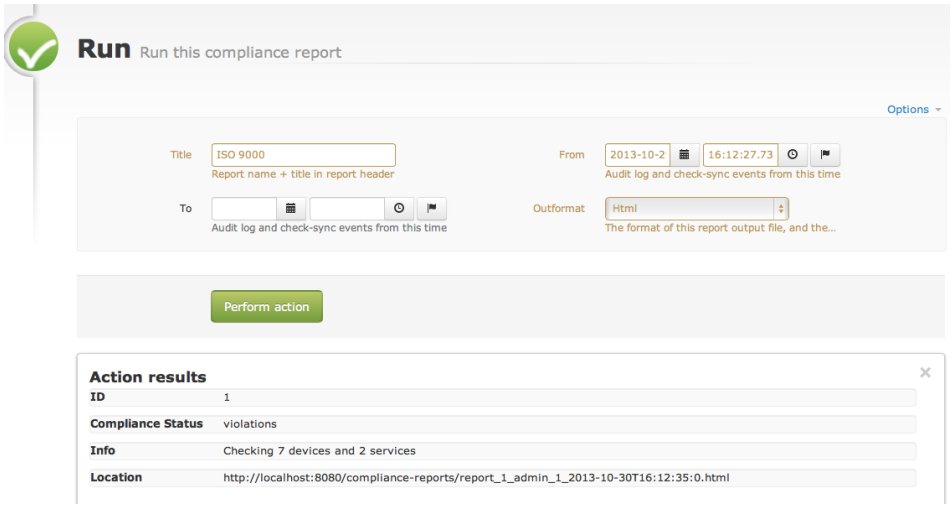
1. Title : a title for this saved report.
2. From - To : time window for entries from audit log, for example failed check-sync.

So the report defined above, `daily-audit` can be executed with:

```
ncs(config)# compliance reports report Compliance-Audit run
       from 2014-09-01T00:00:00 title "ISO check" outformat html
id 1
compliance-status violations
info Checking 17 devices and 2 services
location http://localhost:8080/compliance-reports/report_1_admin_1_2014-9-14T15:10:21:0.html
```

Or run from Web UI:

Figure 6. Running a compliance report



As seen above the report found compliance violations.

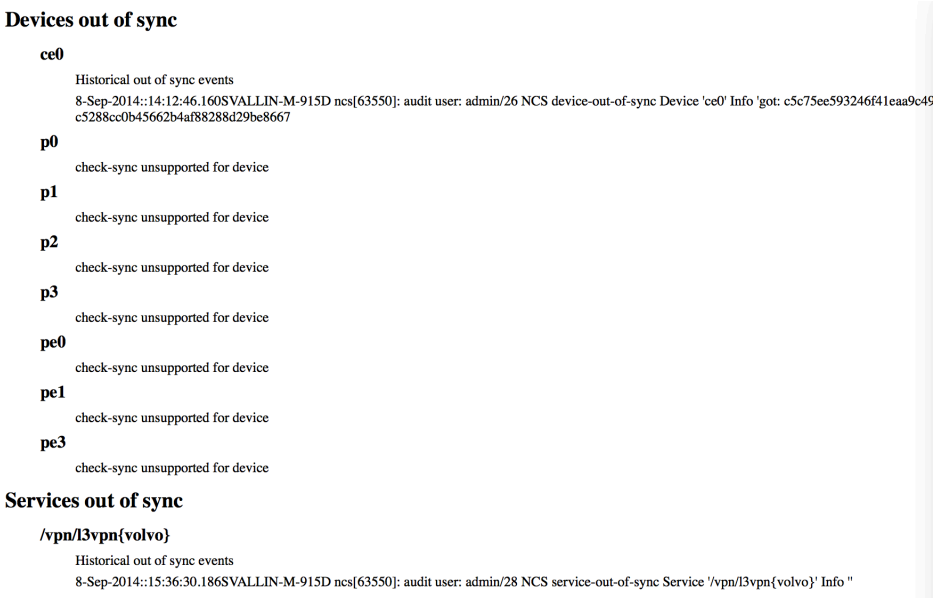
The report

The contents of the report is:

1. Devices and services out of sync : reports on devices out of sync when the report was executed as well as historical out of sync events from the audit log, (based on From - To input parameters). For services and devices out of sync when the report is run, the actual diff is shown.
2. Template discrepancies : reports if any devices are not in sync with the templates as defined by the report. The report also shows how they actually differ.
3. Commit history : the report shows a list of commits that have been performed and commit details for those devices and services that are defined in the report.

Some examples are given below. First we see a report snippet that illustrates that NSO has detected that the device ce0 and the volvo VPN has been out of sync:

Figure 7. Service out of sync, and template discrepancies



Devices out of sync

ce0

Historical out of sync events

8-Sep-2014::14:12:46.160SVALLIN-M-915D ncs[63550]: audit user: admin/26 NCS device-out-of-sync Device 'ce0' Info 'got: c5c75ee593246f41eaa9c49
c5288cc0b45662b4af88288d29be8667

p0

check-sync unsupported for device

p1

check-sync unsupported for device

p2

check-sync unsupported for device

p3

check-sync unsupported for device

pe0

check-sync unsupported for device

pe1

check-sync unsupported for device

pe3

check-sync unsupported for device

Services out of sync

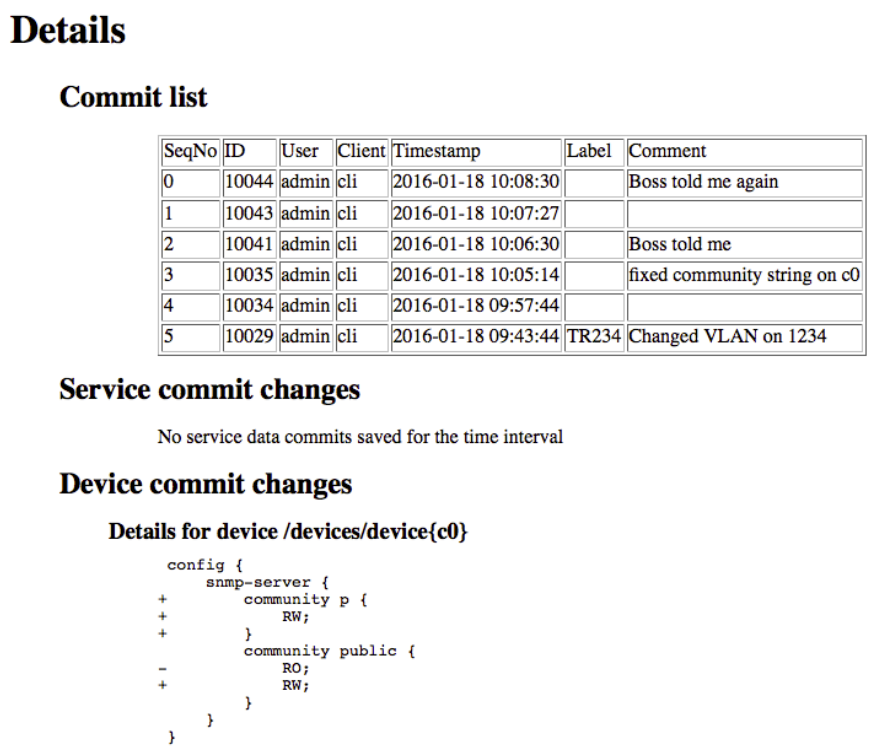
/vpn/l3vpn{volvo}

Historical out of sync events

8-Sep-2014::15:36:30.186SVALLIN-M-915D ncs[63550]: audit user: admin/28 NCS service-out-of-sync Service '/vpn/l3vpn{volvo}' Info ''

The other example shows a summary of all commits. Note that a habit of using labels and commits makes the reports more useful. Also a summary of every commit for the specified devices and services are included in the report.

Figure 8. Commit history



Details

Commit list

SeqNo	ID	User	Client	Timestamp	Label	Comment
0	10044	admin	cli	2016-01-18 10:08:30		Boss told me again
1	10043	admin	cli	2016-01-18 10:07:27		
2	10041	admin	cli	2016-01-18 10:06:30		Boss told me
3	10035	admin	cli	2016-01-18 10:05:14		fixed community string on c0
4	10034	admin	cli	2016-01-18 09:57:44		
5	10029	admin	cli	2016-01-18 09:43:44	TR234	Changed VLAN on 1234

Service commit changes

No service data commits saved for the time interval

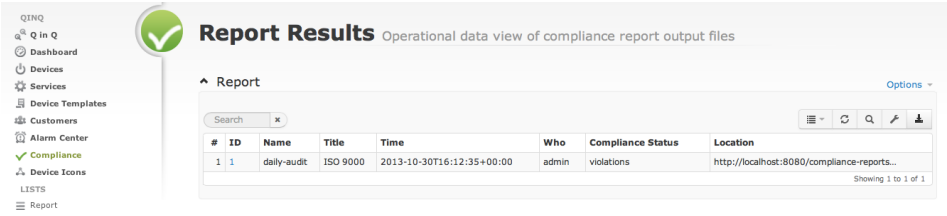
Device commit changes

Details for device /devices/device{c0}

```
 config {
     snmp-server {
+        community p {
+            RW;
+        }
         community public {
-            RO;
+            RW;
         }
     }
 }
```

All report runs are saved in a table and the report itself on disc.

Figure 9. Compliance report runs



CHAPTER 7

Administration

Administration Overview

Installing NSO

See the `NSO Installation Guide` for more information. There are two kinds of NSO installations:

1. Local installation: used for evaluation and development purposes
2. System installation: used for system-wide and deployment scenarios.

This guide is intended for evaluation and development scenarios so local install is briefly covered here. Local Install of NSO is performed in a single, user specified directory. For example in your home directory `$HOME`. The user has to stop and start NSO manually when using local install.

Pre-Installation

- Before installing NSO, ensure that Java JDK-6.0 or higher is installed.
- This type of installation is supported both on Linux OS and Darwin OS for x86_64 and i686 architectures.

Installation

Step 1 Install the NSO Software in a local directory, for example in home directory $HOME. It is recommended to always install NSO in a directory named as the version of the release.

--local-install parameter can be used and it is an optional parameter.

```
sh nso-VERSION.OS.ARCH.installer.bin $HOME/ncs-VERSION
```

The variables in the command 'VERSION' means ncs-version to install. 'OS' means the Operating System (Linux/Darwin). 'ARCH' means the architecture that supports (x86_64 or i686).

```
Example: sh nso-4.0.linux.x86_64.installer.bin $HOME/nso-4.0
```

Step 2 The installation program creates a shell script file named `ncsrc` in each NSO installation, which sets the environment variables. Source this file to get these settings in your shell. You may want to add this sourcing command to your login sequence, such as `'.bashrc'`.

For csh/tcsh users there is a `ncsrc.tcsh` file using csh/tcsh syntax. The example below assume that you are using bash, other versions of /bin/sh may require that you use '.' instead of 'source'.

```
source $HOME/ncs-VERSION/ncsrc
```

Step 3 Create a runtime or deployment directory where NSO will keep its database, state files,logs etc. In these instructions we will assume that this directory is `$HOME/ncs-run`.

```
ncs-setup --dest $HOME/ncs-run
```

There are a set of examples available in the installation directory *$NCS_DIR/examples.ncs* Please go through the examples for information on how to create run-time directories, start ncs, and other important NSO functionalities.

Step 4 Finally start NSO.

```
$ cd $HOME/ncs-run
$ ncs
```

When you start NSO, make sure that you are located in the deployment directory since NSO will look for its runtime directories and configuration file in ./ folder.

NSO UnInstallation

NSO can be uninstalled very easily. A single delete of the directory where NSO was installed is sufficient.

```
rm -rf $HOME/ncs-VERSION
```

Running NSO

NSO needs a deployment/runtime directory where the database files, logs etc are stored. An "empty" default such directory can be created using the **ncs-setup** command:

```
$ ncs-setup --dest /home/ncs
```

In this User Guide we will refer to examples in `$NCS_DIR/examples.ncs`. All of them have ready-made runtime directories. So the above step is not needed for running the examples.

When you start NSO, make sure you are located in the deployment dir since NSO will look for its runtime directories and configuration file in `./`. In the same way as you can have several parallel NSO installations you can have as many deployment directories you like. All of the with different packages, configuration files, and database contents. This is exactly how the examples are structured, every NSO example is a self-contained directory.

Note A common misunderstanding is the there is a dependency between the runtime directory and the installation directory. This is not true. For example lets say you have two NSO installations `.../ncs-2.3.1` and `.../ncs-2.3.2`. So in the below sequence will run ncs-2.3.1 but using an example and configuration from ncs-2.3.2.

```
$ cd .../ncs-2.3.1
$ . ncsrc
$ cd .../ncs-2.3.2/examples.ncs/datacenter-qinq
$ ncs
```

Since the runtime directory is self-contained, this is also the way can move between examples:

```
$ cd $NCS_DIR/examples.ncs/data-center-qinq
$ ncs
$ ncs --stop
$ cd $NCS_DIR/examples.ncs/getting-started/1-simulated-cisco-ios
$ ncs
$ ncs --stop
```

Since the run-time directory is self-contained including also the database files. You can just compress a complete directory and distribute it. Unpacking that directory and starting NSO from there will give an exact copy of all instance data.

Starting and stopping

Whenever we start (or reload) the NSO daemon it reads its configuration from `./ncs.conf` or `${NCS_DIR}/etc/ncs/ncs.conf` or from the file specified with the **-c** option.

```
$ ncs
$ ncs --stop
$ ncs -h
...
```

The command **ncs -h** shows various options when starting NSO. By default, NSO starts in the background without an associated terminal. It is recommended to add NSO to the `/etc/init` scripts of the deployment hosts. See more information in the Reference Guide and **man ncs**.

User Management

Users are configured at the path **aaa authentication users**

```
admin@ncs(config)# show full-configuration aaa authentication users user
aaa authentication users user admin
 uid        1000
 gid        1000
 password   $1$GNwimSPV$E82za8AaDxukAi8Ya8eSR.
 ssh_keydir /var/ncs/homes/admin/.ssh
 homedir    /var/ncs/homes/admin
!
aaa authentication users user oper
 uid        1000
 gid        1000
 password   $1$yOstEhXy$nYKOQgslCPyv9metoQALA.
 ssh_keydir /var/ncs/homes/oper/.ssh
 homedir    /var/ncs/homes/oper
!...
```

Access control, including group memberships, is managed using the NACM model (RFC 6536).

```
admin@ncs(config)# show full-configuration nacm
nacm write-default permit
nacm groups group admin
 user-name [ admin private ]
!
nacm groups group oper
 user-name [ oper public ]
!
nacm rule-list admin
 group [ admin ]
 rule any-access
  action permit
 !
!
nacm rule-list any-group
 group [ * ]
 rule tailf-aaa-authentication
  module-name       tailf-aaa
  path              /aaa/authentication/users/user[name='$USER']
  access-operations read,update
  action            permit
 !
```

So, adding a user includes the following steps:

Step 1 Create the user: **admin@ncs(config)# aaa authentication users user <user-name>**

Step 2 Add the user to a NACM group: **admin@ncs(config)# nacm groups <group-name> admin user-name <user-name>**

Step 3 Verify/change access rules.

It is likely that the new user also needs access to work with device configuration. The mapping from NSO users and corresponding device authentication is configured in `authgroups`.

```
admin@ncs(config)# show full-configuration devices authgroups
devices authgroups group default
 umap admin
  remote-name     admin
  remote-password $4$wIo7Yd068FRwhYYI0d4IDw==
 !
 umap oper
  remote-name     oper
  remote-password $4$zp4zerM68FRwhYYI0d4IDw==
 !
!
```

So the user needs to be added here as well. If the last step is forgotten you will see the following error:

```
jim@ncs(config)# devices device c0 config ios:snmp-server community fee
jim@ncs(config-config)# commit
Aborted: Resource authgroup for jim doesn't exist
```

Packages

NSO Packages contain data-models and code for a specific function. It might be a NED for a specific device, a service application like MPLS VPN, a WebUI customization package etc. Packages can be added, removed and upgrade in run-time. A common task is to add a package to NSO in order to support a new device-type, or upgrade an existing package when the device is upgraded.

(We assume you have the example up and running from previous section). Current installed packages can be viewed with the following command:

```
admin@ncs# show packages
packages package cisco-ios
 package-version 3.0
 description     "NED package for Cisco IOS"
 ncs-min-version [ 3.0.2 ]
 directory       ./state/packages-in-use/1/cisco-ios
 component upgrade-ned-id
  upgrade java-class-name com.tailf.packages.ned.ios.UpgradeNedId
 component cisco-ios
  ned cli ned-id  cisco-ios
  ned cli java-class-name com.tailf.packages.ned.ios.IOSNedCli
  ned device vendor Cisco
NAME      VALUE
--------------------
show-tag  interface

 oper-status up
```

So the above command shows that NSO currently have one package, the NED for Cisco IOS.

NSO reads global configuration parameters from `ncs.conf`. More on NSO configuration later in this guide. By default it tells NSO to look for packages in a `packages` directory where NSO was started. So in this specific example:

```
$ pwd
.../examples.ncs/getting-started/using-ncs/1-simulated-cisco-ios
$ ls packages/
cisco-ios
$ ls packages/cisco-ios
doc
load-dir
netsim
package-meta-data.xml
private-jar
shared-jar
src
```

As seen above a package is a defined file structure with data-models, code and documentation. NSO comes with a couple of ready-made packages: `$NCS_DIR/packages/`. Also there is a library of packages available from Tail-f especially for supporting specific devices.

Adding and upgrading a package

Assume you would like to add support for Nexus devices into the example. Nexus devices have different data-models and another CLI flavor. There is a package for that in `$NCS_DIR/packages/neds/nexus`.

We can keep NSO running all the time, but we will stop the network simulator to add the nexus devices to the simulator.

```
$ ncs-netsim stop
```

Add the nexus package to the NSO runtime directory by creating a symbolic link:

```
$ cd $NCS_DIR/examples.ncs/getting-started/using-ncs/1-simulated-cisco-ios/packages
$ ln -s $NCS_DIR/packages/neds/cisco-nx
$ ls -l
```

```
... cisco-nx -> .../packages/neds/cisco-nx
```

The package is now in place, but until we tell NSO for look for package changes nothing happens:

```
  admin@ncs# show packages packages package
  cisco-ios ...  admin@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-nx
    result true
}
```

So after the packages reload operation NSO also knows about nexus devices. The reload operation also takes any changes to existing packages into account. The datastore is automatically upgraded to cater for any changes like added attributes to existing configuration data.

Simulating the new device

```
$ ncs-netsim add-to-network cisco-nx 2 n
$ ncs-netsim list
ncs-netsim list for  /Users/stefan/work/ncs-3.2.1/examples.ncs/getting-started/using-ncs/1-simul

name=c0 ...
name=c1 ...
name=c2 ...
name=n0 ...
name=n1 ...


$ ncs-netsim start
DEVICE c0 OK STARTED
DEVICE c1 OK STARTED
DEVICE c2 OK STARTED
DEVICE n0 OK STARTED
DEVICE n1 OK STARTED
$ ncs-netsim cli-c n0
n0#show running-config
no feature ssh
no feature telnet
fex 101
 pinning max-links 1
!
fex 102
 pinning max-links 1
!
nexus:vlan 1
!
...
```

Adding the new devices to NCS

We can now add these Nexus devices to NSO according to the below sequence:

```
admin@ncs(config)# devices device n0 device-type cli ned-id cisco-nx
admin@ncs(config-device-n0)# port 10025
admin@ncs(config-device-n0)# address 127.0.0.1
admin@ncs(config-device-n0)# authgroup default
admin@ncs(config-device-n0)# state admin-state unlocked
admin@ncs(config-device-n0)# commit
admin@ncs(config-device-n0)# top
admin@ncs(config)# devices device n0 sync-from
result true
```

Configuring NSO

ncs.conf

The configuration file `ncs.conf` is read at startup and can be reloaded. Below follows an example with the most common settings. It is included here as an example and should be self-explanatory. See **man ncs.conf** for more information. Important configuration settings:

- load-path: where NSO should look for compiled YANG files, such as data-models for NEDs or Services.
- db-dir: the directory on disk which CDB use for its storage and any temporary files being used. It is also the directory where CDB searches for initialization files. This should be local disc and not NFS mounted for performance reasons.
- Various log settings
- AAA configuration
- Rollback file directory and history length.
- Enabling north-bound interfaces like REST, WebUI
- Enabling of High-Availability mode

Run-time configuration

There are also configuration parameters that are more related to how NCS behaves when talking to the devices. These resides in **devices global-settings**.

```
admin@ncs(config)# devices global-settings
Possible completions:
  backlog-auto-run               Auto-run the backlog at successful connection
  backlog-enabled                Backlog requests to non-responding devices
  commit-queue
  commit-retries                 Retry commits on transient errors
  connect-timeout                Timeout in seconds for new connections
  ned-settings                   Control which device capabilities NCS uses
  out-of-sync-commit-behaviour   Specifies the behaviour of a commit operation involving a dev
  read-timeout                   Timeout in seconds used when reading data
  report-multiple-errors         By default, when the NCS device manager commits data southbou
                                 report the first error to the operator, this flag makes NCS
                                 devices
  trace                          Trace the southbound communication to devices
  trace-dir                      The directory where trace files are stored
  write-timeout                  Timeout in seconds used when writing
  data
```

Monitoring NSO

Use the command **ncs --status** to get runtime information on NSO.

Backup and Restore

All parts of the NSO installation, can be backed up and restored with standard file system backup procedures.

The most convenient way to do backup and restore is to use the ncs-backup command (this only works if NSO is installed using the `--system-install` option). In that case the following procedure is used.

Backup

NSO Backup backs up the database (CDB) files, state files, config files and rollback files from the installation directory.

- To take a complete backup (for disaster recovery), use

  ```
  ncs-backup
  ```

  The backup will be stored in Run Directory `/var/opt/ncs/backups/ncs-VERSION@DATETIME.backup`

  For more information on backup, refer to the ncs-backup(1) in *NSO 4.4.2.3 Manual Pages* man page.

NSO Restore

NSO Restore is performed if you would like to switch back to a previous good state or restore a backup.

Note It is always advisable to stop NSO before performing Restore.

- First stop NSO if NSO is not stopped yet.

  ```
  /etc/init.d/ncs stop
  ```

  Then take the backup

  ```
  ncs-backup --restore
  ```

  Select the backup to be restored from the available list of backups. The configuration and database with run-time state files are restored in `/etc/ncs` and `/var/opt/ncs`.
- Start NSO.

  ```
  /etc/init.d/ncs start
  ```