User Guide

Release: NSO 6.2.5

Published: May 17, 2010

Last Modified: May 14, 2024

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

About

Purpose

This guide helps you develop a practical understanding of NSO by exploring areas of operational importance, such as the CLI/Web interface and network management operations. Use this guide to familiarize yourself with topics like NSO's basic operations, Network Element Drivers (NEDs), device management, NSO services, life cycle operations, etc.

Target Audience

The guide is intended for network engineers and users who want to use and operate NSO.

CHAPTER 2

Basic Operations

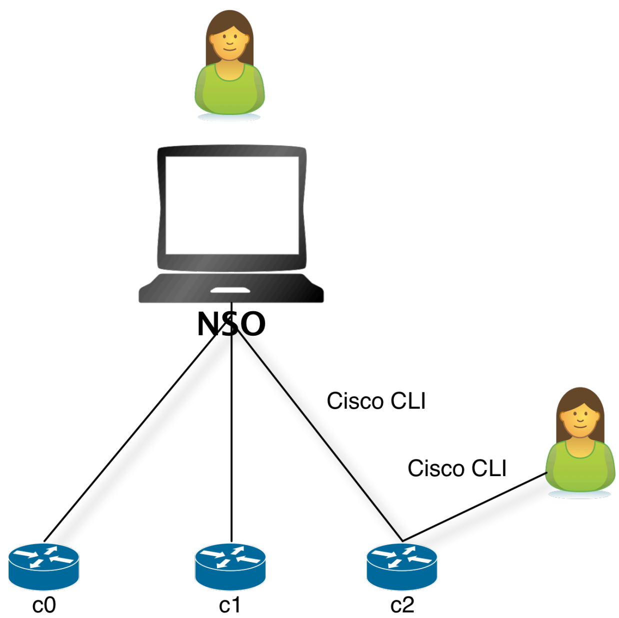
The purpose of this section is to get started with NSO, learn the basic operational scenarios and get acquainted with the most common CLI commands.

Setup

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

We will use the NSO network simulator to simulate three 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 1. 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:

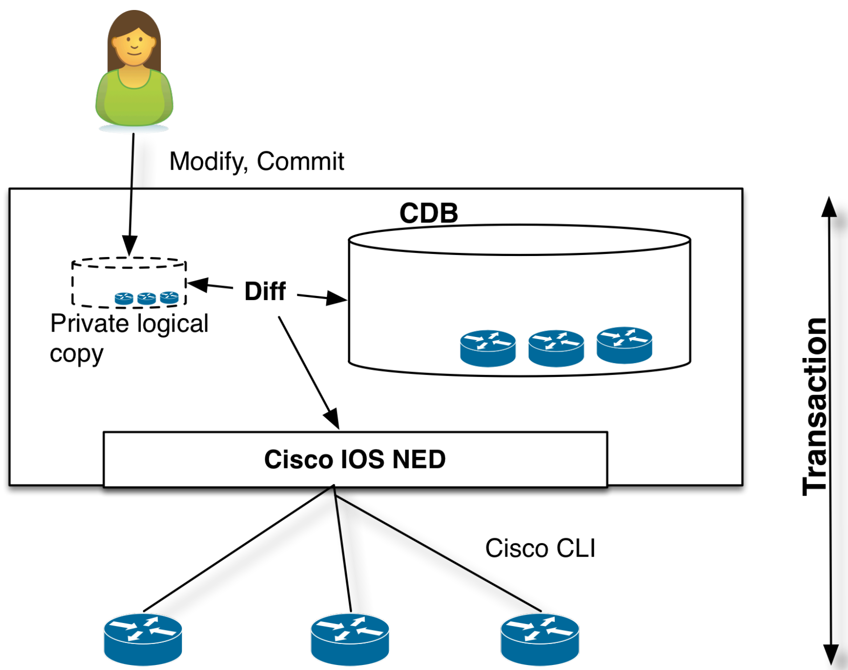
1. At initialization NSO can discover the configuration as shown above.
2. The modus operandi when using NSO to perform configuration changes is that the network engineer uses NSO (CLI, WebUI, REST,...) to modify the representation in NSO CDB. The changes are committed to the network as a transaction 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 2. 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-files apply-rollback-file fixed-number 10005
```

apply-rollback-file by default restores to that saved configuration, adding selective as parameter allows you to just rollback 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. In addition there is the possibility to append the **reverse** flag that will display the device commands for getting back to the current running state in the network if the commit is successfully executed. 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
                                ned-id cisco-ios-cli-3.0
                                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 package comes with a YANG data-model for the device (except for third party YANG NED for which YANG device model must be downloaded and fixed before it can be used). 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
                                ned-id cisco-ios-cli-3.0
                                config 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 do 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.

Also note that device templates have some additional restrictions on the values that can be supplied when applying the template. In particular, a value must either be a number or a single-quoted string. It is currently not possible to specify a value that contains a single quote (').

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". The commit queue should primarily be used for throughput reasons when doing configuration changes in large networks. Atomic transactions comes with a cost, the critical section of the database is locked when committing the transaction on the network. So, in cases where there are northbound systems of NSO that generates many simultaneous large configuration changes these might get queued. The commit queue will send the device commands after the lock has been released, so the database 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 | notab
devices commit-queue queue-item 2236633674
 age              11
 status           executing
 kilo-bytes-size  1
 devices          [ c0 c1 c2 ]
 transient-errors [ c0 ]
 is-atomic        true
```

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

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

$ncs_cli -C -u admin

admin@ncs# show devices commit-queue
devices commit-queue queue-item 2236633674
 age              11
 status           executing
 kilo-bytes-size  1
 devices          [ c0 c1 c2 ]
 transient-errors [ c0 ]
 is-atomic        true

admin@ncs# show devices commit-queue
devices commit-queue queue-item 2236633674
 age              11
 status           executing
 kilo-bytes-size  1
 devices          [ c0 c1 c2 ]
 is-atomic        true

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.

About Conflicts

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 an ordered-by user list in such way that one user rearranges the list, along with other non-conflicting modifications, and one user deletes the entire list the following happens:

```
admin@ncs(config)# no devices device c0 config access-list 10

      joe@ncs(config)# move devices device c0 config access-list 10 permit 168.215.202.0 0.0.0.2
      joe@ncs(config)# devices device c0 config logging history informational
      joe@ncs(config)# devices device c0 config logging source-interface Vlan512
      joe@ncs(config)# devices device c0 config logging 10.1.22.122
      joe@ncs(config)# devices device c0 config logging 66.162.108.21
      joe@ncs(config)# devices device c0 config logging 50.58.29.21

admin@ncs% commit
```

```
System message at 2022-09-01 14:17:59...
Commit performed by admin via console using cli.
joe@ncs(config-config)# commit
Aborted: Transaction 542 conflicts with transaction 562 started by user admin: 'devices
--------------------------------------------------------------------------
This transaction is in a non-resolvable state.
To attempt to reapply the configuration changes made in the CLI,
in a new transaction, revert the current transaction by running
the command 'revert' followed by the command 'reapply-commands'.
--------------------------------------------------------------------------
```

In this case joe commits a change to `access-list` after admin and a conflict message is displayed. Since the conflict is non-resolvable the transaction has to be reverted. In order to reapply the changes made by joe to `logging` in a new transaction, the following commands are entered:

```
joe@ncs(config)# revert no-confirm
joe@ncs(config)# reapply-commands best-effort
move devices device c0 config access-list 10 permit 168.215.202.0 0.0.0.255 first
Error: on line 1: move devices device c0 config access-list 10 permit 168.215.202.0 0.0
devices device c0 config
logging history informational
logging facility local0
logging source-interface Vlan512
logging 10.1.22.122
logging 66.162.108.21
logging 50.58.29.21
joe@ncs(config-config)# show config
logging facility local0
logging history informational
logging 10.1.22.122
logging 50.58.29.21
logging 66.162.108.21
logging source-interface Vlan512
joe@ncs(config-config)# commit
Commit complete.
```

In this case joe tries to reapply the changes made in the previous transaction and since `access-list 10` has been removed, the **`move`** command will fail when applied by the **`reapply-commands`** command. Since the mode is **`best-effort`** the next command will be processed. The changes to `logging` will succeed and joe then commits the transaction.

CHAPTER 3

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” in *Administration Guide*.

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. *A driver element*: running in a Java VM.
2. *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.
3. *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-cli-3.0
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 an SNMP device assuming that 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
```

Live status protocol

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.

Multi NEDs for Statistics

Sometimes we wish to use a different protocol to collect statistics from the live tree than the protocol that is used to configure a managed device. There are many interesting use cases where this pattern applies. For example, if we wish to access SNMP data as statistics in the live tree on a Juniper router, or alternatively if we have a CLI NED to a Cisco type device, and wish to access statistics in the live tree over SNMP.

The solution is to configure additional protocols for the live tree. We can have an arbitrary number of NEDs associated to statistics data for an individual managed device.

The additional NEDs are configured under `/devices/device/live-status-protocol`.

In the configuration snippet below, we have configured two additional NEDs for statistics data.

```
devices {
    authgroups {
        snmp-group g1 {
            umap admin {
                community-name public;
            }
        }
    }
```

```
    mib-group m1 {
        mib-module [ SIMPLE-MIB ];
    }
    device device0 {
        live-status-protocol x1 {
            port 4001;
            device-type {
                snmp {
                    version        v2c;
                    snmp-authgroup g1;
                    mib-group      [ m1 ];
                }
            }
        }
        live-status-protocol x2 {
            authgroup default;
            device-type {
                cli {
                    ned-id xstats;
                }
            }
        }
     }
```

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 Cisco support.

CHAPTER 4

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.

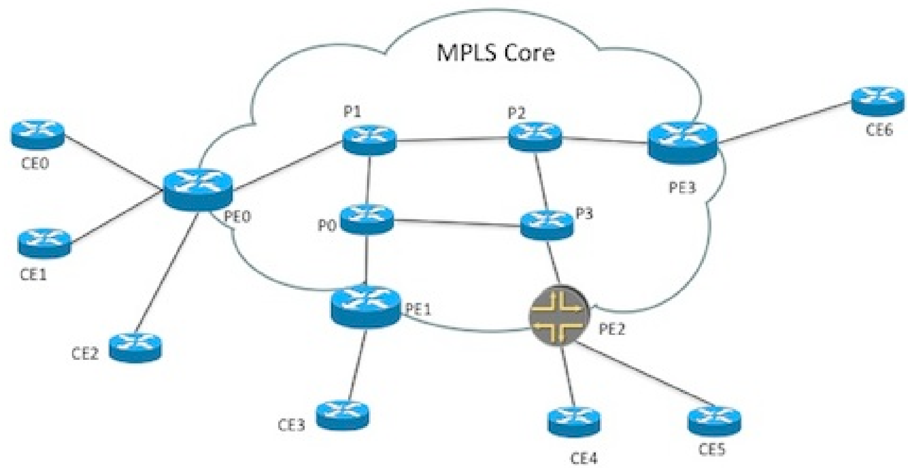
A Service Example

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 You can find a video presentation of this demo on YouTube.

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 3. 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
!
```

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 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.

```
 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 authoritative:

```
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.

Using Commit queues

As described in detail in the Device Manager the section called “Commit Queue” section the commit queue can be used to increase the transaction throughput. When the commit queue are for service activation the services will have states reflecting outstanding commit queue items.

Note When committing a service using the commit queue in *async* mode the northbound system can not rely on the service being fully activated in the network when the activation requests returns.

We will now commit a vpn service using the commit queue and one device is down.

```
$ ncs-netsim stop ce0
DEVICE ce0 STOPPED

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 main-office
  ce-device    ce0
  ce-interface GigabitEthernet0/11
  ip-network   10.10.1.0/24
  bandwidth    12000000
 !
!

ncs# commit commit-queue async
commit-queue-id 10777927137
Commit complete.
ncs(config)# *** ALARM connection-failure: Failed to connect to device ce0: connection refuse
```

This service is not provisioned fully in the network, since **ce0** was down. It will stay in the queue either until the device starts responding or that an action is taken to remove the service or remove the item. The commit queue can be inspected. As shown below we see that we are waiting for **ce0**. Inspecting the queue item shows the outstanding configuration.

```
ncs# show devices commit-queue | notab
devices commit-queue queue-item 10777927137
 age              1934
 status           executing
 kilo-bytes-size  2
 devices          [ ce0 ce1 pe0 ]
 transient-errors [ ce0 ]
 is-atomic        true

ncs# show vpn l3vpn volvo commit-queue | notab
commit-queue queue-item 1498812003922
```

The commit queue will constantly try to push the configuration towards the devices. The number of retry attempts and at what interval they occur can be configured.

```
ncs# show full-configuration devices global-settings commit-queue | details
devices global-settings commit-queue enabled-by-default false
devices global-settings commit-queue atomic true
devices global-settings commit-queue retry-timeout 30
devices global-settings commit-queue retry-attempts unlimited
```

If we start **ce0** and inspect the queue we will see that the queue will finally be empty, and that the commit-queue status for the service is empty.

```
ncs# show devices commit-queue | notab
devices commit-queue queue-item 10777927137
 age              3357
 status           executing
 kilo-bytes-size  2
 devices          [ ce0 ce1 pe0 ]
 transient-errors [ ce0 ]
 is-atomic        true

ncs# show devices commit-queue | notab
devices commit-queue queue-item 10777927137
 age              3359
 status           executing
 kilo-bytes-size  2
 devices          [ ce0 ce1 pe0 ]
 is-atomic        true

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

ncs# show vpn l3vpn volvo commit-queue
% No entries found.

ncs# show devices commit-queue completed | notab
devices commit-queue completed queue-item 10777927137
 when               2015-02-09T16:48:17.915+00:00
 succeeded          true
 devices            [ ce0 ce1 pe0 ]
 completed          [ ce0 ce1 pe0 ]
 completed-services [ /l3vpn:vpn/l3vpn:l3vpn[l3vpn:name='volvo'] ]
```

Un-deploying Services

In some scenarios it makes sense to remove the service configuration from the network but keep the representation of the service in NSO. This is called to **un-deploy** a service.

```
ncs# vpn l3vpn volvo check-sync
in-sync false
ncs# vpn l3vpn volvo re-deploy
ncs# vpn l3vpn volvo check-sync
in-sync true
```

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 *Administration Guide*). Create a directory under, for example, `~/my-sim-ios`, similar to how it is done for the `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
```

Note If the `packages` folder does not exist yet, such as when you have not run this example before, you will need to invoke the **ncs-setup** and **ncs-netsim create-network** commands as described in the `1-simulated-cisco-ios` README file.

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

```
$ ncs-make-package --service-skeleton template --root-container vlans --no-test  vlan
```

This results in a directory structure:

```
vlan
   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;
  }

  container vlans {
  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;
                }
              }
            } // container vlans {
          }
```

If this is your first exposure to YANG you can see that the modeling language is very straightforward and easy to understand. See RFC 7950 for more details and examples for YANG. The concept to understand in the above generated skeleton is that the two lines of `uses ncs:service-data` and `ncs:servicepoint "vlan"` tells NSO that this is a service. The `ncs:service-data` grouping together with the `ncs:servicepoint` YANG extension provide the common definitions for a service. The two are implemented by the `$NCS_DIR/src/ncs/yang/tailf-ncs-services.yang`. 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 {
    container vlans {
      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)# commit dry-run outformat cli
cli {
    local-node {
        data  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
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 dependent 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">
          <?if {interface-type='FastEthernet'}?>
            <FastEthernet tags="nocreate">
              <name>{interface}</name>
              <switchport>
                <trunk>
                  <allowed>
                    <vlan tags="merge">
                      <vlans>{../vlan-id}</vlans>
                    </vlan>
                  </allowed>
                </trunk>
              </switchport>
            </FastEthernet>
          <?end?>
          <?if {interface-type='GigabitEthernet'}?>
            <GigabitEthernet tags="nocreate">
              <name>{interface}</name>
              <switchport>
                <trunk>
                  <allowed>
                    <vlan tags="merge">
                      <vlans>{../vlan-id}</vlans>
                    </vlan>
                  </allowed>
                </trunk>
              </switchport>
            </GigabitEthernet>
          <?end?>
          <?if {interface-type='TenGigabitEthernet'}?>
            <TenGigabitEthernet tags="nocreate">
              <name>{interface}</name>
              <switchport>
                <trunk>
                  <allowed>
```

```
                    <vlan tags="merge">
                      <vlans>{../vlan-id}</vlans>
                    </vlan>
                  </allowed>
                </trunk>
              </switchport>
            </TenGigabitEthernet>
          <?end?>
        </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 and Nano Services

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.

Nano services using Reactive FASTMAP addresses these scenarios with an executable plan that the system can follow to provision the service. The general idea is to implement the service as several smaller (nano) steps or stages, by using reactive FASTMAP and provide a framework to safely execute actions with side effects.

The example in `examples.ncs/development-guide/nano-services/netsim-sshkey` implements key generation to files and service deployment of the key to set up network elements and NSO for public key authentication to illustrate this concept. The example is described in more detail in Chapter 5, *Developing and Deploying a Nano Service* in *Getting Started*.

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 parallel 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`

Advanced Services Orchestration

Some services need to be set up in stages where each stage can consist of setting up some device configuration and then wait for this configuration to take effect before performing next stage. In this scenario each stage must be performed in a separate transaction which is committed separately. Most often an external notification or other event must be detected and trigger the next stage in the service activation.

NSO supports the implementation of such staged services with the use of Reactive FASTMAP patterns in nano services.

From the user perspective it is not important how a certain service is implemented. And the implementation should not have an impact on how the user creates or modifies a service. However the knowledge about this can be necessary to explain the behavior of a certain service.

In short the life-cycle of a RFM nano service in not only controlled by the direct create/set/delete operations. Instead there are one or many implicit `reactive-re-deploy` requests on the service that are triggered from external event detection. If the user examines a RFM service, e.g. using `get-modification`, the device impact will grow over time after the initial create.

Nano Service Plans

Nano services autonomously will do `reactive-re-deploy` until all "stages" of the service are completed. This implies that an nano service normally is not completed when the initial create is committed. For the operator to understand that a nano service has run to completion there must typically be some service specific operational data that can indicate this.

Plans are introduced to standardize the operational data that can show the progress of nano service. This gives the user a standardized view of all nano services and can directly answer the question weather a service instance has run to completion or not.

A *plan* consists of one or many *component* entries. Each *component* consists of two or many *state* entries where the state can be in status *not-reached*, *reached* or *failed*. A *plan* must have a component named *"self"* and can have other components with arbitrary names that have meaning for the implementing nano service. A *plan component* must have a first *state* named *"init"* and a last *state* named *"ready"*. In between *"init"* and *"ready"* a *plan component* can have additional *state* entries with arbitrary naming.

The purpose of the *"self" component* is to describe the main progress of the nano service as a whole. Most importantly the *"self" component* last *state* named *"ready"* must have status *"reached"* if and only if the nano service as a whole has completed. Other arbitrary components as well as states are added to the plan if they have meaning for the specific nano service i.e more specific progress reporting.

A *plan* also defines an empty leaf *failed* which is set if and only if any *state* in any *component* has a status set to *"failed"*. As such this is an aggregation to make it easy to verify if a RFM service is progressing without problems or not.

The following is an illustration of using the plan to report progress of a nano service:

```
ncs# show vpn l3vpn volvo plan
NAME                    TYPE   STATE              STATUS       WHEN
------------------------------------------------------------------------------------
self                    self   init               reached      2016-04-08T09:22:40
                               ready              not-reached  -
endpoint-branch-office  l3vpn  init               reached      2016-04-08T09:22:40
                               qos-configured     reached      2016-04-08T09:22:40
```

```
                                ready              reached      2016-04-08T09:22:40
endpoint-head-office    l3vpn  init               reached      2016-04-08T09:22:40
                                pe-created         not-reached  -
                                ce-vpe-topo-added  not-reached  -
                                vpe-p0-topo-added  not-reached  -
                                qos-configured     not-reached  -
                                ready              not-reached  -
```

Service progress monitoring

Plans were introduced to standardize the operational data that show progress of reactive fastmap (RFM) nano services. This gives the user a standardized view of all nano services and can answer the question weather a service instance has run to completion or not. To keep track of the progress of *plans service progress monitoring* (SPM) is introduced. The idea with *SPM* is that time limits are put on the progress of plan states. To do so, a *policy* and a *trigger* is needed.

A *policy* defines what plan components and states needs to be in what *status* for the *policy* to be true. A *policy* also defines how long time it can be false without considered jeopardized and how long time it can be false without considered violated. Further it may define an action, that is called in case of a policy being jeopardized, violated or successful.

A *trigger* is used to associate a policy with a service and a component.

The following is an illustration of using a SPM to track progress of an RFM service, in this case the policy specifies that the self components ready state must be reached for the policy to be true:

```
ncs# show vpn l3vpn volvo service-progress-monitoring
                                                                JEOPARDY                       VI
NAME  POLICY         START TIME           JEOPARDY TIME        RESULT    VIOLATION TIME       RE
-----------------------------------------------------------------------------------------------------
self  service-ready  2016-04-08T09:22:40  2016-04-08T09:22:40  -         2016-04-08T09:22:40  -
```

CHAPTER 5

NSO CLI

CLI Quick Start

The NSO CLI (command line interface) provides a unified CLI towards the complete network. The NSO CLI is a northbound interface to the NSO representation of the network devices and network services. Do not confuse this with a cut-through CLI that reaches the devices directly. Although the network might be a mix of vendors and device interfaces with different CLI flavors, NSO provides *one* northbound CLI.

Starting the CLI:

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

Note the use of the **-u** parameter which tells NSO which user to authenticate towards NSO. It is a common mistake to forget this. This user must be configured in NSO AAA (Authentication, Authorization and Accounting).

Like many CLI:s there is an operational mode and a configuration mode. Show commands displays different data in those modes. A show in configuration mode displays network configuration data from the NSO configuration database, the CDB. Show in operational mode shows live values from the devices and any operational data stored in the CDB. The CLI starts in operational mode. Note that different prompts are used for the modes (these can be changed in `ncs.conf` configuration file).

NSO organize all managed devices as a list of devices. The path to a specific device is **devices device** ***`DEVICE-NAME`***. The CLI sequence below does the following:

1. Show operational data for all devices: fetches operational data from the network devices like interface statistics, and also operational data that is maintained by NSO like alarm counters.
2. Move to configuration mode. Show configuration data for all devices: In this example this is done before the configuration from the real devices has been loaded in the network to NSO. At this point only the NSO configured data like IP Address, port etc. are shown.

Show device operational data and configuration data:

```
admin@ncs# show devices device
devices device ce0
 ...
 alarm-summary indeterminates 0
 alarm-summary criticals 0
 alarm-summary majors 0
 alarm-summary minors 0
 alarm-summary warnings 0
devices device ce1
 ...
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# show full-configuration devices device
devices device ce0
 address   127.0.0.1
 port      10022
 ssh host-key ssh-dss
 ...
!
devices device ce1
 ...
!
...
```

It can be annoying to move between modes to display configuration data and operational data. The CLI has ways around this.

Show config data in operational mode and vice versa:

```
admin@ncs# show running-config devices device
admin@ncs(config)# do show running-config devices device
```

Look at the device configuration above, there is no configuration that relates to the actual configuration on the devices. In order to boot-strap NSO and discover the device configuration it is possible to perform an action to synchronize NSO from the devices, **devices sync-from**. This reads the configuration over available device interfaces and populates the NSO data-store with the corresponding configuration. The device specific configuration is populated below the devices entry in the configuration tree and can be listed specifically.

Perform action to synchronize from devices:

```
admin@ncs(config)# devices sync-from
sync-result {
```

```
    device ce0
    result true
}
sync-result {
    device ce1
    result true
}
...
```

Display the device configuration after the synchronization:

```
admin@ncs(config)# show full-configuration devices device ce0 config
devices device ce0
 config
  no ios:service pad
  no ios:ip domain-lookup
  no ios:ip http secure-server
  ios:ip source-route
  ios:interface GigabitEthernet0/1
  exit
  ios:interface GigabitEthernet0/10
  exit
  ios:interface GigabitEthernet0/11
  exit
  ios:interface GigabitEthernet0/12
  exit
  ios:interface GigabitEthernet0/13
  exit
  ...
 !
!
...
```

Overview

NSO provides a network CLI in two different style (selectable by the user): J-style and C-style. The CLI is automatically rendered using the data models described by the YANG files. There are three distinctly different types of YANG files, the built-in NSO models describing the device manager and the service manager, models imported from the managed devices and finally service models. Regardless of model type, the NSO CLI seamlessly handles all models as a whole.

This creates a auto-generated CLI, without any extra effort, except the design of our YANG files. The auto-generated CLI supports the following features:

- Unified CLI across complete network, devices and network services.
- Command line history and command line editor.
- Tab completion for content of the configuration database.
- Monitoring and inspecting log files.
- Inspecting the system configuration and system state.
- Copying and comparing different configurations, for example, between two interfaces or two devices.
- Configuring common setting across a range of devices.

The CLI contains commands for manipulating the network configuration.

A alias provides a shortcut for a complex command.

Alias expansion is performed when a command line is entered. Aliases are part of the configuration and are manipulated accordingly. This is done by manipulating the nodes in the alias configuration tree.

Actions in the YANG files are mapped into actual commands. In J-style CLI actions are mapped to the **request** commands.

Even though the auto-generated CLI is fully functional it can be customized and extended in numerous ways:

- Built-in commands can be moved, hidden, deleted, reordered and extended.
- Confirmation prompts can be added to built-in commands.
- New commands can be implemented using the Java API, ordinary executables and shell scripts.
- New commands can be mounted freely in the existing command hierarchy.
- The built-in tab completion mechanism can be overridden using user defined callbacks.
- New command hierarchies can be created.
- A command timeout can be added, both a global timeout for all commands, and command specific timeouts.
- Actions and parts of the configuration tree can be hidden and can later be made visible when the user enters a password.

How to customize and extend the auto-generated CLI is described in the Plug-and-play scripting section Chapter 9, *Plug-and-play Scripting*.

Two CLI modes

The CLI is entirely data model driven. The YANG model(s) define a hierarchy of configuration elements. The CLI follows this tree.

The NSO CLI provides various commands for configuring and monitoring software, hardware, and network connectivity of managed devices. The CLI supports two modes: *operational mode*, for monitoring the state of the NSO node; and *configure mode*, for changing the state of the network.

The prompt indicates which mode the CLI is in. When moving from operational mode to configure mode using the **configure** command, the prompt is changed from `host#` to `host(config)#`. The prompts can be configured using the *`c-prompt1`* and *`c-prompt2`* settings in the `ncs.conf` file.

For example:

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

Operational mode

Operational mode is the initial mode after successful login to the CLI. It is primarily used for viewing the system status, controlling the CLI environment, monitoring and troubleshooting network connectivity, and initiating the configure mode.

The list of base commands available in operational mode is listed below in the "Operational mode commands" section. Additional commands are rendered from the loaded YANG files.

Configure mode

Configure mode can be initiated by entering the **configure** command in operational mode. All changes to the network configuration are done to a copy of the active configuration. These changes do not take effect until a successful **commit** or **commit confirm** command is entered.

The list of base commands available in configure mode is listed below in the "Configure mode commands" section. Additional commands are rendered from the loaded YANG files.

Note When using `config` mode to enter/set passwords, you may face issues if you are using special characters in your password (e.g., `!`, `""`, `\`, etc.). Some characters are automatically escaped by the CLI, while others require manual escaping. Therefore, the recommendation is to always enclose your password in double quotes `" "` and avoid using quotes `"` and backslash `\` characters in your password. If you prefer including quotes and backslash in your password, remember to manually escape them, as shown in the example below:

```
admin@ncs(config)# devices authgroups group default umap
admin remote-name admin remote-password "admin\"admin"
```

Starting the CLI

Basics

The CLI is started using the **ncs_cli** program. It can be used as a login program (replacing the shell for a user), started manually once the user has logged in, or used in scripts for performing CLI operations.

In a some NSO installations, ordinary users would have the **ncs_cli** program as login shell, and the root user would have to login and then start the CLI using **ncs_cli**, whereas in others, the **ncs_cli** can be invoked freely as a normal shell command.

The **ncs_cli** program supports a range of options, primarily intended for debug and development purposes (see description below).

The **ncs_cli** program can also be used for batch processing of CLI commands, either by storing the commands in a file and running **ncs_cli** on the file, or by having the following line at the top of the file (with the location of the program modified appropriately):

`#!/bin/ncs_cli`

When the CLI is run non-interactively it will terminate at the first error and will only show the output of the commands executed. It will not output the prompt or echo the commands. This is the same behavior as for shell scripts.

To run a script non-interactively, such as a script or through a pipe, and still produce prompts and echo commands, use the `--interactive` option.

Command line options:

```
ncs_cli --help
Usage: ncs_cli [options] [file]
Options:
--help, -h            display this help
--host, -H <host>     current host name (used in prompt)
--address, -A <addr>  cli address to connect to
--port, -P <port>     cli port to connect to
--cwd,  -c <dir>      current working directory
--proto, -p <proto>   type of connection (tcp, ssh, console)
--verbose, -v         verbose output
--ip, -i              clients source ip[/port]
--interactive, -n     force interactive mode
--escape-char, -E <C> brute force shutdown when user enters ASCII C
-J                    Juniper style CLI
-C                    Cisco XR style CLI
```

```
--user, -u <user>     clients user name
--uid, -U <uid>       clients user id
--groups, -g <groups> clients group list
--gids, -D <gids>     clients group id list
--gid, -G <gid>       clients group id
--noaaa               disable AAA
--opaque, -O <opaque> pass opaque info
```

--host	The argument to host should be the host name of the device. The **ncs_cli** program will use the result of the system call `gethostname()` as default value. The host name is used in the CLI prompt.
--address	If NSO has been configured to listen to a different address than 127.0.0.1 for the communication between subsystems, then that address should be given as argument to `address`. This can be modified by using the to use the `NCS_IPC_ADDRESS` environment variable or recompile the **ncs_cli** program with the new address compiled in.
--port	If NSO has been configured to use a non-default port for the communication between subsystems, then that port number should be given as argument to `port`. This can be modified by using the `NCS_IPC_PORT` environment variable or recompile the **ncs_cli** program with the new port compiled in.
--cwd	Directory to use as current working directory in the CLI. Normally the user's home directory. The default is the directory where the **ncs_cli** program is started.
--proto	Should be the protocol used by the user to connect to the box, one of **tcp**, **ssh**, and **console**. The default is **ssh** for connections established with OpenSSH (the program inspects the SSH_CONNECTION environment variable), and **console** for everything else. This value is printed in the audit logs.
--verbose	If this argument is given, then the **ncs_cli** program will be a bit more talkative during the NSO handshake phase.
--ip	Should be the user's source IP address, if the user connects through SSH or telnet. The default is 127.0.0.1 to indicate the console. This value is printed in the audit logs.
--interactive	Force the CLI to echo commands and prompts even when not invoked from a terminal, i.e. when reading input from a file or through a pipe.
--escape-char	It is possible to forcefully terminate the CLI by repeating a special character three times in a row. The default character is control underscore. It can be changed to an arbitrary character using this option.
-J	Starts the CLI in J-style.
-C	Starts the CLI in C-style, Cisco XR style.
--user	The name of the user connecting. Used to set proper access rules and assign proper groups (if the group mapping is kept in NSO). The default is to use the login name of the user.
--uid	The numeric user id of the connected user. The uid will be used when executing osCommands, when checking file access permissions, and when creating files.
--gid	The numeric group id of the connected user. The gid will be used when executing osCommands, when checking file access permissions, and when creating files.
--groups	The argument to groups should be a comma-separated list of groups. The default is to send the OS groups that the user belongs to, i.e. the same as the **groups** shell command gives us.
--gids	The argument to gids should be a comma-separated list of numeric group ids representing the Unix supplementary groups for the user. These are used when executing osCommands and when checking file access permissions.

--noaaa	Disables AAA. This is useful during development but should be removed in a production system.
--opaque	Provide an opaque string that can be read by connecting data providers.

Two CLI styles

The CLI comes in two flavors C-Style (Cisco XR style) and J-style. It is possible to choose one specifically or switch between them.

Starting the CLI (C-style, Cisco XR style):

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

Starting the CLI (J-style):

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

It is possible to interactively switch between these styles while inside the CLI using the builtin **switch** command:

```
admin@ncs# switch cli
```

C-style is mainly used throughout the documentation for examples etc. except if otherwise stated.

Starting the CLI in an overloaded system

If the number of ongoing sessions have reached the configured system limit, no more CLI sessions will be allowed until one of the existing sessions have been terminated.

This makes it impossible to get into the system. A situation which may not be acceptable. The CLI therefore has a mechanism for handling this problem. When the CLI detects that the session limit has been reached it will check if the new user has privileges to execute the logout command. If the user does it will display a list of the current user sessions in NSO and ask the user if one of the sessions should be terminated to make room for the new session.

Modifying the configuration

Once NSO synchronized with the devices configuration, done by using the **devices sync-from** command, it is possible to modify the devices. The CLI is used to modify the NSO representation of the device configuration and then committed as a transaction to the network.

As an example, to change the the speed setting on the interface GigabitEthernet0/1 across several devices:

```
admin@ncs(config)# devices device ce0..1 config ios:interface GigabitEthernet0/1 speed auto
admin@ncs(config-if)# top
admin@ncs(config)# show configuration
devices device ce0
 config
  ios:interface GigabitEthernet0/1
   speed auto
  exit
 !
!
devices device ce1
 config
  ios:interface GigabitEthernet0/1
   speed auto
  exit
 !
!
```

```
admin@ncs(config)# commit ?
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
  [<cr>
admin@ncs(config)# commit
Commit complete.
```

Note the availability of commit flags.

Any failure on any device will make the whole transaction fail. It is also possible to perform a manual rollback, a rollback is the undoing of a commit.

This is operational data and the CLI is in configuration mode so the way of showing operational data in config mode is used.

The command **show configuration rollback changes** can be used to view rollback changes in more detail. It will show what will be done when the rollback file is loaded, similar to loading the rollback and using **show configuration**:

```
admin@ncs(config)# show configuration rollback changes 10019
devices device ce0
 config
  ios:interface GigabitEthernet0/1
   no speed auto
  exit
 !
!
devices device ce1
 config
  ios:interface GigabitEthernet0/1
   no speed auto
  exit
 !
!
```

The command **show configuration commit changes** can be used to see which changes were done in a given commit, i.e. the roll-forward commands performed in that commit:

```
admin@ncs(config)# show configuration commit changes 10019
!
! Created by: admin
! Date: 2015-02-03 12:29:08
! Client: cli
!
devices device ce0
 config
  ios:interface GigabitEthernet0/1
   speed auto
  exit
 !
```

```
!
devices device ce1
 config
  ios:interface GigabitEthernet0/1
   speed auto
  exit
 !
!
```

The command **rollback-files apply-rollback-file** can be used to perform the rollback:

```
admin@ncs(config)# rollback-files apply-rollback-file fixed-number 10019
admin@ncs(config)# show configuration
devices device ce0
 config
  ios:interface GigabitEthernet0/1
   no speed auto
  exit
 !
!
devices device ce1
 config
  ios:interface GigabitEthernet0/1
   no speed auto
  exit
 !
!
```

And now the **commit** the rollback:

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

When the command **rollback-files apply-rollback-file fixed-number 10019** is run the changes recorded in rollback 10019-N (where N is the highest, thus the most recent rollback number) will all be undone. In other words the configuration will be rolled back to the state it was in before the commit associated with rollback 10019 was performed.

It is also possible to undo individual changes by running the command **rollback-files apply-rollback-file selective**. E.g. to undo the changes recorded in rollback 10019, but not the changes in 10020-N run the command **rollback-files apply-rollback-file selective fixed-number 10019**.

This operation may fail if the commits following rollback 10019 depends on the changes made in rollback 10019.

Command output processing

It is possible to process the output from a command using an output redirect. This is done using the | character (a pipe character):

```
admin@ncs# show running-config | ?
Possible completions:
  annotation      Show only statements whose annotation matches a pattern
  append          Append output text to a file
  begin           Begin with the line that matches
  best-effort     Display data even if data provider is unavailable or
                  continue loading from file in presence of failures
  context-match   Context match
  count           Count the number of lines in the output
  csv             Show table output in CSV format
  de-select       De-select columns
  details         Display show/commit details
```

```
  display          Display options
  exclude          Exclude lines that match
  extended         Display referring entries
  hide             Hide display options
  include          Include lines that match
  linnum           Enumerate lines in the output
  match-all        All selected filters must match
  match-any        At least one filter must match
  more             Paginate output
  nomore           Suppress pagination
  save             Save output text to a file
  select           Select additional columns
  sort-by          Select sorting indices
  tab              Enforce table output
  tags             Show only statements whose tags matches a pattern
  until            End with the line that matches
```

The precise list of pipe commands depends on the command executed. Some pipe commands, like select and de-select are only available for the **show** command, whereas others are universally available.

Note that the **tab** pipe target is used to enforce table output which is only suitable for the list element. Naturally the table format is not suitable to display arbitrary data output since it needs to map the data to columns and rows. For example:

```
show running-config | tab
show running-config | include aaa | tab
```

clearly are not suitable because the data has a nested structure. It could take incredibly long time to display it if you use the **tab** pipe target on a huge amount of data which is not a list element.

Count the Number of Lines in the Output

This redirect target counts the number of lines in the output. For example:

```
admin@ncs# show running-config | count
Count: 1783 lines
admin@ncs# show running-config aaa | count
Count: 28 lines
```

Search for a String in the Output

The **include** targets is used to only include lines matching a regular expression:

```
admin@ncs# show running-config aaa | include aaa
aaa authentication users user admin
aaa authentication users user oper
aaa authentication users user private
aaa authentication users user public
```

In the example above only lines containing aaa are shown. Similarly lines not containing a regular expression can be included. This is done using the **exclude** target:

```
admin@ncs# show running-config aaa authentication | exclude password
aaa authentication users user admin
 uid        1000
 gid        1000
 ssh_keydir /var/ncs/homes/admin/.ssh
 homedir    /var/ncs/homes/admin
!
aaa authentication users user oper
 uid        1000
 gid        1000
```

```
 ssh_keydir /var/ncs/homes/oper/.ssh
 homedir    /var/ncs/homes/oper
!
aaa authentication users user private
 uid        1000
 gid        1000
 ssh_keydir /var/ncs/homes/private/.ssh
 homedir    /var/ncs/homes/private
!
aaa authentication users user public
 uid        1000
 gid        1000
 ssh_keydir /var/ncs/homes/public/.ssh
 homedir    /var/ncs/homes/public
 !
```

It is possible to display context for a match using the pipe command **include -c** . Matching lines will be prefixed by *<line no>:* and context lines with *<line no>-*. For example:

```
admin@ncs# show running-config aaa authentication | include -c 3 homes/admin
 2- uid        1000
 3- gid        1000
 4- password   $1$brH6BYLy$iWQA2T1I3PMonDTJOd0Y/1
 5: ssh_keydir /var/ncs/homes/admin/.ssh
 6: homedir    /var/ncs/homes/admin
 7-!
 8-aaa authentication users user oper
 9- uid        1000
```

It is possible to display context for a match using the pipe command **context-match**:

```
admin@ncs# show running-config aaa authentication | context-match homes/admin
aaa authentication users user admin
 ssh_keydir /var/ncs/homes/admin/.ssh
aaa authentication users user admin
 homedir    /var/ncs/homes/admin
```

It is possible to display the output starting at the first match of a regular expression. This is done using the **begin** pipe command:

```
admin@ncs# show running-config aaa authentication users | begin public
aaa authentication users user public
 uid        1000
 gid        1000
 password   $1$DzGnyJGx$BjxoqYEj0QKxwVX5fbfDx/
 ssh_keydir /var/ncs/homes/public/.ssh
 homedir    /var/ncs/homes/public
!
```

Saving the Output to a File

The output can also be saved to a file using the **save** or **append** redirect target:

```
admin@ncs# show running-config aaa | save /tmp/saved
```

Or to save the configuration, except all passwords:

```
admin@ncs# show running-config aaa | exclude password | save /tmp/saved
```

Regular expressions

The regular expressions is a subset of the regular expressions found in egrep and in the AWK programming language. Some common operators are:

.	Matches any character.
^	Matches the beginning of a string.
$	Matches the end of a string.
[abc...]	Character class, which matches any of the characters abc... Character ranges are specified by a pair of characters separated by a -.
[^abc...]	negated character class, which matches any character except abc....
r1 | r2	Alternation. It matches either r1 or r2.
r1r2	Concatenation. It matches r1 and then r2.
r+	Matches one or more rs.
r*	Matches zero or more rs.
r?	Matches zero or one rs.
(r)	Grouping. It matches r.

For example, to only display uid and gid do the following:

```
admin@ncs# show running-config aaa | include "(uid)|(gid)"
 uid        1000
 gid        1000
 uid        1000
 gid        1000
 uid        1000
 gid        1000
 uid        1000
 gid        1000
```

Displaying the configuration

There are several options for displaying the configuration and stats data in NSO. The most basic command consists of displaying a leaf or a subtree of the configuration by giving the path to the element.

To display the configuration of a device do:

```
admin@ncs# show running-config devices device ce0 config
devices device ce0
 config
  no ios:service pad
  no ios:ip domain-lookup
  no ios:ip http secure-server
  ios:ip source-route
  ios:interface GigabitEthernet0/1
  exit
  ios:interface GigabitEthernet0/10
  exit
  ...
 !
 !
```

This can also be done for a group of devices by substituting the instance name (ce0 in this case) with a the section called “Regular expressions”.

To display the config of all devices:

```
admin@ncs# show running-config devices device * config
devices device ce0
 config
  no ios:service pad
  no ios:ip domain-lookup
```

```
  no ios:ip http secure-server
  ios:ip source-route
  ios:interface GigabitEthernet0/1
  exit
  ios:interface GigabitEthernet0/10
  exit
  ...
 !
!
devices device ce1
 config
  ...
 !
!
...
```

It is possible to limit the output even further. View only the http settings on each device:

```
admin@ncs# show running-config devices device * config ios:ip http
devices device ce0
 config
  no ios:ip http secure-server
 !
!
devices device ce1
 config
  no ios:ip http secure-server
 !
!
...
```

There is an alternative syntax for this using the **select** pipe command:

```
admin@ncs# show running-config devices device * | \
    select config ios:ip http
devices device ce0
 config
  no ios:ip http secure-server
 !
!
devices device ce1
 config
  no ios:ip http secure-server
 !
!
...
```

The **select** pipe command can be used multiple times for adding additional content:

```
admin@ncs# show running-config devices device * | \
    select config ios:ip http | \
    select config ios:ip domain-lookup
devices device ce0
 config
  no ios:ip domain-lookup
  no ios:ip http secure-server
 !
!
devices device ce1
 config
  no ios:ip domain-lookup
  no ios:ip http secure-server
 !
```

```
!
...
```

There is also a **de-select** pipe command that can be used to instruct the CLI to not display certain parts of the config. The above printout could also be achieved by first selecting the ip container, and then de-selecting the source-route leaf:

```
admin@ncs# show running-config devices device * | \
    select config ios:ip | \
    de-select config ios:ip source-route
devices device ce0
 config
  no ios:ip domain-lookup
  no ios:ip http secure-server
 !
!
devices device ce1
 config
  no ios:ip domain-lookup
  no ios:ip http secure-server
 !
!
...
```

A use-case for the **de-select** pipe command is to de-select the config container in order to only display the device settings without actually displaying their config:

```
admin@ncs# show running-config devices device * | de-select config
devices device ce0
 address   127.0.0.1
 port      10022
 ssh host-key ssh-dss
  ...
 !
 authgroup default
 device-type cli ned-id cisco-ios
 state admin-state unlocked
!
devices device ce1
 ...
!
...
```

The above statements also work for the **save** command. To save the devices managed by NSO, but not the contents of their config container:

```
admin@ncs# show running-config devices device * | \
    de-select config | save /tmp/devices
```

It is possible to use the **select** command to select which list instances to display. To display all devices that has the interface GigabitEthernet 0/0/0/4:

```
admin@ncs# show running-config devices device * | \
    select config cisco-ios-xr:interface GigabitEthernet 0/0/0/4
devices device p0
 config
  cisco-ios-xr:interface GigabitEthernet 0/0/0/4
   shutdown
  exit
 !
!
devices device p1
```

```
 config
  cisco-ios-xr:interface GigabitEthernet 0/0/0/4
   shutdown
  exit
 !
!
...
```

This means "display all device instances that has the interface GigabitEthernet 0/0/0/4". Only the subtree defined by the select path will be displayed. It is also possible to display the entire content of the config container for each instance by using an additional select statement:

```
admin@ncs# show running-config devices device * | \
    select config cisco-ios-xr:interface GigabitEthernet 0/0/0/4 | \
    select config | match-all
devices device p0
 config
  cisco-ios-xr:hostname PE1
  cisco-ios-xr:interface MgmtEth 0/0/CPU0/0
  exit
  ...
  cisco-ios-xr:interface GigabitEthernet 0/0/0/4
   shutdown
  exit
 !
!
devices device p1
 config
  ...
  cisco-ios-xr:interface GigabitEthernet 0/0/0/4
   shutdown
  exit
 !
!
...
```

The **match-all** pipe command is used for telling the CLI to only display instances that matches all select commands. The default behavior is **match-any** which means to display instances that matches any of the given **select** commands.

The **display** command is used to format configuration and statistics data. There are several output formats available, some of these are unique to specific modes, such as configuration or operational mode. The output formats **json**, **keypath**, **xml**, and **xpath** are available in most modes and CLI styles (J, I, and C). The output formats **netconf** and **maagic** are only available if **devtools** has been set to **true** in the CLI session settings.

For instance, assuming we have a data model featuring a set of hosts, each containing a set of servers, we can display the configuration data as JSON. This is depicted in the example below.

```
admin@ncs# show running-config hosts | display json
{
  "data": {
    "pipetargets_model:hosts": {
      "host": [
        {
          "name": "host1",
          "enabled": true,
          "numberOfServers": 2,
          "servers": {
            "server": [
              {
```

```
              "name": "serv1",
              "ip": "192.168.0.1",
              "port": 5001
            },
            {
              "name": "serv2",
              "ip": "192.168.0.1",
              "port": 5000
            }
          ]
        }
      },
      {
        "name": "host2",
        "enabled": false,
        "numberOfServers": 0
...
```

Still working with the same data model as used in the example above, we might want to see the current configuration in keypath format. The following example shows how to do that, and shows the resulting output.

```
admin@ncs# show running-config hosts | display keypath
/hosts/host{host1} enabled
/hosts/host{host1}/numberOfServers 2
/hosts/host{host1}/servers/server{serv1}/ip 192.168.0.1
/hosts/host{host1}/servers/server{serv1}/port 5001
/hosts/host{host1}/servers/server{serv2}/ip 192.168.0.1
/hosts/host{host1}/servers/server{serv2}/port 5000
/hosts/host{host2} disabled
/hosts/host{host2}/numberOfServers 0
```

Range expressions

To modify a range of instances at the same time use range expressions or to display a specific range of instances.

Basic range expressions are written with a combination of x..y (meaning from x to y), x,y (meaning x and y) and * (meaning any value), example:

```
1..4,8,10..18
```

It is possible to use range expressions for all key elements of integer type, both for setting values, executing actions and displaying status and config.

Range expressions are also supported for key elements of non-integer types as long as they are restricted to the pattern [a-zA-Z-]*[0-9]+/[0-9]+/[0-9]+/.../[0-9]+ and the annotation tailf:cli-allow-range is used on the key leaf. This is the case for the device list.

The following can be done in the CLI to display a subset of the devices (ce0, ce1, ce3):

```
admin@ncs# show running-config devices device ce0..1,3
```

If the devices have names with slashes, for example, Firewall/1/1, Firewall/1/2, Firewall/1/3, Firewall/2/1, Firewall/2/2, and Firewall/2/3, expressions like this are possible:

```
admin@ncs# show running-config devices device Firewall/1-2/*
admin@ncs# show running-config devices device Firewall/1-2/1,3
```

In configure mode it is possible to edit a range of instances in one command:

```
admin@ncs(config)# devices device ce0..2 config ios:ethernet cfm ieee
```

or like this:

```
admin@ncs(config)# devices device ce0..2 config
admin@ncs(config-config)# ios:ethernet cfm ieee
admin@ncs(config-config)# show config
devices device ce0
 config
  ios:ethernet cfm ieee
 !
!
devices device ce1
 config
  ios:ethernet cfm ieee
 !
!
devices device ce2
 config
  ios:ethernet cfm ieee
 !
!
```

Command history

Command history is maintained separate for each mode. When entering configure mode from operational for the first time a empty history be used. It is not possible to access the command history from operational mode when in configure mode and vice versa. When exiting back into operational mode access to the command history from the preceding operational mode session will be used. Likewise the old command history from the old configure mode session will be used when re-entering configure mode.

Command line editing

The default key strokes for editing the command line and moving around the command history are as follows.

Moving the cursor:

Move the cursor back one character
: Ctrl-b or Left Arrow

Move the cursor back one word
: Esc-b or Alt-b

Move the cursor forward one character
: Ctrl-f or Right Arrow

Move the cursor forward one word
: Esc-f or Alt-f

Move the cursor to the beginning of the command line
: Ctrl-a or Home

Move the cursor to the end of the command line
: Ctrl-e or End

Delete characters:

Delete the character before the cursor
: Ctrl-h, Delete, or Backspace

Delete the character following the cursor
    Ctrl-d
Delete all characters from the cursor to the end of the line
    Ctrl-k
Delete the whole line
    Ctrl-u or Ctrl-x
Delete the word before the cursor
    Ctrl-w, Esc-Backspace, or Alt-Backspace
Delete the word after the cursor
    Esc-d or Alt-d

Insert recently deleted text:

Insert the most recently deleted text at the cursor
    Ctrl-y

Display previous command lines:

Scroll backward through the command history
    Ctrl-p or Up Arrow
Scroll forward through the command history
    Ctrl-n or Down Arrow
Search the command history in reverse order
    Ctrl-r
Show a list of previous commands
    run the "show cli history" command

Capitalization:

Capitalize the word at the cursor, i.e. make the first character uppercase and the rest of the word lowercase
    Esc-c
Change the word at the cursor to lowercase
    Esc-l
Change the word at the cursor to uppercase
    Esc-u

Special:

Abort a command/Clear line
    Ctrl-c
Quote insert character, i.e. do not treat the next keystroke as an edit command
    Ctrl-v/ESC-q
Redraw the screen
    Ctrl-l
Transpose characters
    Ctrl-t

Enter multi-line mode. Enables entering multi-line values when prompted for a value in the CLI
    ESC-m
Exit configuration mode.
    Ctrl-z

Using CLI completion

It is not necessary to type the full command or option name for the CLI to recognize it. To display possible completions, type the partial command followed immediately by <tab> or <space>.

If the partially typed command uniquely identifies a command, the full command name will appear. Otherwise a list of possible completions is displayed.

Long lines can be broken into multiple lines using the backslash (\) character at the end of the line. This is primarily useful inside scripts.

Completion is disabled inside quotes. To type an argument containing spaces either quote them with a \ (e.g. **file show foo\ bar**) or with a " (e.g. **file show "foo bar"**). Space completion is disabled when entering a filename.

Command completion also applies to filenames and directories:

```
admin@ncs# <space>
Possible completions:
  alarms                  Alarm management
  autowizard              Automatically query for mandatory elements
  cd                      Change working directory
  clear                   Clear parameter
  cluster                 Cluster configuration
  compare                 Compare running configuration to another
                          configuration or a file
  complete-on-space       Enable/disable completion on space
  compliance              Compliance reporting
  config                  Manipulate software configuration information
  describe                Display transparent command information
  devices                 The managed devices and device communication settings
  display-level           Configure show command display level
  exit                    Exit the management session
  file                    Perform file operations
  help                    Provide help information
  ...
admin@ncs# dev<space>ices <space>
Possible completions:
  check-sync             Check if the NCS config is in sync with the device
  check-yang-modules     Check if NCS and the devices have compatible YANG
                         modules
  clear-trace            Clear all trace files
  commit-queue           List of queued commits
  ...
admin@ncs# devices check-s<space>ync
```

Comments, annotations and tags

All characters following a **!**, up to the next newline are ignored. This makes it possible to have comments in a file containing CLI commands, and still be able to paste the file into the command-line interface. For example:

```
! Command file created by Joe Smith
```

```
! First show the configuration before we change it
show running-config
! Enter configuration mode and configure an ethernet setting on the ce0 device
config
devices device ce0 config ios:ethernet cfm global
commit
top
exit
exit
! Done
```

To enter the comment character as an argument, it has to be prefixed with a backslash (\) or used inside quotes (").

The **/*** ... ***/** comment style is also supported.

When using large configurations it may make sense to be able to associate comments (annotations) and tags with the different parts. Then filter the configuration with respect to the annotations or tags. For example, tagging parts of the configuration that relates to a certain department or customer.

NSO has support for both tags and annotations. There is a specific set of commands available in the CLI for annotating and tagging parts of the configuration. There is also a set of pipe commands for controlling whether the tags and annotations should be displayed and for filtering depending on annotation and tag content.

The commands are:

- **annotate** <statement> <text>
- **tag add** <statement> <tag>
- **tag clear** <statement> <tag>
- **tag del** <statement> <tag>

Example:

```
admin@ncs(config)# annotate aaa authentication users user admin \
"Only allow the XX department access to this user."
admin@ncs(config)# tag add aaa authentication users user oper oper_tag
admin@ncs(config)# commit
Commit complete.
```

In order to view the placement of tags and annotations in the configuration it is recommended to use the pipe command **display curly-braces**. The annotations and tags will be displayed as comments where the tags are prefixed by **Tags:**. For example:

```
admin@ncs(config)# do show running-config aaa authentication users user | \
    tags oper_tag | display curly-braces
/* Tags: oper_tag */
user oper {
    uid        1000;
    gid        1000;
    password   $1$9qV138GJ$.olmolTfRbFGQhWJMZ9kA0;
    ssh_keydir /var/ncs/homes/oper/.ssh;
    homedir    /var/ncs/homes/oper;
}
admin@ncs(config)# do show running-config aaa authentication users user | \
    annotation XX | display curly-braces
/* Only allow the XX department access to this user. */
user admin {
    uid        1000;
```

```
    gid        1000;
    password   $1$EcQwYvnP$Rvq3MPTMSz29UaVOHA/511;
    ssh_keydir /var/ncs/homes/admin/.ssh;
    homedir    /var/ncs/homes/admin;
}
```

It is possible to hide the tags and annotations when viewing the configuration, or to explicitly include them in the listing. This is done using the **display annotations/tags** and **hide annotations/tags** pipe commands. To hide all attributes (annotations, tags and FASTMAP attributes) use the **hide attributes** pipe command.

Annotations and tags are part of the configuration. When adding, removing or modifying a annotation or a tag the configuration needs to be committed similar to any other change to the configuration.

CLI messages

Messages appear when entering and exiting configure mode, when committing a configuration and when typing a command or value that is not valid:

```
admin@ncs# show c
-----------------^
syntax error:
Possible alternatives starting with c:
  cli           - Display cli settings
  configuration - Commit configuration changes
admin@ncs# show configuration
------------------------------^
syntax error: expecting
  commit - Commit configuration changes
```

When committing a configuration, the CLI first validates the configuration and if there is a problem it will indicate what the problem is.

If a missing identifier or a value is out of range a message will indicate where the errors are:

```
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# nacm rule-list any-group rule allowrule
admin@ncs(config-rule-allowrule)# commit
Aborted: 'nacm rule-list any-group rule allowrule action' is not configured
```

ncs.conf settings

Parts of the CLI behavior can be controlled from the `ncs.conf` file. See the ncs.conf(5) in *Manual Pages* manual page for a comprehensive description of all the options.

CLI Environment

There are a number of session variables in the CLI. They are only used during the session and are not persistent. Their values are inspected using **show cli** in operational mode, and set using **set** in operational mode. Their initial values are in order derived from the content of the ncs.conf file, and the global defaults as configured at `/aaa:session` and user specific settings configured at `/aaa:user{<user>}/setting`.

```
admin@ncs# show cli
autowizard            false
complete-on-space     true
display-level         99999999
history               100
```

```
idle-timeout          1800
ignore-leading-space  false
output-file           terminal
paginate              true
prompt1               \h\M#
prompt2               \h(\m)#
screen-length         71
screen-width          80
service prompt config true
show-defaults         false
terminal              xterm-256color
...
```

The different values control different parts of the CLI behavior.

autowizard (*true* | *false*)

When enabled, the CLI will prompt the user for required settings when a new identifier is created.

For example:

```
admin@ncs(config)# aaa authentication users user John
Value for 'uid' (<int>): 1006
Value for 'gid' (<int>): 1006
Value for 'password' (<hash digest string>): ******
Value for 'ssh_keydir' (<string>): /var/ncs/homes/john/.ssh
Value for 'homedir' (<string>): /var/ncs/homes/john
```

This helps the user set all mandatory settings.

It is recommended to disable the autowizard before pasting in a list of commands, in order to avoid prompting. A good practice is to start all such scripts with a line that disables the autowizard:

```
autowizard false
...
autowizard true
```

complete-on-space (*true* | *false*)

Controls if command completion should be attempted when <space> is entered. Entering <tab> always results in command completion.

devtools (*true* | *false*)

Controls if certain commands that are useful for developers should be enabled. The command **xpath** and **timecmd** are examples of such a command.

history (*<integer>*)

Size of CLI command history.

idle-timeout (*<seconds>*)

Maximum idle time before being logged out. Use 0 (zero) for for infinity.

ignore-leading-space (*true* | *false*)

Controls if leading spaces should be ignored or not. This is useful to turn off when pasting commands into the CLI.

paginate (*true* | *false*)

Some commands paginate (or MORE process) the output, for example **show running-config**. This can be disabled or enabled. It is enabled by default. Setting the screen length to 0 has the same effect as turning off pagination.

screen length (*<integer>*)

Current length of terminal. This is used when paginating output to get proper line count. Setting this to 0 (zero) means it becomes maximum length and turns off pagination.

screen width (<*integer*>)

Current width of terminal. This is used when paginating output to get proper line count. Setting this to 0 (zero) means it becomes maximum width.

service prompt config

Controls whether a prompt should be displayed in configure mode. If set to false then no prompt will be displayed. The setting is changed using the commands **no service prompt config** and **service prompt config** in configure mode.

terminal (*string*)

Terminal type. This setting is used for controlling how line editing is performed. Supported terminals are: dumb, vt100, xterm, linux, and ansi. Other terminals may also work but have no explicit support.

Commands

To get a full XML listing of the commands available in a running NSO instance use the ncs option `--cli-c-dump <file>`. The generated file is only intended for documentation purposes and *cannot* be used as input to the ncsc compiler. The command **show parser dump** can be used get a command listing.

Operational mode commands

Invoke an action

<path> <parameters>

Invokes the action found at `path` using the supplied *parameters*.

This command is auto-generated from the YANG file.

For example, given the following action specification in a YANG file:

```
tailf:action shutdown {
  tailf:actionpoint actions;
  input {
    tailf:constant-leaf flags {
      type uint64 {
        range "1 .. max";
      }
      tailf:constant-value 42;
    }
    leaf timeout {
      type xs:duration;
      default PT60S;
    }
    leaf message {
      type string;
    }
    container options {
      leaf rebootAfterShutdown {
        type boolean;
        default false;
      }
      leaf forceFsckAfterReboot {
        type boolean;
        default false;
      }
      leaf powerOffAfterShutdown {
        type boolean;
        default true;
      }
```

```
    }
  }
}
```

The action can be invoked in the following way

```
admin@ncs> shutdown timeout 10s message reboot options { \
    forceFsckAfterReboot true }
```

Builtin commands

commit (**abort** | **confirm**)

Abort or confirm a pending confirming commit. A pending confirming commit will also be aborted if the CLI session is terminated without doing **commit confirm**. The default is confirm.

Example:

```
admin@ncs# commit abort
```

config (**exclusive** | **terminal**) [**no-confirm**]

Enter configure mode. The default is **terminal**.

terminal

Edit a private copy of the running configuration, no lock is taken.

no-confirm

Enter configure mode ignoring any confirm dialog

Example:

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

file list *<directory>*

List files in *<directory>*.

Example:

```
admin@ncs# file list /config
rollback10001
rollback10002
rollback10003
rollback10004
rollback10005
```

file show *<file>*

Display contents of a *<file>*.

Example:

```
admin@ncs# file show /etc/skel/.bash_profile
# /etc/skel/.bash_profile

# This file is sourced by bash for login shells.  The following line
# runs our .bashrc and is recommended by the bash info pages.
[[ -f ~/.bashrc ]] && . ~/.bashrc
```

help *<command>*

Display help text related to *<command>*.

Example:

```
admin@ncs# help job
Help for command: job
    Job operations
```

job stop *<job id>*

Stop a specific background job. In the default CLI the only command that creates background jobs is **monitor start**.

Example:

```
admin@ncs# monitor start /var/log/messages
[ok][...]
admin@ncs# show jobs
JOB COMMAND
3   monitor start /var/log/messages
admin@ncs# job stop 3
admin@ncs# show jobs
JOB COMMAND
```

logout session *<session>*

Log out a specific user session from NSO. If the user held the **configure exclusive** lock, it will be released.

<sessionid>	Log out a specific user session.

Example:

```
admin@ncs# who
Session User  Context From         Proto Date     Mode
 25     oper  cli     192.168.1.72 ssh   12:10:40 operational
*24     admin cli     192.168.1.72 ssh   12:05:50 operational
admin@ncs# logout session 25
admin@ncs# who
Session User  Context From         Proto Date     Mode
*24     admin cli     192.168.1.72 ssh   12:05:50 operational
```

logout user *<username>*

Log out a specific user from NSO. If the user held the **configure exclusive** lock, it will be released.

<username>	Log out a specific user.

Example:

```
admin@ncs# who
Session User  Context From         Proto Date     Mode
 25     oper  cli     192.168.1.72 ssh   12:10:40 operational
*24     admin cli     192.168.1.72 ssh   12:05:50 operational
admin@ncs# logout user oper
admin@ncs# who
Session User  Context From         Proto Date     Mode
*24     admin cli     192.168.1.72 ssh   12:05:50 operational
```

script reload

Reload scripts found in the `scripts/command`directory. New scripts will be added and if a script file has been removed the corresponding CLI command will be purged. See the Plug-and-play scripting section Chapter 9, *Plug-and-play Scripting*.

send (**all** | *<user>*) *<message>*

Display a message on the screens of all users who are logged in to the device or on a specific screen.

all

Display the message to all currently logged in users.

<user>

Display the message to a specific user.

Example:

```
admin@ncs# send oper "I will reboot system in 5 minutes."
```

In oper's session:

```
oper@ncs# Message from admin@ncs at 13:16:41...
I will reboot system in 5 minutes.
EOF
```

show cli

Display CLI properties.

Example:

```
admin@ncs# show cli
autowizard            false
complete-on-space     true
display-level         99999999
history               100
idle-timeout          1800
ignore-leading-space  false
output-file           terminal
paginate              true
prompt1               \h\M#
prompt2               \h(\m)#
screen-length         71
screen-width          80
service prompt config true
show-defaults         false
terminal              xterm-256color
timestamp             disable
```

show history [<*limit*>]

Display CLI command history. By default the last 100 commands are listed. The size of the history list is configured using the history CLI setting. If a history limit has been specified only the last number of commands up to that limit will be shown.

Example:

```
admin@ncs# show history
06-19 14:34:02 -- ping router
06-20 14:42:35 -- show running-config
06-20 14:42:37 -- who
06-20 14:42:40 -- show history
admin@ncs# show history 3
14:42:37 -- who
14:42:40 -- show history
14:42:46 -- show history 3
```

show jobs

Display currently running background jobs.

Example:

```
admin@ncs# show jobs
JOB COMMAND
3   monitor start /var/log/messages
```

show parser dump <*command prefix*>

Shows all possible commands starting with *command prefix*.

show running-config [<*pathfilter*> [**sort-by** <*idx*>]]

Display current configuration. By default the whole configuration is displayed. It is possible to limit what is shown by supplying a pathfilter.

The *pathfilter* may be either a path pointing to a specific instance, or if an instance id is omitted, the part following the omitted instance is treated as a filter.

The **sort-by** argument can be given when the *pathfilter* points to a list element with secondary indexes. *idx* is the name of a secondary index. When given, the table will be sorted in the order defined by the secondary index. This makes it possible for the CLI user to control in which order instances should be displayed.

To show the aaa settings for the admin user:

```
admin@ncs# show running-config aaa authentication users user admin
aaa authentication users user admin
 uid        1000
 gid        1000
 password   $1$JA.1O3Tx$Zt1ycpnMlg1bVMqM/zSZ7/
 ssh_keydir /var/ncs/homes/admin/.ssh
 homedir    /var/ncs/homes/admin
!
```

To show all users that have group id 1000, omit the user id and instead specify gid 1000:

```
admin@ncs# show running-config aaa authentication users user * gid 1000
...
```

show *<path>* [**sort-by** *<idx>*]

This command shows the configuration as a table provided that *path* leads to a list element and the data can be rendered as a table (ie, the table fits on the screen). It is also possible to force table formatting of a list by using the **| tab** pipe command.

The **sort-by** argument can be given when the *path* points to a list element with secondary indexes. *idx* is the name of a secondary index. When given, the table will be sorted in the order defined by the secondary index. This makes it possible for the CLI user to control in which order instances should be displayed.

Example:

```
admin@ncs# show devices device ce0 module
NAME                       REVISION    FEATURE  DEVIATION
-----------------------------------------------------------
tailf-ned-cisco-ios        2015-03-16  -        -
tailf-ned-cisco-ios-stats  2015-03-16  -        -
```

source *<file>*

Execute commands from <file> as if they had been entered by the user. The autowizard is disabled when executing commands from the file, also any commands that require input from user (commands added by clispec, for example) will receive interrupt signal upon attempt to read from stdin.

timecmd *<command>*

Time command. It measures and displays the execution time of *<command>*.

*Note that this command will only be available if **devtools** has been set to **true** in the CLI session settings.*

Example:

```
admin@ncs# timecmd id
user = admin(501), gid=20, groups=admin, gids=12,20,33,61,79,80,81,98,100
Command executed in 0.00 sec
admin@ncs#
```

who

Display currently logged on users. The current session, i.e. the session running the show status command, is marked with an asterisk.

Example:

```
admin@ncs# who
Session User  Context From         Proto Date     Mode
 25     oper  cli     192.168.1.72 ssh   12:10:40 operational
*24     admin cli     192.168.1.72 ssh   12:05:50 operational
admin@ncs#
```

Configure mode commands

Configure a value

<*path*> [<*value*>]

Set a parameter. If a new identifier is created and **autowizard** is enabled, then the CLI will prompt the user for all mandatory sub-elements of that identifier.

This command is auto-generated from the YANG file.

If no <*value*> is provided, then the CLI will prompt the user for the value. No echo of the entered value will occur if <*path*> is an encrypted value, i.e. of the type *MD5DigestString*, *DESDigestString*, *DES3CBCEncryptedString*, *AESCFB128EncryptedString* or *AES256CFB128EncryptedString* as documented in the tailf-common.yang data-model.

Builtin commands

annotate <*statement*> <*text*>

Associate an annotation with a given configuration. To remove an annotation leave the text empty.

Only available when the system has been configured with attributes enabled.

commit (**check** | **and-quit** | **confirmed** | **to-startup**) [**comment** <*text*>] [**label** <*text*>]

Commit current configuration to running.

check

Validate current configuration.

and-quit

Commit to running and quit configure mode.

comment <*text*>

Associate a comment with the commit. The comment can later be seen when examining rollback files.

label <*text*>

Associate a label with the commit. The label can later be seen when examining rollback files.

copy <*instance path*> <*new id*>

Make a copy of an instance.

Copying between different ned-id versions works as long as the schema nodes being copied has not changed between the versions.

copy cfg [**merge** | **overwrite**] <*src path*> **to** <*dest path*>

Copy data from one configuration tree to another. Only data that makes sense at the destination will be copied. No error message will be generated for data that cannot be copied and the operation can fail completely without any error messages being generated.

For example to create a template from a part of a device config. First configure the device then copy the config into the template configuration tree.

```
admin@ncs(config)# devices template host_temp
admin@ncs(config-template-host_temp)# exit
```

```
admin@ncs(config)# copy cfg merge devices device ce0 config \
    ios:ethernet to devices template host_temp config ios:ethernet
admin@ncs(config)# show configuration diff
+devices template host_temp
+ config
+  ios:ethernet cfm global
+ !
+!
```

copy compare *<src path>* **to** *<dest path>*

Compare two arbitrary configuration trees. Items that does only appears in the src tree are ignored.

delete *<path>*

Delete a data element.

do *<command>*

Run command in operational mode.

edit *<path>*

Edit a sub-element. Missing elements in *path* will be created.

exit (**level** | **configuration-mode**)

level	Exit from this level. If performed on the top level, will exit configure mode. This is the default if no option is given.
configuration-mode	Exit from configuration mode regardless of which edit level.

help *<command>*

Shows help text for command.

hide *<hide-group>*

Re-hides the elements and actions belonging to the hide groups. No password is required for hiding. This command is hidden and not shown during command completion.

insert *<path>*

Inserts a new element. If the element already exists and has the indexedView option set in the data model, then the old element will be renamed to element+1 and the new element inserted in its place.

insert *<path>*[**first**| **last**| **before** *key*| **after** *key*]

Inject a new element into an ordered list. The element can be added first, last (default), before or after another element.

load (**merge** | **override** | **replace**) (**terminal** | *<file>*)

Load configuration from file or terminal.

merge	Merge content of file/terminal with current configuration.
override	Configuration from file/terminal overwrites the current configuration.
replace	Configuration from file/terminal replaces the current configuration.

If this is the current configuration:

```
devices device p1
 config
  cisco-ios-xr:interface GigabitEthernet 0/0/0/0
   shutdown
  exit
  cisco-ios-xr:interface GigabitEthernet 0/0/0/1
   shutdown
 !
!
```

And the **shutdown** value for the entry **GigabitEthernet 0/0/0/0** should be deleted. As the configuration file is basically just a sequence of commands with comments in between, the configuration file should look like this:

```
devices device p1
 config
  cisco-ios-xr:interface GigabitEthernet 0/0/0/0
   no shutdown
  exit
 !
!
```

The file can then be used with the command **load merge *`FILENAME`*** to achieve the desired results.

move *<path>*[**first**| **last**| **before** *key*| **after** *key*]

Move an existing element to a new position in an ordered list. The element can be moved first, last (default), before or after another element.

rename *<instance path> <new id>*

Rename an instance.

revert

Copy running configuration into current configuration, eg remove all uncommitted changes.

rload (**merge** | **override** | **replace**) (**terminal** | *<file>*)

Load file relative to the current submode. For example, given a file with a device config it is possible to enter one device and issue the **rload merge/override/replace <file>** command to load the config for that device, then enter another device and load the same config file using **rload**. See also the **load** command.

merge	Merge content of file/terminal with current configuration.
override	Configuration from file/terminal overwrites the current configuration.
replace	Configuration from file/terminal replaces the current configuration.

rollback-files apply-rollback-file (**id** | **fixed-number**) *<number>* [**path** *<path>*] [**selective**]

Return the configuration to a previously committed configuration. The system stores a limited number of old configurations. The number of old configurations to store is configured in the `ncs.conf` file. If more than the configured number of configurations are stored, then the oldest configuration is removed before creating a new one.

The configuration changes are stored in rollback files where the most recent changes are stored in the file rollbackN with the highest number N.

Only the deltas are stored in the rollback files. When rolling back the configuration to rollback N, all changes stored in rollback10001-rollbackN are applied.

There are two ways to address which rollback file to use, either **fixed-number <number>** to address an absolute rollback number or **id <number>** to address a relative number. E.g. the latest commit has relative rollback id 0, second-latest has id 1, and so on.

The optional path argument allows subtrees to be rolled back while the rest of the configuration tree remains unchanged.

Instead of undoing all changes from rollback10001 to rollbackN it is possible to undo only the changes stored in a specific rollback file. This may or may not work depending on which changes have been made to the configuration after the rollback was created. In some cases applying the rollback file may fail, or the configuration may require additional changes in order to be valid. E.g. to undo the changes recorded in rollback 10019, but not the changes in 10020-N run the command **rollback-files apply-rollback-file selective fixed-number 10019**.

Example:

```
admin@ncs(config)# rollback-files apply-rollback-file fixed-number 10005
```

This command is only available if rollback has been enabled in `ncs.conf`.

show full-configuration [<*pathfilter*> [**sort-by** <*idx*>]]

Show current configuration, taking local changes into account. The **show** command can be limited to a part of the configuration by providing a <*pathfilter*>.

The **sort-by** argument can be given when the *pathfilter* points to a list element with secondary indexes. *idx* is the name of a secondary index. When given, the table will be sorted in the order defined by the secondary index. This makes it possible for the CLI user to control in which order instances should be displayed.

show configuration [<*pathfilter*>]

Show current edits to the configuration.

show configuration merge [<*pathfilter*> [**sort-by** <*idx*>]]

Show current configuration, taking local changes into account. The **show** command can be limited to a part of the configuration by providing a <*pathfilter*>.

The **sort-by** argument can be given when the *pathfilter* points to a list element with secondary indexes. *idx* is the name of a secondary index. When given, the table will be sorted in the order defined by the secondary index. This makes it possible for the CLI user to control in which order instances should be displayed.

show configuration commit changes [<*number*> [<*path*>]]

Display edits associated with a commit, identified by the rollback number created for the commit. The changes are displayed as forward changes, as opposed to **show configuration rollback changes** which displays the commands for undoing the changes.

The optional path argument allows only edits related to a given subtree to be listed.

show configuration commit list [<*path*>]

List rollback files

The optional path argument allows only rollback files related to a given subtree to be listed.

show configuration rollback listed [<*number*>]

Display the operations needed to undo the changes performed in a commit associated with a rollback file. These are the changes that will be applied if the configuration is rolled back to that rollback number.

show configuration running [<*pathfilter*>]

Display running-configuration without taking uncommitted changes into account. An optional *pathfilter* can be provided to limit what is displayed.

show configuration diff [<*pathfilter*>]

Display uncommitted changes to the running-config in diff-style, ie with + and - in front of added and deleted configuration lines.

show parser dump <*command prefix*>

Shows all possible commands starting with *command prefix*.

tag add <*statement*> <*tag*>

Add a tag to a configuration statement.

Only available in when the system has been configured with attributes enabled.

tag del <*statement*> <*tag*>

Remove a tag from a configuration statement.

Only available in when the system has been configured with attributes enabled.

tag clear <*statement*>

Remove all tags from a configuration statement.

Only available in when the system has been configured with attributes enabled.

timecmd *<command>*

Time command. It measures and displays the execution time of *<command>*.

*Note that this command will only be available if **devtools** has been set to **true** in the CLI session settings.*

Example:

```
admin@ncs# timecmd id
user = admin(501), gid=20, groups=admin, gids=12,20,33,61,79,80,81,98,100
Command executed in 0.00 sec
admin@ncs#
```

top [*command*]

Exit to top level of configuration, or execute a command at the top level of the configuration.

unhide *<hide-group>*

Unhides all elements and actions belonging to the hide-group. It may be required to enter a password. This command is hidden and not shown during command completion

validate

Validates current configuration. This is the same operation as **commit check**.

xpath [**ctx** *<path>*] (**eval** | **must** | **when**) *<expression>*

Evaluate an XPath expression. A context-path may be given to be used as the current context for the evaluation of the expression. If no context-path is given, the current sub-mode will be used as the context-path. The pipe command **trace** may be used to display debug/trace information during execution of the command.

*Note that this command will only be available if **devtools** has been set to **true** in the CLI session settings.*

eval

Evaluate an XPath expression.

must

Evaluate the expression as a YANG *must* expression.

when

Evaluate the expression as a YANG *when* expression.

reapply-commands [**best-effort** | **list**]

Reapply entered config commands since the latest commit. The command will stop on the first error by default.

Commands that may have unknown side-effects will be skipped and thus not reapplied, such as actions, custom commands etc. To display all commands, including those that will be skipped, the pipe command **details** can be used.

Note that this command will only be available if there is a conflict.

best-effort	Do not stop on the first error but continue to process the rest of the commands.
list	Display the current set of commands.

Customizing the CLI

Adding new commands

New commands can be added by placing a script in the `scripts/command` directory. See the Plug-and-play scripting section Chapter 9, *Plug-and-play Scripting*.

File access

The default behavior is to enforce Unix style access restrictions. That is, the users *uid*, *gid*, and *gids* are used to control what the user has read and write access to.

However, it is also possible to jail a CLI user to its home directory (or the directory where ncs_cli is started). This is controlled using the ncs.conf parameter *restricted-file-access*. If this is set to *true*, then the user only has access to the home directory.

Help texts

Help and information texts are specified in a number of places. In the yang files the *tailf:info* element is used to specify a descriptive text that is shown when the user enters ? in the CLI. The first sentence of the *info* text is used when showing one-line descriptions in the CLI.

Quoting and escaping scheme

Canonical quoting scheme

NCS understands multiple quoting schemes on input and de-quotes a value when parsing the command, but it uses what it considers a canonical quoting scheme when printing out this value, e.g. when pushing a configuration change to the device. However, different devices may have different quoting schemes, possibly not compatible with NCS canonical quoting scheme. For example, the following value:

```
"foo\\/bar\\?baz"
```

cannot be printed out by NCS as two backslashes "\\" match "\" in quoting scheme used by NCS when encoding values.

General rules for NCS to represent backslash are as followings:

- "\" and "\\" are represented as "\"
- "\\\" and "\\\\" are represented as "\\\"
- "\\\\\" and "\\\\\\" are represented as "\\\\\"

and so on. It can only get an odd number of backslashes output from NCS.

A backslash "\" is represented as a backslash "\" when it is followed by a character that does not need to be escaped, but is represented as double backslashes "\\" if the next character could be escaped. With remote passwords, if you are using special characters, be sure to follow recommended guidelines, see the section called “Configure mode” for more information.

Escape backslash handling

In order to let NCS to pass through a quoted string verbatim, one can do as stated below:

- Enable NCS configuration parameter `escapeBackslash` in the `ncs.conf` file. This is a global setting on NCS which affects all the NEDs.

Alternatively, a certain NED may be updated on request to be able to transform the value printed by NCS to what the device expects if one only wants to affect a certain device instead of all the connected ones.

Octal numbers handling

If there are numeric triplets following a backslash "\", NCS will treat them as octal numbers and convert them to one character based on ASCII code. For example:

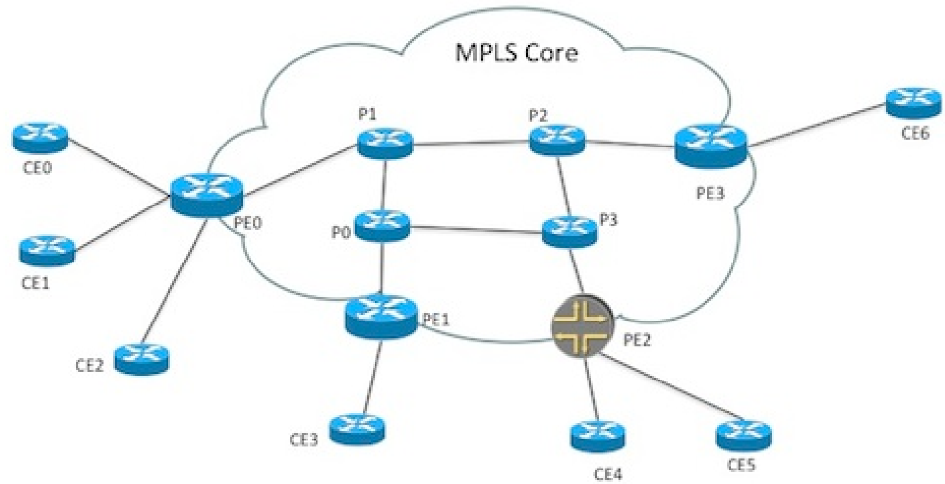
- "\123" is converted to "S"
- "\067" is converted to "7"

CHAPTER 6

The NSO Device Manager

Introduction

Throughout this section we will use the `examples.ncs/service-provider/mpls-vpn` example. The example network consists of Cisco ASR 9k and Juniper core routers (P and PE) and Cisco IOS based CE routers.



NSO Example network

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

- *NETCONF capable device*. The Device Manager will produce NETCONF `edit-config` RPC operations for each participating device.
- *SNMP device*. The Device Manager translates the changes made to the configuration into the corresponding SNMP SET PDUs
- *Device with Cisco CLI*. The device has a CLI with the same structure as Cisco IOS or XR routers. The Device Manager and a CLI NED is used to produce the correct sequence of CLI commands which reflects the changes made to the configuration.
- *Other devices* Devices which do not fit into any of the above mentioned categories a corresponding Generic NED is invoked. Generic NEDs are used for proprietary protocols like REST and for CLI flavours that are not resembling IOS or XR. The Device Manager will inform the Generic NED about the made changes and the NED will translate these to the appropriate operations toward the device.

NSO orchestrates an atomic transaction that has the very desirable characteristic of either the transaction as a whole ends up on all participating devices *and* in the NSO primary copy, or alternatively the whole transaction is aborted and all changes are automatically rolled-back.

The architecture of the NETCONF protocol is the enabling technology making it possible to push out configuration changes to managed devices and then in the case of other errors, roll back changes. Devices that do not support NETCONF, i.e., devices that do not have transactional capabilities can also participate, however depending on the device, error recovery may not be as good as it is for a proper NETCONF enabled device.

In order to understand the main idea behind the NSO device manager it is necessary to understand the NSO data model and how NSO incorporates the YANG data models from the different managed devices.

The NEDs will publish YANG data models even for non-NETCONF devices. In case of SNMP the YANG models are generated from the MIBs. For JunOS devices the JunOS NED generates a YANG from the JunOS XML Schema. For Schema-less devices like CLI devices the NED developer writes YANG models corresponding to the CLI structure. The result of this is the device manager and NSO CDB has YANG data models for all devices independent of underlying protocol.

The Managed Device Tree

The central part of the NSO YANG model, in the file `tailf-ncs-devices.yang`, has the following structure:

Example 4. tailf-ncs-devices.yang

```
submodule tailf-ncs-devices {
  belongs-to tailf-ncs {
    prefix ncs;
  }
  ...
  container devices {
    ......
    list device {
      key name;

      description
        "This list contains all devices managed by NCS.";

      leaf name {
        type string;
        description
          "A string uniquely identifying the managed device.";
      }

      leaf address {
        type inet:host;
        mandatory true;
        description
          "IP address or host name for the management interface on
           the device.";
      }
      leaf port {
        type inet:port-number;
        description
          "Port for the management interface on the device.  If this leaf
           is not configured, NCS will use a default value based on the
           type of device.  For example, a NETCONF device uses port 830,
           a CLI device over SSH uses port 22, and a SNMP device uses
           port 161.";
      }
      ....
      leaf authgroup {
        ....
      }
      container device-type {
```

```
      .......
      container config {
         ...
      }
   }
}
```

Each managed device is uniquely identified by its name, which is a free form text string. This is typically the DNS name of the managed device but could equally well be the string format of the IP address of the managed device or anything else. Furthermore, each managed device has a mandatory address/port pair that together with the `authgroup` leaf provides information to NSO how to connect and authenticate over SSH/NETCONF to the device. Each device also has a mandatory parameter `device-type` that specifies which southbound protocol to use for communication with the device. The following device types are available:

- NETCONF
- CLI - a corresponding CLI NED is needed to communicate with the device. This requires YANG models with the appropriate annotations for the device CLI.
- SNMP - The device speaks SNMP, preferably in read-write mode.
- Generic NED - a corresponding Generic NED is needed to communicate with the device. This requires YANG models and Java code.

The NSO CLI command below lists the NED types for the devices in the example network.

```
ncs(config)# show full-configuration devices device device-type
devices device ce0
 device-type cli ned-id cisco-ios-cli-3.8
!
...
devices device p0
 device-type cli ned-id cisco-iosxr-cli-3.5
!
devices device p1
 device-type cli ned-id cisco-iosxr-cli-3.5
!
...
devices device pe2
 device-type netconf ned-id juniper-junos-nc-3.0
!
```

The empty container `/ncs:devices/device/config` is used as a mount point for the YANG models from the different managed devices.

As previously mentioned, NSO needs the following information in order to manage a device:

- The IP/Port of the device and authentication information.
- Some or all of the YANG data models for the device.

In the example setup, the address and authentication information is provided in the NSO database (CDB) initialization file. There are many different ways to add new managed devices. Actually, all of the NSO northbound interfaces can be used to manipulate the set of managed devices. This will be further described later.

Once NSO has started you can inspect the meta information for the managed devices through the NSO CLI. This is an example session:

Example 5. Show device configuration in NSO CLI

```
ncs(config)# show full-configuration devices device
```

```
devices device ce0
 address   127.0.0.1
 port      10022
 ssh host-key ssh-dss
 ...
 authgroup default
 device-type cli ned-id cisco-ios-cli-3.8
 state admin-state unlocked
 config
 ...
 !
!
devices device ce1
 address   127.0.0.1
 port      10023
 ssh host-key ssh-dss
...
 !
 authgroup default
 device-type cli ned-id cisco-ios-cli-3.8
 state admin-state unlocked
 config
 ...
 !
!
```

Or alternatively, this information could be retrieved from the NSO northbound NETCONF interface by running the simple Python based netconf-console program towards the NSO NETCONF server.

Example 6. Show device configuration in NETCONF

```
$ netconf-console --get-config -x "/devices/device[name='ce0']"
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
  <data>
    <devices xmlns="http://tail-f.com/ns/ncs">
      <device>
        <name>ce0</name>
        <address>127.0.0.1</address>
        <port>10022</port>
        <ssh>
          <host-key>
            <algorithm>ssh-dss</algorithm>

            ...

        <authgroup>default</authgroup>
        <device-type>
          <cli>
          <ned-id xmlns:cisco-ios-cli-3.8="http://tail-f.com/ns/ned-id/cisco-ios-cli-3.8">
            cisco-ios-cli-3.8:cisco-ios-cli-3.8
          </ned-id>
          </cli>
        </device-type>
        <state>
          <admin-state>unlocked</admin-state>
        </state>
        <config>

        ...
```

```
        </config>
      </device>
    </devices>
  </data>
</rpc-reply>
```

All devices in Example 5, "Show device configuration in NSO CLI" and Example 6, "Show device configuration in NETCONF" have `/devices/device/state/admin-state` set to `unlocked`, this will be described later in this chapter.

The NED Packages

In order to communicate with a managed device, a NED for that device type needs to be loaded by NSO. A NED contains the YANG model for the device and corresponding driver code to talk CLI, REST, SNMP, etc. NEDs are distributed as packages.

Example 7. Installed Packages

```
ncs# show packages
packages package cisco-ios-cli-3.8
 package-version 3.8.0.1
 description     "NED package for Cisco IOS"
 ncs-min-version [ 3.2.2 3.3 3.4 ]
 directory       ./state/packages-in-use/1/cisco-ios-cli-3.8
 component IOSDp2
  callback java-class-name [ com.tailf.packages.ned.ios.IOSDp2 ]
 component IOSDp
  callback java-class-name [ com.tailf.packages.ned.ios.IOSDp ]
 component cisco-ios
  ned cli ned-id  cisco-ios-cli-3.8
  ned cli java-class-name com.tailf.packages.ned.ios.IOSNedCli
  ned device vendor Cisco
 ...
 oper-status up
packages package cisco-iosxr-cli-3.5
 package-version 3.5.0.7
 description     "NED package for Cisco IOS XR"
 ncs-min-version [ 3.2.2 3.3 ]
 directory       ./state/packages-in-use/1/cisco-iosxr-cli-3.5
 component cisco-ios-xr
  ned cli ned-id  cisco-iosxr-cli-3.5
  ned cli java-class-name com.tailf.packages.ned.iosxr.IosxrNedCli
  ned device vendor Cisco
 ...
 oper-status up
packages package juniper-junos-nc-3.0
 package-version 3.0.14.2
 description     "NED package for all JunOS based Juniper routers"
 ncs-min-version [ 3.0.0.1 3.1 3.2 3.3 3.4 ]
 directory       ./state/packages-in-use/1/juniper-junos-nc-3.0
 component junos
  ned netconf ned-id juniper-junos-nc-3.0
  ned device vendor Juniper
 oper-status up
 ...
```

The CLI command in Example 7, "Installed Packages" shows all the loaded packages. NSO loads packages at startup and can reload packages at run-time. By default the packages reside in the `packages` directory in the NSO run-time directory.

```
$ ls -l $NCS_DIR/examples.ncs/service-provider/mpls-vpn
total 160
...
drwxr-xr-x   8 stefan  staff    272 Oct  1 16:57 packages
...
$ ls -l $NCS_DIR/examples.ncs/service-provider/mpls-vpn/packages
total 24
cisco-ios
cisco-iosxr
juniper-junos
...
```

Starting the NSO Daemon

Once you have access to the network information for a managed device, its IP address and authentication information, as well as the data models of the device, you can actually manage the device from NSO.

You start the **ncs** daemon in a terminal like:

```
% ncs
```

Which is the same as, NSO loads it config from a `ncs.conf` file

```
% ncs -c ./ncs.conf
```

During development it is sometimes convenient to run ncs in the foreground as:

```
% ncs -c ./ncs.conf --foregound --verbose
```

Once the daemon is running you can issue the command:

```
% ncs --status
vsn: 7.1
SMP support: yes, using 8 threads
Using epoll: yes
available modules: backplane,netconf,cdb,cli,snmp,webui
...
... lots of output
```

To get more information about options to **ncs** do:

```
% ncs --help
```

The **ncs --status** command produces a lengthy list describing for example which YANG modules are loaded in the system. This is a valuable debug tool.

The same information is also available in the NSO CLI (and thus through all available northbound interfaces, including Maapi for Java programmers)

```
ncs# show ncs-state
ncs-state version 7.1
ncs-state smp number-of-threads 8
ncs-state epoll true
ncs-state daemon-status started
...
```

Synchronizing Devices

When the NSO daemon is running and has been initialized with IP/Port and authentication information as well as imported all modules you can start to manage devices through NSO.

NSO provides the ability to synchronize the configuration *to* or *from* the device. If you know that the device has the correct configuration you can choose to synchronize from a managed device whereas if you know NSO has the correct device configuration and the device is incorrect, you can choose to synchronize from NSO to the device.

In the normal case, the configuration on the device and the copy of the configuration inside NSO should be identical.

In a cold start situation like in the mpls-vpn example, where NSO is empty and there are network devices to talk to, it makes sense to synchronize from the devices. You can choose to synchronize from one device at a time or from all devices at once. Here is a CLI session to illustrate this.

Example 8. Synchronize from Devices

```
ncs(config)# devices sync-from
sync-result {
    device ce0
    result true
}
sync-result {
    device ce1
    result true
}
sync-result {
    device ce2
    result true
...
ncs(config)# show full-configuration devices device ce0
devices device ce0
...
 config
  no ios:service pad
  no ios:ip domain-lookup
  no ios:ip http secure-server
  ios:ip source-route
  ios:interface GigabitEthernet0/1
  exit
  ios:interface GigabitEthernet0/10
  exit
  ios:interface GigabitEthernet0/11
  exit
...
[ok][2010-04-13 16:29:15]
```

The command **devices sync-from**, in Example 8, “Synchronize from Devices”, is an action that is defined in the NSO data model. It is important to understand the model-driven nature of NSO. All devices are modeled in YANG, network services like MPLS VPN are also modeled in YANG, and the same is true for NSO itself. Anything that can be performed over the NSO CLI or any north-bound is defined in the YANG files. The NSO YANG files are located here:

```
$ls $NCS_DIR/src/ncs/yang/
```

All packages comes with YANG files as well. For example the directory `packages/cisco-ios/src/yang/` contains the YANG definition of an IOS device.

The `tailf-ncs.yang` is the main part of the NSO YANG data model. The file mode `tailf-ncs.yang` includes all parts of the model from different files.

The actions `sync-from` and `sync-to` are modeled in the file `tailf-ncs-devices.yang`. The sync action(s) are defined as:

Example 9. tailf-ncs-devices.yang sync actions

```
grouping sync-from-output {
  list sync-result {
    key device;
    leaf device {
      type leafref {
        path "/devices/device/name";
      }
    }
    uses sync-result;
  }
}

grouping sync-result {
  description
    "Common result data from a 'sync' action.";

  choice outformat {
    leaf result {
      type boolean;
    }
    anyxml result-xml;
    leaf cli {
      tailf:cli-preformatted;
      type string;
    }
  }
  leaf info {
    type string;
    description
      "If present, contains additional information about the result.";
  }
}

...

container devices {

  ...

  tailf:action sync-from {
    description
      "Synchronize the configuration by pulling from all unlocked
       devices.";
    tailf:info "Synchronize the config by pulling from the devices";
    tailf:actionpoint ncsinternal {
      tailf:internal;
    }
    input {
      leaf suppress-positive-result {
        type empty;
        description
          "Use this additional parameter to only return
           devices that failed to sync.";
      }
      container dry-run {
        presence "";
        leaf outformat {
```

```
          type outformat2;
          description
            "Report what would be done towards CDB, without
             actually doing anything.";
        }
      }
    }
    output {
      uses sync-from-output;
    }
  }

  ...

  tailf:action sync-to {
    ...
  }

  ...

  list device {
    description
      "This list contains all devices managed by NCS.";

    key name;

    leaf name {
      description "A string uniquely identifying the managed device";
      type string;
    }

    ...

    tailf:action sync-from {
      description
        "Synchronize the configuration by pulling from the device.";
      tailf:info "Synchronize the config by pulling from the device";
      tailf:actionpoint ncsinternal {
        tailf:internal;
      }
      input {
        container dry-run {
          presence "";
          leaf outformat {
            type outformat2;
            description
              "Report what would be done towards CDB, without
               actually doing anything.";
          }
        }
      }
      output {
        uses sync-result;
      }
    }
    tailf:action sync-to {

    ...
```

Synchronizing from NSO to the device is common when a device has been configured out-of-band. NSO has no means to enforce that devices are not directly reconfigured behind the scenes of NSO; however, once a out-of-band configuration has been performed, NSO has the ability to detect the fact. When this happens it may (or may not, depending on the situation at hand) make sense to synchronize from NSO to the device, i.e. undo the rogue reconfigurations.

The command to do that is:

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

A `dry-run` option is available for the action `sync-to`.

```
ncs# devices device ce0 sync-to dry-run
data {
    ...
}
```

This makes it possible to investigate the changes before they are transmitted to the devices.

Partial sync-from

It is possible to synchronize a part of the configuration (a certain subtree) from the device using the partial-sync-from action located under /devices. While it it primarily intended to be used by service developers as described in the section called “Partial Sync” in *Development Guide*, it is also possible to use directly from the NSO CLI (or any other northbound interface). Example 10, “Example of running partial-sync-from action via CLI” illustrates using this action via CLI, using "router" device from examples.ncs/getting-started/developing-with-ncs/0-router-network.

Example 10. Example of running partial-sync-from action via CLI

```
$ ncs_cli -C -u admin
ncs# devices partial-sync-from path [ \
/devices/device[name='ex0']/config/r:sys/interfaces/interface[name='eth0'] \
/devices/device[name='ex1']/config/r:sys/dns/server ]
sync-result {
    device ex0
    result true
}
sync-result {
    device ex1
    result true
}
ncs# show running-config devices device ex0..1 config
devices device ex0
 config
  r:sys interfaces interface eth0
   unit 0
    enabled
   !
   unit 1
    enabled
   !
   unit 2
    enabled
    description "My Vlan"
    vlan-id     18
   !
  !
 !
!
devices device ex1
 config
  r:sys dns server 10.2.3.4
  !
```

```
 !
!
```

Configuring Devices

It is now possible to configure several devices through the NSO inside the same network transaction. To illustrate this start the NSO CLI from a terminal application.

Example 11. Configure Devices

```
$ ncs_cli -C -u admin
ncs# config
Entering configuration mode terminal
ncs(config)# devices device pe1 config cisco-ios-xr:snmp-server \
     community public RO
ncs(config-config)# top
ncs(config)# devices device ce0 config ios:snmp-server community public RO
ncs(config-config)# devices device pe2 config junos:configuration \
      snmp community public view RO
ncs(config-community-public)# top
ncs(config)# show configuration
devices device ce0
 config
  ios:snmp-server community public RO
 !
!
devices device pe1
 config
  cisco-ios-xr:snmp-server community public RO
 !
!
devices device pe2
 config
  ! first
  junos:configuration snmp community public
   view RO
  !
 !
!
ncs(config)# commit dry-run outformat native
native {
    device {
        name ce0
        data snmp-server community public RO
    }
    device {
        name pe1
        data snmp-server community public RO
    }
    device {
        name pe2
        data <rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
                  message-id="1">
               <edit-config xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
                 <target>
                   <candidate/>
                 </target>
                 <test-option>test-then-set</test-option>
                 <error-option>rollback-on-error</error-option>
                 <config>
                   <configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm">
                     <snmp>
```

```
                            <community>
                              <name>public</name>
                              <view>RO</view>
                            </community>
                          </snmp>
                        </configuration>
                      </config>
                    </edit-config>
                  </rpc>
    }
}
ncs(config)# commit
```

The Example 11, “Configure Devices” illustrates a multi host transaction. In the same transaction three hosts were re-configured. Had one of them failed, or been non-operational, the transaction as a whole would have failed.

As seen from the output of the command **commit dry-run outformat native**, NSO generates the native CLI and NETCONF commands which will be sent to each device when the transaction is committed.

Since the `/devices/device/config` path contains different models depending on the augmented device model NSO uses the data model prefix in the CLI names; `ios`, `cisco-ios-xr` and `junos`. Different data models might use the same name for elements and the prefix avoids name clashes.

NSO uses different underlying techniques to implement the atomic transactional behaviour in case of any error. NETCONF devices are straight-forward using confirmed commit. For CLI devices like IOS NSO calculates the reverse diff to restore the configuration to the state before the transaction was applied.

Connection Management

Each managed device needs to be configured with the IP address and the port where the CLI, NETCONF server etc of the managed device listens for incoming requests.

Connections are established on demand as they are needed. It is possible to explicitly establish connections, but that functionality is mostly there for troubleshooting connection establishment. We can for example do:

```
ncs# devices connect
connect-result {
    device ce0
    result true
    info (admin) Connected to ce0 - 127.0.0.1:10022
}
connect-result {
    device ce1
    result true
    info (admin) Connected to ce1 - 127.0.0.1:10023
}
...
```

We were able to connect to all managed devices. It is also possible to explicitly attempt to test connections to individual managed devices:

```
ncs# devices device ce0 connect
result true
info (admin) Connected to ce0 - 127.0.0.1:10022
```

Established connections are typically not closed right away when not needed, but rather pooled according to the rules described in the section called “Device Session Pooling”. This applies to NETCONF sessions as well as sessions established by CLI or generic NEDs via a connection-oriented protocol. In addition to session pooling, underlying SSH connections for NETCONF devices are also reused. Note that a single NETCONF session occupies one SSH channel inside an SSH connection, so multiple NETCONF sessions can co-exist in a single connection. When an SSH connection has been idle (no SSH channels open) for 2 minutes, the SSH connection is closed. If a new connection is needed later, a connection is established on demand.

There are three configuration parameters which can be used to control the connection establishment, `connect-timeout`, `read-timeout` and `write-timeout`. In the NSO data model file `tailf-ncs-devices.yang` these timeouts are modeled as:

```
submodule tailf-ncs-devices {
  ...
  container devices {
    ...
    grouping timeouts {
      description
        "Timeouts used when communicating with a managed device.";

      leaf connect-timeout {
        type uint32;
        units "seconds";
        description
          "The timeout in seconds for new connections to managed
           devices.";
      }
      leaf read-timeout {
        type uint32;
        units "seconds";
        description
          "The timeout in seconds used when reading data from a
           managed device.";
      }
      leaf write-timeout {
        type uint32;
        units "seconds";
        description
          "The timeout in seconds used when writing data to a
           managed device.";
      }
    }
    ...
    container global-settings {
      ...
      uses timeouts {
        description
          "These timeouts can be overridden per device.";

        refine connect-timeout {
          default 20;
        }
        refine read-timeout {
          default 20;
        }
        refine write-timeout {
          default 20;
        }
      }
      ....
```

Thus to change these parameters (globally for all managed devices) you do:

```
ncs(config)# devices global-settings connect-timeout 30
```

```
ncs(config)# devices global-settings read-timeout 30
ncs(config)# commit
```

Or, to use a profile:

```
ncs(config)# devices profiles profile slow-devices connect-timeout 60
ncs(config-profile-slow-devices)# read-timeout 60
ncs(config-profile-slow-devices)# write-timeout 60
ncs(config-profile-slow-devices)# commit

ncs(config)# devices device ce3 device-profile slow-devices
ncs(config-device-ce3)# commit
```

Authentication Groups

When NSO connects to a managed device, it requires authentication information for that device. The authgroups are modeled in the NSO data model:

Example 12. tailf-ncs-devices.yang - Authgroups

```
submodule tailf-ncs-devices {
  ...
  container devices {
    ...

    container authgroups {
      description
        "Named authgroups are used to decide how to map a local NCS user to
         remote authentication credentials on a managed device.

         The list 'group' is used for NETCONF and CLI managed devices.

         The list 'snmp-group' is used for SNMP managed devices.";

      list group {
        key name;

        description
          "When NCS connects to a managed device, it locates the
           authgroup configured for that device.  Then NCS looks up
           the local NCS user name in the 'umap' list.  If an entry is
           found, the credentials configured is used when
           authenticating to the managed device.

           If no entry is found in the 'umap' list, the credentials
           configured in 'default-map' are used.

           If no 'default-map' has been configured, and the local NCS
           user name is not found in the 'umap' list, the connection
           to the managed device fails.";

        grouping remote-user-remote-auth {
          description
            "Remote authentication credentials.";

          choice login-credentials {
            mandatory true;
            case stored {
              choice remote-user {
                mandatory true;
                leaf same-user {
                  type empty;
```

```
          description
            "If this leaf exists, the name of the local NCS user is used
             as the remote user name.";
        }
        leaf remote-name {
          type string;
          description
            "Remote user name.";
        }
      }

      choice remote-auth {
        mandatory true;
        leaf same-pass {
          type empty;
          description
            "If this leaf exists, the password used by the local user
             when logging in to NCS is used as the remote password.";
        }
        leaf remote-password {
          type tailf:aes-256-cfb-128-encrypted-string;
          description
            "Remote password.";
        }
        case public-key {
          uses public-key-auth;
        }
      }
      leaf remote-secondary-password {
        type tailf:aes-256-cfb-128-encrypted-string;
        description
          "Some CLI based devices require a second
           additional password to enter config mode";
      }
    }
    case callback {
      leaf callback-node {
        description
          "Invoke a standalone action to retrieve login credentials for
          managed devices on the 'callback-node' instance.

          The 'action-name' action is invoked on the callback node that
          is specified by an instance identifer.";
        mandatory true;
        type instance-identifier;
      }
      leaf action-name {
        description
          "The action to call when a notification is received.

          The action must use 'authgroup-callback-input-params'
          grouping for input and 'authgroup-callback-output-params'
          grouping for output from tailf-ncs-devices.yang.";
        type yang:yang-identifier;
        mandatory true;
        tailf:validate ncs {
           tailf:internal;
           tailf:dependency "../callback-node";
        }
      }
    }
  }
```

```
        }
        leaf name {
          type string;
          description
            "The name of the authgroup.";
        }

        container default-map {
          presence "Map unknown users";
          description
            "If an authgroup has a default-map, it is used if a local
             NCS user is not found in the umap list.";
          tailf:info "Remote authentication parameters for users not in umap";
          uses remote-user-remote-auth;
        }

        list umap {
          key local-user;
          description
            "The umap is a list with the local NCS user name as key.
             It maps the local NCS user name to remote authentication
             credentials.";
          tailf:info "Map NCS users to remote authentication parameters";
          leaf local-user {
            type string;
            description
              "The local NCS user name.";
          }
          uses remote-user-remote-auth;
        }
      }
```

Each managed device must refer to a named authgroup. The purpose of an authgroup is to map local users to remote users together with the relevant SSH authentication information.

Southbound authentication can be done in two ways. One is to configure stored user and credential components as shown in Example 13, "Configured authgroup" and Example 14, "authgroup default-map". The other way is to configure a callback to retrieve user and credentials on demand as shown in Example 15, "authgroup-callback".

Example 13. Configured authgroup

```
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==
 !
```

```
!
```

In Example 13, “Configured authgroup” in the authgroup named `default` the two local users `oper` and `admin` shall use the remote users name `oper` and `admin` respectively with identical passwords.

Inside an authgroup, all local users need to be enumerated. Each local user name must have credentials configured which should be used for the remote host. In centralized AAA environments this is usually a bad strategy. You may also choose to instantiate a `default-map`. If you do that it probably only makes sense to specify the same user name/password pair should be used remotely as the pair that was used to log into NSO.

Example 14. authgroup default-map

```
ncs(config)# devices authgroups group default default-map same-user same-pass
ncs(config-group-default)# commit
Commit complete.
ncs(config-group-default)# top
ncs(config)# show full-configuration devices authgroups
devices authgroups group default
 default-map same-user
 default-map same-pass
 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==
 !
!
```

In Example 13, “Configured authgroup” only two users `admin` and `oper` were configured. If the `default-map` in Example 14, “authgroup default-map” is configured all local users not found in the `umap` list will end up in the `default-map`. For example if user `rocky` logs in to NSO with password "secret". Since NSO has a built-in SSH server and also a built-in HTTPS server, NSO will be able to pick up the clear text passwords and can then reuse the same password when NSO attempts to establish southbound SSH connections. The user `rocky` will end up in the `default-map` and when NSO attempts to propagate rocky's changes towards the managed devices, NSO will use the remote user name `rocky` with whatever password `rocky` used to log into NSO.

Authenticating southbound using stored configuration has two main components to define remote user and remote credentials. This is defined by the authgroup. As for southbound user, there exist two options, the same user logged in to NSO or another user, as specified in the authgroup. As for the credentials, there are three options.

1. Regular password.
2. Public key. This means that a private key, either from a file in the user's SSH key directory, or one that is configured in the /ssh/private-key list in the NSO configuration, is used for authentication. Refer to the section called “Publickey Authentication” for the details of how the private key is selected.

3 Finally, an interesting option is to use the 'same-pass' option. Since NSO runs its own SSH server and its own SSL server, NSO can pick up the password of a user in clear text. Hence, if the 'same-pass' option is chosen for an authgroup, NSO will reuse the same password when attempting to connect southbound to a managed device.

In case of authenticating southbound using a callback, remote user and remote credentials are obtained by an action invocation. The action is defined by the `callback-node` and `action-name` as in Example 15, "authgroup-callback" and supported credentials are remote password and optionally a secondary password for the provided local user, authgroup and device.

With remote passwords, you may encounter issues if you use special characters, such as quotes (`"`) and backslash (`\`) in your password. See the section called "Configure mode" for recommendations on how to avoid running into password issues.

Example 15. authgroup-callback

```
ncs(config)# devices authgroups group default umap oper
ncs(config-umap-oper)# callback-node /callback action-name auth-cb
ncs(config-group-oper)# commit
Commit complete.
ncs(config-group-oper)# top
ncs(config)# show full-configuration devices authgroups
devices authgroups group default
 default-map same-user
 default-map same-pass
 umap admin
  remote-name     admin
  remote-password $4$wIo7Yd068FRwhYYI0d4IDw==
 !
 umap oper
  callback-node /callback
  action-name   auth-cb
 !
!
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==
 !
!
```

Example 16. authgroup-callback.yang

```
module authgroup-callback {
  namespace "http://com/example/authgroup-callback";
  prefix authgroup-callback;

  import tailf-common {
    prefix tailf;
  }

  import tailf-ncs {
    prefix ncs;
  }

  container callback {
    description
      "Example callback that defines an action to retrieve
```

```
       remote authentication credentials";
     tailf:action auth-cb {
       tailf:actionpoint auth-cb-point;
       input {
         uses ncs:authgroup-callback-input-params;
       }
       output {
         uses ncs:authgroup-callback-output-params;
       }
     }
   }
 }
```

In Example 15, “authgroup-callback”, configuration for the umap entry of the `oper` user is changed to use a callback to retrieve southbound authentication credentials. Thus, NSO is going to invoke the action `auth-cb` defined in the callback-node `callback`. The callback-node is of type `instance-identifier` and refers to the container called `callback` defined in Example 16, “authgroup-callback.yang”, which includes an action defined by action-name `auth-cb` and uses groupings `authgroup-callback-input-params` and `authgroup-callback-output-params` for input and output parameters respectively. In Example 15, “authgroup-callback”, `authgroup-callback` module was loaded in NSO within an example package. Package development and action callbacks are not described here but more can be read in Chapter 18, *Package Development* in *Development Guide*, the section called “DP API” in *Development Guide* and Chapter 16, *Python API Overview* in *Development Guide*.

Caveats

Authentication groups and the functionality it brings comes with some limitations on where and how it is used.

- Callback option that enables authgroup-callback feature is not applicable for members of snmp-group list.
- Generic devices that implement own authentication scheme are not using any mapping or callback functionality provided by Authgroups.
- Cluster nodes use their own Authgroups and mapping model thus functionality differ, e.g callback option is not applicable.

Device Session Pooling

Opening a session towards a managed device is potentially time and resource consuming. Also, the probability that a recently accessed device is still subject to further request is reasonably high. These are motives for having a managed devices session pool in NSO.

The NSO device session pool is by default active and normally needs no maintenance. However under certain circumstances there might be of interest to modify its behaviour. Examples can be when some device type has characteristics that makes session pooling undesired, or when connections to a specific device is very costly and therefore the time that open sessions can stay in the pool should increase.

Note Changes from the default configuration of the NSO session pool should only be performed when absolutely necessary and when all effects of the change are understood.

NSO presents operational data that represent the current state of the session pool. To visualize this we use the CLI to connect to NSO and force connection to all known devices:

```
$ ncs_cli -C -u admin

admin connected from 127.0.0.1 using console on ncs
ncs# devices connect suppress-positive-result
```

We can now list all open sessions in the session-pool. But note that this is a live pool. Sessions will only remain open for a certain amount of time, the idle-time.

```
ncs# show devices session-pool
        DEVICE            MAX        IDLE
DEVICE  TYPE    SESSIONS  SESSIONS   TIME
-------------------------------------------
ce0     cli     1         unlimited  30
ce1     cli     1         unlimited  30
ce2     cli     1         unlimited  30
ce3     cli     1         unlimited  30
ce4     cli     1         unlimited  30
ce5     cli     1         unlimited  30
pe0     cli     1         unlimited  30
pe1     cli     1         unlimited  30
pe2     cli     1         unlimited  30
```

In addition to the idle-time for sessions we can also see the type of device, current number of pooled sessions and maximum number of pooled session.

We can close pooled sessions for specific devices.

```
ncs# devices session-pool pooled-device pe0 close
ncs# devices session-pool pooled-device pe1 close
ncs# devices session-pool pooled-device pe2 close
ncs# show devices session-pool
        DEVICE            MAX        IDLE
DEVICE  TYPE    SESSIONS  SESSIONS   TIME
-------------------------------------------
ce0     cli     1         unlimited  30
ce1     cli     1         unlimited  30
ce2     cli     1         unlimited  30
ce3     cli     1         unlimited  30
ce4     cli     1         unlimited  30
ce5     cli     1         unlimited  30
```

And we can close all pooled sessions in the session pool.

```
ncs# devices session-pool close
ncs# show devices session-pool
% No entries found.
```

The session pool configuration is found in the tailf-ncs-devices.yang submodel. The following part of the YANG device-profile-parameters grouping controls how the session pool is configured:

```
grouping device-profile-parameters {

  ...

    container session-pool {
      tailf:info "Control how sessions to related devices can be pooled.";
      description
        "NCS uses NED sessions when performing transactions, actions
         etc towards a device. When such a task is completed the NED
         session can either be closed or pooled.

         Pooling a NED session means that the session to the
```

```
          device is kept open for a configurable amount of
          time. During this time the session can be re-used for a new
          task. Thus the pooling concept exists to reduce the number
          of new connections needed towards a device that is often
          used.

          By default NCS uses pooling for all device types except
          SNMP. Normally there is no need to change the default
          values.";

       leaf max-sessions {
         type union {
           type enumeration {
             enum unlimited;
           }
           type uint32;
         }
         description
           "Controls the maximum number of open sessions in the pool for
            a specific device. When this threshold is exceeded the oldest
            session in the pool will be closed.
            A Zero value will imply that pooling is disabled for
            this specific device. The label 'unlimited' implies that no
            upper limit exists for this specific device";
       }

       leaf idle-time {
         tailf:info
           "The maximum time that a session is kept open in the pool";
         type uint32 {
           range "1 .. max";
         }
         units "seconds";
         description
           "The maximum time that a session is kept open in the pool.
            If the session is not requested and used before the
            idle-time has expired, the session is closed.
            If no idle-time is set the default is 30 seconds.";
       }
     }
   }
}
```

This grouping can be found in the NSO model under `/ncs:devices/global-settings/session-pool`, `/ncs:devices/profiles/profile/session-pool` and `/ncs:devices/device/session-pool` to be able to control session pooling for all devices, a group of devices and a specific device respectively.

In addition under `/ncs:devices/global-settings/session-pool/default` it is possible to control the global max size of the session pool, as defined by the following yang snippet:

```
container global-settings {
  tailf:info "Global settings for all managed devices.";
  description
    "Global settings for all managed devices. Some of these
     settings can be overridden per managed device.";

  uses device-profile-parameters {

    ...

    augment session-pool {
```

```
      leaf pool-max-sessions {
        type union {
          type enumeration {
            enum unlimited;
          }
          type uint32;
        }
        description
          "Controls the grand total session count in the pool.
           Independently on how different devices are pooled the grand
           total session count can never exceed this value.
           A Zero value will imply that pooling is disabled for all devices.
           The label 'unlimited' implies that no upper limit exists for
           the number open sessions in the pool";
      }
    }
  }
}
```

Lets illustrate the possibilities with an example configuration of the session pool:

```
ncs# configure
ncs(config)# devices global-settings session-pool idle-time 100
ncs(config)# devices profiles profile small session-pool max-sessions 3
ncs(config-profile-small)# top
ncs(config)# devices device ce* device-profile small
ncs(config-device-ce*)# top
ncs(config)# devices device pe0 session-pool max-sessions 0
ncs(config-device-pe0)# top
ncs(config)# commit
Commit complete.
ncs(config)# exit
```

In the above configuration the default idle-time is set to 100 seconds for all devices. A device profile called `small` is defined which contains a max-session value of 3 sessions, this profile is set on all `ce*` devices. The devices `pe0` has a max-sessions 0 which implies that this device cannot be pooled. Lets connect all devices and see what happens in the session pool:

```
ncs# devices connect suppress-positive-result
ncs# show devices session-pool
        DEVICE             MAX        IDLE
DEVICE  TYPE    SESSIONS  SESSIONS   TIME
-------------------------------------------
ce0     cli     1         3          100
ce1     cli     1         3          100
ce2     cli     1         3          100
ce3     cli     1         3          100
ce4     cli     1         3          100
ce5     cli     1         3          100
pe1     cli     1         unlimited  100
pe2     cli     1         unlimited  100
```

Now we set an upper limit to the maximum number of sessions in the pool. Setting the value to 4 is too small for a real situation but serves the purpose of illustration:

```
ncs# configure
ncs(config)# devices global-settings session-pool pool-max-sessions 4
ncs(config)# commit
Commit complete.
ncs(config)# exit
```

The number of open sessions in the pool will be adjusted accordingly:

```
ncs# show devices session-pool
        DEVICE            MAX        IDLE
DEVICE  TYPE    SESSIONS  SESSIONS   TIME
-------------------------------------------
ce4     cli     1         3          100
ce5     cli     1         3          100
pe1     cli     1         unlimited  100
pe2     cli     1         unlimited  100
```

Device Session Limits

Some devices only allow a small number of concurrent sessions, in the extreme case it only allows one (for example through a terminal server). For this reason NSO can limit the number of concurrent sessions to a device and make operations wait if the maximum number of sessions has been reached.

In other situations, we need to limit the number of concurrent connect attempts made by NSO. For example, the devices managed by NSO talk to the same server for authentication which can only handle a limited number of connections at a time.

The configuration for session limits is found in the tailf-ncs-devices.yang submodel. The following part of the YANG device-profile-parameters grouping controls how the session limits are configured:

```
grouping device-profile-parameters {

  ...

    container session-limits {
      tailf:info "Parameters for limiting concurrent access to the device.";
      leaf max-sessions {
        type union {
          type enumeration {
            enum unlimited;
          }
          type uint32 {
            range "1..max";
          }
        }
        default unlimited;
        description
          "Puts a limit to the total number of concurrent sessions
           allowed for the device. The label 'unlimited' implies that no
           upper limit exists for this device.";
      }
    }

  ...

  }
```

This grouping can be found in the NSO model under `/ncs:devices/global-settings/session-limits`, `/ncs:devices/profiles/profile/session-limits` and `/ncs:devices/device/session-limits` to be able to control session limits for all devices, a group of devices and a specific device respectively.

In addition under `/ncs:devices/global-settings/session-limits` it is possible to control the number of concurrent connect attempts allowed and the maximum time to wait for a device being available to connect.

```
container global-settings {
  tailf:info "Global settings for all managed devices.";
```

```
  description
    "Global settings for all managed devices. Some of these
     settings can be overridden per managed device.";

  uses device-profile-parameters {

  ...

    augment session-limits {
      description
        "Parameters for limiting concurrent access to devices.";
      container connect-rate {
        leaf burst {
          type union {
            type enumeration {
              enum unlimited;
            }
            type uint32 {
              range "1..max";
            }
          }
          default unlimited;
          description
            "The number of concurrent connect attempts allowed.
             For example, the devices managed by NSO talk to the same
             server for authentication which can only handle a limited
             number of connections at a time. Then we can limit
             the concurrency of connect attempts with this setting.";
        }
      }
      leaf max-wait-time {
        tailf:info
          "Max time in seconds to wait for device to be available.";
        type union {
          type enumeration {
            enum unlimited;
          }
          type uint32 {
            range "0..max";
          }
        }
        units "seconds";
        default 10;
        description
          "Max time in seconds to wait for a device being available
           to connect. When the maximum time is reached an error
           is returned. Setting this to 0 means that the error is
           returned immediately.";
      }
    }

  ...

}
```

Tracing Device Communication

It is possible to turn on and off NED traffic tracing. This is often a good way to troubleshoot problems. In order to understand the trace output, a basic prerequisite is a good understanding of the native device interface. For NETCONF devices an understanding NETCONF RPC is a prerequisite. Similarly for CLI NEDs, a good understanding of the CLI capabilities of the managed devices is required.

To turn on southbound traffic tracing, we need to enable the feature and we must also configure a directory where we want the trace output to be written. It is possible to have the trace output in two different formats, `pretty` and `raw`. The format of the data depends on the type of the managed device. For NETCONF devices, the `pretty` mode indents all the XML data for enhanced readability and the `raw` mode does not. Sometimes when the XML is broken, `raw` mode is required to see all the data received. Tracing in `raw` mode will also signal to the corresponding NED to log more verbose tracing information.

To enable tracing do:

```
ncs(config)# devices global-settings trace raw trace-dir .logs
ncs(config)# commit
```

The trace setting only affect new NED connections, so to ensure that we get any tracing data, we can do:

```
ncs(config)# devices disconnect
```

The above command terminates all existing connections.

At this point if you execute a transaction towards one or several devices and then view the trace data.

```
ncs(config)# do file show logs/ned-cisco-ios-ce0.trace
>> 8-Oct-2014::18:23:18.512 CLI CONNECT to ce0-127.0.0.1:10022 as admin (Trace=true)

  *** output 8-Oct-2014::18:23:18.514 ***
-- SSH connecting to host: 127.0.0.1:10022 --
-- SSH initializing session --

  *** input 8-Oct-2014::18:23:18.547 ***

admin connected from 127.0.0.1 using ssh on ncs
...
ce0(config)#
  *** output 8-Oct-2014::18:23:19.428 ***
snmp-server community topsecret RW
```

It is possible to clear all existing trace files through the command

```
ncs(config)# devices clear-trace
```

Finally, it is worth mentioning the trace functionality does not come for free. It is fairly costly to have the trace turned on. Also, there exists no trace log wrapping functionality.

Checking Device Configuration

When managing large networks with NSO a good strategy is to consider the NSO copy of the network configuration to be the main primary copy. All device configuration changes must go through NSO and all other device re-configurations are considered rogue.

NSO does not contain any functionality which disallows rogue re-configurations of managed devices, however it does contain a mechanism whereby it is a very cheap operation to discover if one or several devices have been configured out-of-band.

The underlying mechanism for the cheap check-sync is to compare time-stamps, transaction-ids, hash-sums, etc depending on what the device supports. This in order not to have to read the full configuration to check if the NSO copy is in sync.

The transaction ids are store in CDB and can be viewed as:

```
ncs# show devices device state last-transaction-id
```

```
NAME  LAST TRANSACTION ID
----------------------------------------
ce0   ef3bbd344ef94b3fecec5cb93ac7458c
ce1   48e91db163e294bf5c3978d154922c9
ce2   48e91db163e294bf5c3978d154922c9
ce3   48e91db163e294bf5c3978d154922c9
ce4   48e91db163e294bf5c3978d154922c9
ce5   48e91db163e294bf5c3978d154922c9
ce6   48e91db163e294bf5c3978d154922c9
ce7   48e91db163e294bf5c3978d154922c9
ce8   48e91db163e294bf5c3978d154922c9
p0    -
p1    -
p2    -
p3    -
pe0   -
pe1   -
pe2   1412-581909-661436
pe3   -
```

Some of the devices does not have a transaction-id, this is the case where the NED has not implemented the cheap check-sync mechanism. Although it is called transaction-id, the underlying value in the device can be anything to detect a config change, like for example a time-stamp.

To actually check for consistency, we execute:

```
ncs# devices check-sync
sync-result {
    device ce0
    result in-sync
}
...
sync-result {
    device p1
    result unsupported
}
...
```

Or alternatively for all (or a subset) managed devices,

```
ncs# devices device ce0..3 check-sync
devices device ce0 check-sync
    result in-sync
devices device ce1 check-sync
    result in-sync
devices device ce2 check-sync
    result in-sync
devices device ce3 check-sync
    result in-sync
```

The following YANG grouping is used for the return value from the `check-sync` command:

```
grouping check-sync-result {
    description
      "Common result data from a 'check-sync' action.";

    leaf result {
      type enumeration {
        enum unknown {
          description
            "NCS have no record, probably because no
             sync actions have been executed towards the device.
             This is the initial state for a device.";
```

```
        }
        enum locked {
          tailf:code-name 'sync_locked';
          description
            "The device is administratively locked, meaning that NCS
             cannot talk to it.";
        }
        enum in-sync {
          tailf:code-name 'in-sync-result';
          description
            "The configuration on the device is in sync with NCS.";
        }
        enum out-of-sync {
          description
            "The device configuration is known to be out of sync, i.e.,
             it has been reconfigured out of band.";
        }
        enum unsupported {
          description
            "The device doesn't support the tailf-netconf-monitoring
             module.";
        }
        enum error {
          description
            "An error occurred when NCS tried to check the sync status.
             The leaf 'info' contains additional information.";
        }
      }
    }
  }
```

Comparing Device Configurations

In the previous section we described how we can easily check if a managed device is in sync. If the device is not in sync, we are interested to know what the difference is. The CLI sequence below shows how to modify `ce0` out of band using the ncs-netsim tool. Finally the sequence shows how to do an explicit configuration comparison.

```
$ ncs-netsim cli-i ce0
admin connected from 127.0.0.1 using console on ncs
ce0> enable
ce0# configure
Enter configuration commands, one per line. End with CNTL/Z.
ce0(config)# snmp-server community foobar RW
ce0(config)# exit
ce0# exit
$ ncs_cli -C -u admin

admin connected from 127.0.0.1 using console on ncs
ncs# devices device ce0 check-sync
result out-of-sync
info got: 290fa2b49608df9975c9912e4306110 expected: ef3bbd344ef94b3fecec5cb93ac7458c

ncs# devices device ce0 compare-config
diff
 devices {
     device ce0 {
         config {
             ios:snmp-server {
+                community foobar {
+                    RW;
```

```
+                 }
              }
          }
      }
 }
```

The diff in the above output should be interpreted as: what needs to be done in NSO to become in sync with the device.

Previously in Example 8, “Synchronize from Devices” NSO was brought in sync with the devices by fetching configuration from the devices. In this case where the device has a rogue re-configuration NSO has the correct configuration. In such cases you want to to reset the device configuration to what is store inside NSO.

When you decide to reset the configuration with the copy kept in NSO use the option `dry-run` in conjunction with `sync-to` and inspect what will be sent to the device:

```
ncs# devices device ce0 sync-to dry-run
data
      no snmp-server community foobar RW
ncs#
```

As this is the desired data to send to the device a `sync-to` can now safely be performed.

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

The device configuration should now be in sync with the copy in NSO and `compare-config` ought to yield an empty output:

```
ncs# devices device ce0 compare-config
ncs#
```

Initialize Device

There exists several ways to initialize new devices. The two common ways are to initialize a device from another existing device or to use device-templates.

From other

For example another CE router has been added to our example network. You want to base the configuration of that host on the configuration of the managed device `ce0` which has a valid configuration:

```
ncs(config)# show full-configuration devices device ce0
devices device ce0
 address   127.0.0.1
 port      10022
 ssh host-key ssh-dss
  key-data "AAAAB3NzaC1kc3MAAACBAO9tkTdZgAqJMz8m...
 !
 authgroup default
 device-type cli ned-id cisco-ios-cli-3.8
 state admin-state unlocked
 config
  no ios:service pad
  no ios:ip domain-lookup
  no ios:ip http secure-server
  ios:ip source-route
  ios:interface GigabitEthernet0/1
```

```
  exit
  ios:interface GigabitEthernet0/10
  exit
  ios:interface GigabitEthernet0/11
  exit
  ios:interface GigabitEthernet0/12
  exit
  ios:interface GigabitEthernet0/13
  exit
  ios:interface GigabitEthernet0/14
  exit
....
```

If the configuration is accurate you can create a new managed device based on that configuration as:

Example 17. Instantiate device from other

```
ncs(config)# devices device ce9 address 127.0.0.1 port 10031
ncs(config-device-ce9)# device-type cli ned-id cisco-ios-cli-3.8
ncs(config-device-ce9)# authgroup default
ncs(config-device-ce9)# instantiate-from-other-device device-name ce0
ncs(config-device-ce9)# top
ncs(config)# show configuration
devices device ce9
 address   127.0.0.1
 port      10031
 authgroup default
 device-type cli ned-id cisco-ios-cli-3.8
 config
  no ios:service pad
  no ios:ip domain-lookup
  no ios:ip http secure-server
  ios:ip source-route
  ios:interface GigabitEthernet0/1
  exit
....
ncs(config)# commit
Commit complete.
```

In Example 17, “Instantiate device from other” the commands first creates the new managed device, `ce9` and then populates the configuration of the new device based on the configuration of `ce0`.

This new configuration might not be entirely correct, you can modify any configuration before committing it.

The above concludes the instantiation of a new managed device. The new device configuration is committed and NSO returned OK without the device existing in the network (netsim). Try force a sync to the device:

```
ncs(config)# devices device ce9 sync-to
result false
info Device ce9 is southbound locked
```

The device is `southbound locked`, this is a mode which is used where you can reconfigure a device, but any changes done to it are never sent to the managed device. This will be thoroughly described in the next section. Devices are by default created southbound locked. Default values are not shown if not explicitly requested:

```
(config)# show full-configuration devices device ce9 state | details
devices device ce9
 state admin-state southbound-locked
```

```
!
```

By Template

An other alternative to instantiating a device from the actual working configuration of another device is to have a number of named device templates which manipulates the configuration.

The template tree looks like:

```
submodule tailf-ncs-devices {
  namespace "http://tail-f.com/ns/ncs";
  ...
container devices {
    ........
    list template {
      description
        "This list is used to define named template configurations that
         can be used to either instantiate the configuration for new
         devices, or to apply snippets of configurations to existing
         devices.
         ...
         ";

      key name;
      leaf name {
        description "The name of a specific template configuration";
        type string;
      }
      list ned-id {
        key id;
        leaf id {
          type identityref {
            base ned:ned-id;
          }
        }
        container config {
          tailf:mount-point ncs-template-config;
          tailf:cli-add-mode;
          tailf:cli-expose-ns-prefix;
          description
            "This container is augmented with data models from the devices.";
        }
      }
    }
```

The tree for device templates is generated from all device YANG models. All constrains are removed and the data type of all leafs is changed to `string`

A device template is created by setting the desired data in the configuration. The created device template is stored in NSO CDB.

Example 18. Create ce-initialize template

```
ncs(config)# devices template ce-initialize ned-id cisco-ios-cli-3.8 config
ncs(config-config)# no ios:service pad
ncs(config-config)# no ios:ip domain-lookup
ncs(config-config)# ios:ip dns server
ncs(config-config)# no ios:ip http server
ncs(config-config)# no ios:ip http secure-server
ncs(config-config)# ios:ip source-route true
ncs(config-config)# ios:interface GigabitEthernet 0/1
```

```
ncs(config- GigabitEthernet-0/1)# exit
ncs(config-config)# ios:interface GigabitEthernet 0/2
ncs(config- GigabitEthernet-0/2)# exit
ncs(config-config)# ios:interface GigabitEthernet 0/3
ncs(config- GigabitEthernet-0/3)# exit
ncs(config-config)# ios:interface Loopback 0
ncs(config-Loopback-0)# exit
ncs(config-config)# ios:snmp-server community public RO
ncs(config-community-public)# exit
ncs(config-config)# ios:snmp-server trap-source GigabitEthernet 0/2
ncs(config-config)# top
ncs(config)# commit
```

The device template created in Example 18, “Create ce-initialize template” can now be used to initialize single devices or device groups, the section called “Device Groups”

In the following CLI session a new device `ce10` is created:

```
ncs(config)# devices device ce10 address 127.0.0.1 port 10032
ncs(config-device-ce10)# device-type cli ned-id cisco-ios-cli-3.8
ncs(config-device-ce10)# authgroup default
ncs(config-device-ce10)# top
ncs(config)# commit
```

Initialize the newly created device `ce10` with the device template `ce-initialize`:

```
ncs(config)# devices device ce10 apply-template template-name ce-initialize
apply-template-result {
    device ce10
    result no-capabilities
    info No capabilities found for device: ce10. Has a sync-from the device
         been performed?
}
```

When initializing devices NSO does not have any knowledge about the capabilities of the device, no connect has been done. This can be overridden by the option `accept-empty-capabilities`

```
ncs(config)# devices device ce10 \
apply-template template-name ce-initialize accept-empty-capabilities
apply-template-result {
    device ce10
    result ok
}
```

Inspect the changes made by the template `ce-initialize`

```
ncs(config)# show configuration
devices device ce10
 config
  ios:ip dns server
  ios:interface GigabitEthernet0/1
  exit
  ios:interface GigabitEthernet0/2
  exit
  ios:interface GigabitEthernet0/3
  exit
  ios:interface Loopback0
  exit
  ios:snmp-server community public RO
  ios:snmp-server trap-source GigabitEthernet0/2
 !
!
```

Device Templates

Note This section shows how *Device-templates* can be used to create and change device configuration. See Chapter 8, *Templates* in *Development Guide* for other ways of using templates.

device-templates are part of the NSO configuration. *device-templates* are created and changed in the tree `/devices/template/config` the same way as any other configuration data and are affected by rollbacks and upgrades. *Device-templates* can only manipulate configuration data in the `/devices/device/config` tree i.e. only device data.

The `$NCS_DIR/examples.ncs/service-provider/mpls-vpn` example comes with a pre-populated template for SNMP settings.

```
ncs(config)# show full-configuration devices template
devices template snmp1
 ned-id cisco-ios-cli-3.8
  config
   ios:snmp-server community {$COMMUNITY}
    RO
   !
  !
 !
 ned-id cisco-iosxr-cli-3.5
  config
   cisco-ios-xr:snmp-server community {$COMMUNITY}
    RO
   !
  !
 !
 ned-id juniper-junos-nc-3.0
  config
   junos:configuration snmp community {$COMMUNITY}
    authorization read-only
   !
  !
 !
!
```

Note The variable *$DEVICE* is used internally by NSO and can not be used in a template.

Templates can be created like any configuration data and use the CLI tab completion to navigate. Variables can be used instead of hard-coded values. In the template above the community string is a variable. The template can cover several device-types/NEDs, by making use of the namespace information. This will make sure that only devices modeled with this particular namespace will be affected by this part of the template. Hence, it is possible for one template to handle a multitude of devices from various manufacturers.

A template can be applied to a device, a device-group and a range of devices. It can be used as shown in the section called “By Template” to create the day zero config for a newly created device.

Applying the `snmp1` template, providing a value for the `COMMUNITY` template variable:

```
ncs(config)# devices device ce2 apply-template template-name \
      snmp1 variable { name COMMUNITY value 'FUZBAR' }
```

```
ncs(config)# show configuration
devices device ce2
 config
  ios:snmp-server community FUZBAR RO
 !
!
ncs(config)# commit dry-run outformat native
native {
    device {
        name ce2
        data snmp-server community FUZBAR RO
    }
}
ncs(config)# commit
Commit complete.
```

The result of applying the template:

```
ncs(config)# show full-configuration devices device ce2 config\
   ios:snmp-server
devices device ce2
 config
  ios:snmp-server community FUZBAR RO
 !
!
```

Tags

The default operation for templates is to merge the configuration. Tags can be added to templates to have the template `merge`, `replace`, `delete`, `create` or `nocreate` configuration. A tag is inherited to its sub-nodes until a new tag is introduced.

- *merge:* Merge with a node if it exists, otherwise create the node. This is the default operation if no operation is explicitly set.
- *replace:* Replace a node if it exists, otherwise create the node.
- *create:* Creates a node. The node can not already exist.
- *nocreate:* Merge with a node if it exists. If it does not exist, it will *not* be created.

Example on how to set a tag:

```
ncs(config)# tag add devices template snmp1 ned-id cisco-ios-cli-3.8 config\
 ios:snmp-server community {$COMMUNITY} replace
```

Displaying Tags information::

```
ncs(config)# show configuration
devices template snmp1
 ned-id cisco-ios-cli-3.8
  config
   ! Tags: replace
   ios:snmp-server community {$COMMUNITY}
   !
  !
 !
!
```

Debug

By adding the CLI pipe flag `debug template` when applying a template, the CLI will output detailed information on what is happening when the template is being applied:

```
ncs(config)# devices device ce2 apply-template template-name \
      snmp1 variable { name COMMUNITY value 'FUZBAR' } | debug template
Operation 'merge' on existing node: /devices/device[name='ce2']
The device /devices/device[name='ce2'] does not support
namespace 'http://tail-f.com/ned/cisco-ios-xr' for node "'snmp-server'"
Skipping...
The device /devices/device[name='ce2'] does not support
namespace 'http://xml.juniper.net/xnm/1.1/xnm' for node "configuration"
Skipping...
Variable $COMMUNITY is set to "FUZBAR"
Operation 'merge' on non-existing node:
/devices/device[name='ce2']/config/ios:snmp-server/community[name='FUZBAR']
Operation 'merge' on non-existing node:
/devices/device[name='ce2']/config/ios:snmp-server/community[name='FUZBAR']/RO
```

Oper State and Admin State

NSO differentiates between oper state and admin state for a managed device. Oper state is the actual state of the device. We have chosen to implement a very simple oper state model. A managed device oper state is either enabled or disabled. Oper state can be mapped to an alarm for the device. If the device is disabled, we may have additional error information. For example the `ce9` device created from another device and `ce10` created with a device template in the previous section is disabled, no connection has been established with the device, so its state is completely unknown:

```
ncs# show devices device ce9 state oper-state
state oper-state disabled
```

Or slight more interesting, CLI usage:

```
ncs# show devices device state oper-state
      OPER
NAME  STATE
----------------
ce0   enabled
ce1   enabled
ce10  disabled
ce2   enabled
ce3   enabled
ce4   enabled
ce5   enabled
ce6   enabled
ce7   enabled
ce8   enabled
ce9   disabled
p0    enabled
p1    enabled
p2    enabled
p3    enabled
pe0   enabled
pe1   enabled
pe2   enabled
pe3   enabled

ncs# show devices device ce0..9 state oper-state
      OPER
NAME  STATE
----------------
ce0   enabled
ce1   enabled
ce2   enabled
ce3   enabled
```

```
ce4   enabled
ce5   enabled
ce6   enabled
ce7   enabled
ce8   enabled
ce9   disabled
```

If you manually stop a managed device, for example `ce0`, NSO doesn't immediately indicate that. NSO may have an active SSH connection to the device, but the device may voluntarily choose to close its end of that (idle) SSH connection. Thus the fact that a socket from the device to NSO is closed by the managed device doesn't indicate anything. The only certain method NSO has to decide a managed device is non-operational - from the point of view of NSO - is NSO cannot SSH connect to it. If you manually stop managed device `ce0`, you still have:

```
$ ncs-netsim stop ce0
DEVICE ce0 STOPPED
$ ncs_cli -C -u admin
ncs# show devices device ce0 state oper-state
state oper-state enabled
```

NSO cannot draw any conclusions from the fact that a managed device closed its end of the SSH connection. It may have done so because it decided to time out an idle SSH connection. Whereas if NSO tried to initiate any operations towards the dead device, the device will be marked as oper state `disabled`:

```
ncs(config)# devices device ce0 config ios:snmp-server contact joe@acme.com
ncs(config-config)# commit
Aborted: Failed to connect to device ce0: connection refused: Connection refused
ncs(config-config)# *** ALARM connection-failure: Failed to
connect to device ce0: connection refused: Connection refused
```

Now, NSO has failed to connect to it, NSO knows that `ce0` is dead:

```
ncs# show devices device ce0 state oper-state
state oper-state disabled
```

This concludes the oper state discussion. Next state to be illustrated is the admin state. The admin state is what the operator configures, this is the desired state of the managed device.

In `tailf-ncs.yang` we have the following configuration definition for admin state:

Example 19. tailf-ncs-devices.yang - Admin state

```
submodule tailf-ncs-devices {
  ....

  typedef admin-state {
    type enumeration {
      enum locked {
        description
          "When a device is administratively locked, it is not possible
           to modify its configuration, and no changes are ever
           pushed to the device.";
      }
      enum unlocked {
        description
          "Device is assumed to be operational.
           All changes are attempted to be sent southbound.";
      }
      enum southbound-locked {
        description
          "It is possible to configure the device, but
```

```
           no changes are sent to the device. Useful admin mode
           when pre provisioning devices. This is the default
           when a new device is created.";
      }
      enum config-locked {
        description
          "It is possible to send live-status commands or RPCs
           but it is not possible to modify the configuration
           of the device.";
      }
    }
  }

  ....
  container devices {
     ....
     container state {
        ....
        leaf admin-state {
          type admin-state;
          default southbound-locked;
        }

        leaf admin-state-description {
          type string;
          description
            "Reason for the admin state.";

        }
```

In Example 19, “tailf-ncs-devices.yang - Admin state” you can see the four different admin states for a managed device as defined in the YANG model.

- `locked` - This means that all changes to the device are forbidden. Any transaction which attempts to manipulate the configuration of the device will fail. It is still possible to read the configuration of the device.
- `unlocked` -This is the state a device is set into when the device is operational. All changes to the device are attempted to be sent southbound.
- `southbound-locked` - This is the default value. It means that it is possible to manipulate the configuration of the device but changes done to the device configuration are never pushed to the device. This mode is useful during e.g. pre-provisioning, or when we instantiate new devices.
- `config-locked` - This means that any transaction which attempts to manipulate the configuration of the device will fail. It is still possible read the configuration of the device and send live-status commands or RPCs.

Configuration Source

NSO manages a set of devices which are given to NSO through any means like CLI, inventory system integration through XML APIs, or configuration files at startup. The list of devices to manage in an overall integrated network management solution is shared between different tools and therefore it is important to keep an authoritative database of this and share it between different tools including NSO. The purpose of this part is to identify the source of the population of managed devices. The source attribute should indicate the source of the managed device like "inventory", "manual", "EMS".

Example 20. tailf-ncs-devices.yang - source

```
submodule tailf-ncs-devices {
  ...
```

```
container source {
  tailf:info "How the device was added to NCS";
  leaf added-by-user {
    type string;
  }
  leaf context {
    type string;
  }
  leaf when {
    type yang:date-and-time;
  }
  leaf from-ip {
    type inet:ip-address;
  }
  leaf source {
    type string;
    reference "TMF518 NRB Network Resource Basics";
  }
}
```

These attributes should be automatically set by the integration towards the inventory source rather then manipulated manually.

- `added-by-user` - Identify the user which loaded the managed device.
- `context` in what context was the device loaded.
- `when` - when the device was added to NSO.
- `from-ip` - from which IP the load activity was run..
- `source` - identify the source of the managed device such as the inventory system name or the name of the source file.

Capabilities, Modules and Revision Management

The NETCONF protocol mandates that the first thing both the server and the client has to do is to send its list of NETCONF capabilities in the `<hello>` message. A capability indicates what the peer can actually do. For example the `validate:1.0` indicates that the server can validate a proposed configuration change, whereas the capability `http://acme.com/if` indicates the device implements the `http://acme.com` proprietary capability.

The NEDs report the capabilities for the devices at connection time. The NEDs also load the YANG modules for NSO. For a NETCONF/YANG device all this is straight-forward, for non NETCONF devices the NEDs does the translation.

The capabilities announced by a device also contain the YANG version 1 modules supported. In addition to this, YANG version 1.1 modules are advertised in the YANG library module on the device. NSO checks both the capabilities and the YANG library to find out which YANG modules a device supports.

The capabilities and modules detected by NSO are available in two different lists, `/devices/device/capability` and `devices/device/module`. The `capability` list contains all capabilities announced and all YANG modules in the YANG library. The `module` list contains all YANG modules announced that are also supported by the NED in NSO.

```
ncs# show devices device ce0 capability
capability urn:ietf:params:netconf:capability:with-defaults:1.0?basic-mode=trim
capability urn:ios
 revision 2015-03-16
 module   tailf-ned-cisco-ios
capability urn:ios-stats
 revision 2015-03-16
```

```
 module   tailf-ned-cisco-ios-stats

ncs#  show devices device ce0 capability module
NAME                        REVISION    FEATURE  DEVIATION
-----------------------------------------------------------
tailf-ned-cisco-ios         2015-03-16  -        -
tailf-ned-cisco-ios-stats   2015-03-16  -        -
```

NSO can be used to handle all or some of the YANG configuration modules for a device. A device may announce several modules through its capability list which NSO ignores. NSO will only handle the YANG modules for a device which are loaded (and compiled through **ncsc --ncs-compile-bundle**) or **ncsc --ncs-compile-module**) all other modules for the device are ignored. If you require a situation where NSO is entirely responsible for a device so that complete device backup/configurations are stored in NSO you must ensure NSO indeed has support for all modules for the device. It is not possible to automate this process since a capability URI doesn't necessary indicate actual configuration.

Discovery of a NETCONF Device

When a device is added to NSO its NED id must be set. For a NETCONF device, it is possible to configure the generic NETCONF NED id `netconf` (defined in the YANG module tailf-ncs-ned). If this NED id is configured, we can then ask NSO to connect to the device and then check the `capability` list to see which modules this device implements.

```
ncs(config)# devices device foo address 127.0.0.1 port 12033 authgroup default
ncs(config-device-foo)# device-type netconf ned-id netconf
ncs(config-device-foo)# state admin-state unlocked
ncs(config-device-foo)# commit
Commit complete.
ncs(config-device-foo)# exit
ncs(config)# exit
ncs# devices fetch-ssh-host-keys device foo
fetch-result {
    device foo
    result updated
    fingerprint {
        algorithm ssh-rsa
        value 14:3c:79:87:69:8e:e2:f0:6d:43:07:8c:89:41:fd:7f
    }
}
ncs# devices device foo connect
result true
info (admin) Connected to foo - 127.0.0.1:12033
ncs# show devices device foo capability
capability :candidate:1.0
capability :confirmed-commit:1.0
...
capability http://xml.juniper.net/xnm/1.1/xnm
 module junos
capability urn:ietf:params:xml:ns:yang:ietf-yang-types
 revision 2013-07-15
 module   ietf-yang-types
capability urn:juniper-rpc
 module junos-rpc
...
```

We can also check which modules the loaded NEDs supports. Then we can pick the most suitable NED and configure the device with this NED id.

```
ncs# show devices ned-ids
ID                     NAME                          REVISION
```

```
-----------------------------------------------------------
cisco-ios-xr-v2       tailf-ned-cisco-ios-xr        -
                      tailf-ned-cisco-ios-xr-stats  -
lsa-netconf
netconf
snmp
alu-sr-cli-3.4        tailf-ned-alu-sr              -
                      tailf-ned-alu-sr-stats        -
cisco-ios-cli-3.8     tailf-ned-cisco-ios           -
                      tailf-ned-cisco-ios-stats     -
cisco-iosxr-cli-3.5   tailf-ned-cisco-ios-xr        -
                      tailf-ned-cisco-ios-xr-stats  -
juniper-junos-nc-3.0  junos                         -
                      junos-rpc                     -
ncs# config
Entering configuration mode terminal
ncs(config)# devices device foo device-type netconf ned-id juniper-junos-nc-3.0
ncs(config-device-foo)# commit
Commit complete.
```

Configuration Datastore Support

NSO works best if the managed devices support the NETCONF candidate configuration datastore. However, NSO reads the capabilities of each managed devices and executes different sequences of NETCONF commands towards different types of devices.

For implementations of the NETCONF protocol that do not support the candidate datastore, and in particular devices that do not support NETCONF commit with a timeout, NSO tries to do the best of the situation.

NSO divides devices in the following groups.

- *start_trans_running* - This mode is used for devices that support the Tail-f proprietary transaction extension defined by `http://tail-f.com/ns/netconf/transactions/1.0`. Read more on this in the Tail-f ConfD user guide. In principle it's a means to - over the NETCONF interface - control transaction processing towards the running data store. This may be more efficient than going through the candidate data store. The downside is that it is Tail-f proprietary non-standardized technology.
- *lock_candidate* - This mode is used for devices that support the candidate data store but disallow direct writes to the running data store.
- *lock_reset_candidate* - This mode is used for devices that support the candidate data and also allow direct writes to the running data store. This is the default mode for Tail-f ConfD NETCONF server. Since the running data store is configurable, we must, prior to each configuration attempt, copy all of running to the candidate. (ConfD has optimized this particular usage pattern, so this is a very cheap operation for ConfD)
- *startup* - This mode is used for devices that have writable running, no candidate but do support the startup data store. This is the typical mode for Cisco like devices.
- *running-only* - This mode is used for devices that only support writable running.
- *NED* - The transaction is controlled by a Network Element Driver. The exact transaction mode depends on the type of the NED.

Which category NSO chooses for a managed device depends on which NETCONF capabilities the devices sends to NSO in its NETCONF *hello* message. You can see in the CLI what NSO has decided for a device as in:

```
ncs# show devices device ce0 state transaction-mode
```

```
state transaction-mode ned
ncs# show devices device pe2 state transaction-mode
state transaction-mode lock-candidate
```

NSO talking to ConfD device running in its standard configuration, thus *lock-reset-candidate*

Another important discriminator between managed devices is whether they support the confirmed commit with a timeout capability, i.e. the *confirmed-commit:1.0* standard NETCONF capability. If a device supports this capability, NSO utilizes it. This is the case with for example Juniper routers.

If a managed device does not support this capability, NSO attempts to the best it can.

This is how NSO handles common failure scenarios:

- The operator aborts the transaction, or NSO looses the SSH connection to another managed device which is also participating in the same network transaction.

  If the device does support the *confirmed-commit* capability, NSO aborts the outstanding yet-uncommitted transaction simply by closing the SSH connection.

  When the device does not support the *confirmed-commit* capability, NSO has the reverse diff and simply sends the precise undo information to the device instead.
- The device rejects the transaction in the first place, i.e. the NSO attempt to modify its running data store. This is an easy case since NSO then simply aborts the transaction as a whole in the initial `commit confirmed [time]` attempt.
- NSO looses SSH connectivity to the device during the timeout period. This is a real error case and the configuration is now in an unknown state. NSO will abort the entire transaction, but the configuration of the failing managed device is now probably in error. The correct procedure once network connectivity has been restored to the device is to sync it in direction from NSO to the device. The NSO copy of the device configuration will be what was configured prior to the failed transaction.

Thus, even if not all participating devices have first class NETCONF server implementations, NSO will attempt to fake the *confirmed-commit* capability.

Action Proxy

When the managed device defines top level NETCONF RPCs or alternatively define *tailf:action* points inside the YANG model, these RPCs and actions are also imported into the data model that resides in NSO.

For example the Juniper NED comes with a set of JunOS RPCs defined in: `$NCS_DIR/packages/neds/juniper-junos/src/yang/junos-rpc.yang`

```
module junos-rpc {
  ...
  rpc request-package-add {
  ...
  rpc request-reboot {
  ...
  rpc get-software-information {
  ...
  rpc ping {
```

Thus, since all RPCs and actions from the devices are accessible through the NSO data model, these actions are also accessible through all NSO northbound APIs, REST, JAVA MAAPI etc. Hence it is possible to - from user scripts/code - invoke actions and RPCs on all managed devices. The RPCs are augmented below an RPC container:

```
ncs(config)# devices device pe2 rpc rpc-
Possible completions:
  rpc-get-software-information  rpc-idle-timeout  rpc-ping \
  rpc-request-package-add  rpc-request-reboot

ncs(config)# devices device pe2 rpc \
rpc-get-software-information get-software-information brief
```

In the simulated environment of the mpls-vpn example these RPCs might not have been implemented.

Device Groups

The NSO device manager has a concept of groups of devices. A group is nothing more than a named group of devices. What makes this interesting is that we can invoke several different actions on the group, thus implicitly invoking the the action on all members in the group. This is especially interesting for the *apply-template* action.

The definition of device groups reside at the same layer in the NSO data model as the device list, thus we have:

Example 21. Device Groups

```
submodule tailf-ncs-devices {
  namespace "http://tail-f.com/ns/ncs";
  ...
  container devices {
     .....
    list device {
     ...
     }
    list device-group {
      key name;
      leaf name {
        type string;
      }
      description
        "A named group of devices, some actions can be
         applied to an entire  group of devices, for example
         apply-template, and the sync actions.";
      leaf-list device-name {
        type leafref {
          path "/devices/device/name";
        }
      }
      leaf-list device-group {
        type leafref {
          path "/devices/device-group/name";
        }
        description
          "A list of device groups contained in this device group.

           Recursive definitions are not valid.";
      }
      leaf-list member {
        type leafref {
          path "/devices/device/name";
        }
        config false;
        description
          "The current members of the device-group.  This is a flat list
           of all the devices in the group.";
```

```
      }
      uses connect-grouping ;
      uses sync-grouping;
      uses check-sync-grouping;
      uses apply-template-grouping;
    }
  }
}
```

The MPLS VPN example comes with a couple of pre-defined device-groups:

```
ncs(config)# show full-configuration devices device-group
devices device-group C
 device-name [ ce0 ce1 ce3 ce4 ce5 ce6 ce7 ce8 ]
!
devices device-group P
 device-name [ p0 p1 p2 p3 ]
!
devices device-group PE
 device-name [ pe0 pe1 pe2 pe3 ]
!
```

Device groups are created like below:

Example 22. Create device group

```
ncs(config)# devices device-group my-group device-name ce0
ncs(config-device-group-my-group)# device-name pe
Possible completions:
  pe0  pe1  pe2  pe3
ncs(config-device-group-my-group)# device-name pe0
ncs(config-device-group-my-group)# device-name p0
ncs(config-device-group-my-group)# commit
```

Device-groups can reference other device-groups. There is a an operational attribute that flattens all members in the group. The CLI sequence below adds the PE group to my-group. Then it shows the configuration of that group followed by the status for this group. The status for the group contains a `members` attribute that lists all device members.

```
ncs(config-device-group-my-group)# device-group PE
ncs(config-device-group-my-group)# commit

ncs(config)# show full-configuration devices device-group my-group
devices device-group my-group
 device-name  [ ce0 p0 pe0 ]
 device-group [ PE ]
!
ncs(config)# exit

ncs# show devices device-group my-group
NAME      MEMBER                      INDETERMINATES  CRITICALS  MAJORS  MINORS  WARNINGS
-------------------------------------------------------------------------------------------
my-group  [ ce0 p0 pe0 pe1 pe2 pe3 ]  0               0          1       0       0
```

Once you have a group, you can sync and check-sync the entire group.

```
ncs# devices device-group C sync-to
```

However, what make device groups really interesting is the ability to apply a template to a group. You can use the pre-populated templates to apply SNMP settings to device-groups.

```
ncs(config)# devices device-group C apply-template \
```

```
template-name snmp1 variable { name COMMUNITY value 'cinderella' }
ncs(config)# show configuration
devices device ce0
 config
  ios:snmp-server community cinderella RO
 !
!
devices device ce1
 config
  ios:snmp-server community cinderella RO
 !
!
...
ncs(config)# commit
```

Policies

Policies allows you to specify network wide constraints that always must be true. If someone tries to apply a configuration change over any northbound interface that would evaluate to false the configuration change is rejected by NSO. Policies can be of type warning means that it is possible to override them, or error which cannot be overridden.

Assume you would like to enforce all CE routers to have a Gigabit interface `0/1`.

Example 23. Policies

```
ncs(config)# policy rule gb-one-zero
ncs(config-rule-gb-one-zero)# foreach /ncs:devices/device[starts-with(name,'ce')]/config
ncs(config-rule-gb-one-zero)# expr ios:interface/ios:GigabitEthernet[ios:name='0/1']
ncs(config-rule-gb-one-zero)# warning-message "{../name} should have 0/1 interface"
ncs(config-rule-gb-one-zero)# commit
zork(config-rule-gb-one-zero)# top
zork(config)# !
ncs(config)# show full-configuration policy
policy rule gb-one-zero
 foreach         /ncs:devices/device[starts-with(name,'ce')]/config
 expr            ios:interface/ios:GigabitEthernet[ios:name='0/1']
 warning-message "{../name} should have 0/1 interface"
!
ncs(config)# no devices device ce0 config ios:interface GigabitEthernet 0/1
ncs(config)# validate
Validation completed with warnings:
  ce0 should have 0/1 interface
ncs(config)# no devices device ce1 config ios:interface GigabitEthernet 0/1
ncs(config)# validate
Validation completed with warnings:
  ce1 should have 0/1 interface
  ce0 should have 0/1 interface
ncs(config)# commit
The following warnings were generated:
  ce1 should have 0/1 interface
  ce0 should have 0/1 interface
Proceed? [yes,no] yes
Commit complete.
```

As seen in Example 23, “Policies” a policy rule has (an optional) for each statement and a mandatory expression and error-message. The `foreach` statement evaluates to a node set, the expression is then evaluated on each node. So in this example the expression would be evaluated for every device in NSO which begins with ce. The name variable in the warning-message refers to a leaf available from the for-each node-set.

Validation is always performed at commit but can also be requested interactively.

Note any configuration can be activated or deactivated. This means that in order to temporarily turn off a certain policy you can deactivate it. Note also that if the configuration was changed by any other means than NSO by local tools to the device like a CLI, a **devices sync-from** operation might fail if the device configuration violates the policy.

Commit Queue

One of the strengths of NSO is the concept of "network wide transactions". When you commit data to NSO that spans multiple devices in the `/ncs:devices/device` tree, NSO will - within the NSO transaction - commit the data on all devices or none, keeping the network consistent with CDB. The NSO transaction doesn't return until all participants have acknowledged the proposed configuration change. The downside of this is that the slowest device in each transaction limits the overall transactional throughput in NSO. Such things as out of sync checks, network latency, calculation of changes sent southbound or device deficiencies all affects the throughput.

Typically when automation software north of NSO generates network change requests it may very well be the case more requests arrive than what can be handled. In NSO deployments scenarios where you wish to have higher transactional throughput than what is possible using "network wide transactions", you can use the commit queue instead. The goal of the commit queue is to increase the transactional throughput of NSO while keeping an eventual consistency view of the database. With the commit queue, NSO will compute the configuration change for each participating device, put it in an outbound queue item and immediately return. The queue is then independently run.

Another use case where you can use the commit queue is when you wish to push a configuration change to a set of devices and don't care about whether all devices accept the change or not. You do not want the default behavior for transactions which is to reject the transaction as a whole if one or more participating devices fail to process its part of the transaction.

An example of the above could be you wish to set a new NTP server on all managed devices in our entire network, if one or more devices currently are non operational, you still want to push out the change. You also want the change automatically pushed to the non operational devices once they go live again.

The big upside of this scheme is that the transactional throughput through NSO is considerably higher. Also transient devices are handled better. The downsides are:

1. If a device rejects the proposed change, NSO and the device are now *out of sync* until any error recovery is performed. Whenever this happens, an NSO alarm (called commit-through-queue-failed) is generated.
2. While a transaction remains in the queue, i.e it has been accepted for delivery by NSO but is not yet delivered, the view of the network in NSO is not (yet) correct. Eventually though, the queued item will be delivered, thus achieving eventual consistency.

To facilitate the two use cases of the commit queue the outbound queue item can be either in an atomic or non-atomic mode.

In atomic mode the outbound queue item will push all configuration changes concurrently once there are no intersecting devices ahead in the queue. If any device rejects the proposed change, all device configuration changes in the queue item will be rejected as a whole, leaving the network in a consistent state. The atomic mode also allows for automatic error recovery to be performed by NSO.

In the non-atomic mode the outbound queue item will push configuration changes for a device whenever all occurrences of it is completed or it doesn't exist ahead in the queue. The drawback to this mode is that there are no automatic error recovery that can be performed by NSO.

In the following sequences the simulates device ce0 is stopped to illustrate the commit queue. This can be achieved by the following sequence including returning to the NSO CLI config mode:

```
$ ncs-netsim stop ce0
DEVICE ce0 STOPPED
$ ncs_cli -C -u admin

admin connected from 127.0.0.1 using console on ncs
ncs# config
```

By default the commit queue is turned off. You can configure NSO to run a transaction, device or device group through the commit queue in a number of different ways, either by providing a flag to the `commit` command as:

```
ncs(config)# commit commit-queue
Possible completions:
  async    Commit through commit queue and return immediately
  bypass   Bypass commit-queue when queue is enabled by default
  sync     Commit through commit queue and wait for reply
ncs(config)# commit commit-queue async
```

or by configuring NSO to always run all transactions through the commit queue as in:

```
ncs(config)# devices global-settings commit-queue enabled-by-default
[false,true] (false): true
ncs(config)# commit
```

or by configuring a number of devices to run through the commit queue as default:

```
ncs(config)# devices device ce0..2 commit-queue enabled-by-default
[false,true] (false): true
ncs(config)# commit
```

When enabling the commit queue as default on a per device/device group basis, a NSO transaction will compute the configuration change for each participating device, put the devices enabled for the commit queue in the outbound queue and then proceed with the normal transaction behaviour for those devices not commit queue enabled. The transaction will still be successfully committed even if some of the devices added to the outbound queue will fail. If the transaction fails in the validation phase the entire transaction will be aborted, including the configuration change for those devices added to commit queue. If the transaction fails after the validation phase, the configuration change for the devices in the commit queue will still be delivered.

Do some change and commit through the commit queue:

Example 24. Commit through Commit Queue

```
ncs(config)# devices device ce0..2 config ios:snmp-server \
    trap-source GigabitEthernet 0/1
ncs(config-config)# commit
commit-queue-id 9494446997
Commit complete.
ncs(config-config)# *** ALARM connection-failure: Failed to
connect to device ce0: connection refused: Connection refused
```

Commit Queue Scheduling

In Example 24, "Commit through Commit Queue" the commit affected three devices, `ce0`, `ce1` and `ce2`. If you immediately would have launched yet another transaction, as in:

```
ncs(config)# devices device ce0 config ios:interface GigabitEthernet 0/25
ncs(config-if)# commit
```

```
commit-queue-id 9494530158
Commit complete.
ncs(config-if)# *** ALARM commit-through-queue-blocked:
Commit Queue item 9494530158 is blocked because qitem 9494446997
cannot connect to ce0
```

the second one, manipulating an interface of `ce2`, that transaction would have been queued instead of immediately launched. The idea here is to queue entire transactions that touch any device which has anything queued ahead in the queue.

Each transaction committed through the queues becomes a *queue item*. A queue item has an id number. A bigger number means that its scheduled later. Each queue item waits for something to happen. A queue item is in either of three states.

1. *waiting* - The queue item is waiting for other queue items to finish. This is because the *waiting* queue item has participating devices that are part of other queue items, ahead in the queue. It is waiting for a set of devices, to not occur ahead of itself in the queue.
2. *executing* - The queue item is currently being processed. Multiple queue item can run currently as long as they don't share any managed devices. Transient errors might be present. These errors occur when NSO fails to communicate with some of the devices. The errors are shown in the leaf-list `transient-errors`. Retries will take place at intervals specified in `/ncs:devices/global-settings/commit-queue/retry-timeout`. Examples of transient errors are connection failures and that the changes are rejected due to the device being locked. Transient errors are potentially bad, since the queue might grow if new items are added, waiting for the same device.
3. *locked* - This queue item is locked and will not be processed until it has been unlocked, see the action `/ncs:devices/commit-queue/queue-item/unlock`. A locked queue item will block all subsequent queue items which are using any device in the locked queue item.

Viewing and Manipulating the Commit Queue

You can view the queue in the CLI. There are three different view modes, *summary*, *normal* and *detailed*. Depending on the output, both the *summary* and the *normal* look good:

Example 25. Viewing queue items

```
ncs# show devices commit-queue | notab
devices commit-queue queue-item 9494446997
 age             144
 status          executing
 kilo-bytes-size 1
 devices         [ ce0 ce1 ce2 ]
 transient-errors [ ce0 ]
 is-atomic       true
devices commit-queue queue-item 9494530158
 age             61
 status          blocked
 kilo-bytes-size 1
 devices         [ ce0 ]
 waiting-for     [ ce0 ]
 is-atomic       true
```

The *age* field indicated how many seconds a queue item has been in the queue.

You can also view the queue items in *detailed* mode:

```
ncs# show devices commit-queue queue-item 9494530158 details | notab
devices commit-queue queue-item 9494530158
 age            278
 status         blocked
```

```
kilo-bytes-size 1
devices         [ ce0 ]
waiting-for     [ ce0 ]
is-atomic       true
modification ce0
 data       <interface xmlns="urn:ios">
              <GigabitEthernet>
                <name>0/25</name>
              </GigabitEthernet>
            </interface>

 local-user admin
```

The queue items are stored persistently, thus if NSO is stopped and restarted, the queue remains the same. Similarly, if NSO runs in HA (High Availability) mode, the queue items are replicated, ensuring the queue is processed even in case of failover.

Note The commit queue is disabled when both HA is enabled, and its HA role is `none`, i.e., not `primary` or `secondary`. See the section called "Mode of operation" in *Administration Guide* for more details.

A number of useful actions are available to manipulate the queue:

1 **devices commit-queue add-lock device [...]** This adds a fictive queue-item to the commit-queue. Any queue item, affecting the same devices, which is entering the commit-queue will have to wait for this lock item to be unlocked or deleted. If no devices are specified, all devices in NSO are locked.

2 **devices commit-queue clear** This action clears the entire queue. All devices present in the commit queue will after this action has executed be out of sync. The `clear` action is a rather blunt tool and is not recommended to be used in any normal use case.

3 **devices commit-queue prune device [...]** This action prunes all specified devices from all queue items in the commit queue. The affected devices will, after this action has been executed, be out of sync. Devices which are currently being committed to will not be pruned, unless the `force` option is used. Atomic queue items will not be affected, unless all devices in it are pruned.

   The `force` option will brutally kill an ongoing commit. This could leave the device in a bad state. It is not recommended in any normal use case.

4 **devices commit-queue set-atomic-behaviour atomic [true,false]** This actions sets the atomic behaviour of all queue items. If these are set to false, the devices contained in these queue items can start executing if the same devices in other non-atomic queue items ahead of it in the queue are completed. If set to true, the atomic integrity of these queue items are preserved.

5 **devices commit-queue wait-until-empty** This action waits until the commit queue is empty. Default is to wait `infinity`. A `timeout` can be specified to wait for a number of seconds. The result is *empty* if the queue is empty or *timeout* if there are still items in the queue to become processed.

6 **devices commit-queue queue-item [id] lock** This action puts a lock on an existing queue item. A locked queue item will not start executing until it has been unlocked.

7 **devices commit-queue queue-item [id] unlock** This action unlocks a locked queue item. Unlocking a queue item which is not locked is silently ignored.

8 **devices commit-queue queue-item [id] delete** This action deletes a queue item from the queue. If other queue items are waiting for this (deleted) item, they will all automatically start to run. The devices of the deleted queue item will, after the action has executed, by out of sync if they haven't start executing. Any error-option set for the queue item will also be disregarded.

   The `force` option will brutally kill an ongoing commit. This could leave the device in a bad state. It is not recommended in any normal use case.

9 **devices commit-queue queue-item [id] prune device [...]** This action prunes the specified devices from the queue item. Devices which are currently being committed to will not be pruned, unless the `force` option is used. Atomic queue items will not be affected, unless all devices in it are pruned.

   The `force` option will brutally kill an ongoing commit. This could leave the device in a bad state. It is not recommended in any normal use case.

10 **devices commit-queue queue-item [id] set-atomic-behaviour atomic [true,false]** This action sets the atomic behaviour of this queue item. If this is set to false, the devices contained in this queue item can start executing if the same devices in other non-atomic queue items ahead of it in the queue are completed. If set to true, the atomic integrity of the queue item is preserved.

11 **devices commit-queue queue-item [id] wait-until-completed** This action waits until the queue item is completed. Default is to wait `infinity`. A `timeout` can be specified to wait for a number of seconds. The result is *completed* if the queue item is completed or *timeout* if the timer expired before the queue item was completed.

12 **devices commit-queue queue-item [id] retry** This action retries devices with transient errors instead of waiting for the automatic retry attempt. The `device` option will let you specify the devices to retry.

A typical use scenario is where one or more devices are not operational. In Example 25, “Viewing queue items”, there are two queue items, waiting for device `ce0` to come alive. `ce0` is listed as a transient error, and this is blocking the entire queue. Actually, whenever a queue item is blocked because another item ahead of it cannot connect to a specific managed device, an alarm is generated:

```
ncs# show alarms alarm-list alarm ce0 commit-through-queue-blocked
alarms alarm-list alarm ce0 commit-through-queue-blocked /devices/device[name='ce0'] 94945301
 is-cleared              false
 last-status-change      2015-02-09T16:48:17.915+00:00
 last-perceived-severity warning
 last-alarm-text         "Commit queue item 9494530158 is blocked because item 9494446997 can
 status-change 2015-02-09T16:48:17.915+00:00
  received-time      2015-02-09T16:48:17.915+00:00
  perceived-severity warning
  alarm-text         "Commit queue item 9494530158 is blocked because item 9494446997 cannot
```

1 Block other affecting device `ce0` from entering the commit-queue:

```
ncs(config)# devices commit-queue add-lock device [ ce0 ] block-others
commit-queue-id 9577950918
ncs# show devices commit-queue | notab
devices commit-queue queue-item 9494446997
 age              1444
 status           executing
 kilo-bytes-size  1
 devices          [ ce0 ce1 ce2 ]
 transient-errors [ ce0 ]
 is-atomic        true
devices commit-queue queue-item 9494530158
 age              1361
 status           blocked
 kilo-bytes-size  1
 devices          [ ce0 ]
 waiting-for      [ ce0 ]
 is-atomic        true
devices commit-queue queue-item 9577950918
 age              55
 status           locked
 kilo-bytes-size  1
 devices          [ ce0 ]
 waiting-for      [ ce0 ]
 is-atomic        true
```

Now queue item `9577950918` is blocking other items using `ce0` from entering the queue.

2 Prune the usage of device `ce0` from all queue items in the commit-queue:

```
ncs(config)# devices commit-queue set-atomic-behaviour atomic false
ncs(config)# devices commit-queue prune device [ ce0 ]
num-affected-queue-items 2
num-deleted-queue-items 1
ncs(config)# show devices commit-queue | notab
devices commit-queue queue-item 9577950918
 age             102
 status          locked
 kilo-bytes-size  1
 devices         [ ce0 ]
 is-atomic       true
```

The lock will be in the queue until it has been deleted or unlocked. Queue items affecting other devices are still allowed entering the queue.

3 Fix the problem with device `ce0`, remove the lock item and sync from the device:

```
ncs(config)# devices commit-queue queue-item 9577950918 delete
ncs(config)# devices device ce0 sync-from
result true
```

Commit Queue in a Cluster Environment

In an LSA cluster each remote NSO has its own commit queue. When committing through the commit queue on the upper node NSO will automatically create queue items on the lower nodes where the devices in the transaction resides. The progress of the lower node queue items are monitored through a queue item on the upper node. The remote NSO is treated itself as a device in the queue item and the remote queue items and devices are opaque to user of the upper node.

Example 26. Commit queue in an LSA cluster

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

ncs(config-if)# commit commit-queue async
commit-queue-id 9494530158

ncs# show devices commit-queue | notab
devices commit-queue queue-item 9494446997
 age             60
 status          executing
 kilo-bytes-size 1
 devices         [ lsa-nso2 lsa-nso3 ]
 is-atomic       true
```

```
ncs# show devices commit-queue | notab
devices commit-queue queue-item 9494446997
 age            66
 status         executing
 kilo-bytes-size 1
 devices        [ lsa-nso2 ]
 completed      [ lsa-nso3 ]
 is-atomic      true

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

Warning Generally it is not recommended to interfere with the queue items of the lower nodes that have been created by an upper NSO. This can cause the upper queue item to not synchronize with the lower ones correctly.

Configuring Commit Queue in a Cluster Environment

To be able to track the commit queue on the lower cluster nodes, NSO uses the built-in stream `ncs-events` that generates northbound notifications for internal events. This stream is required if running the commit queue in a clustered scenario. It is enabled in `ncs.conf`:

Example 27. Enabling the ncs-events stream

```
<stream>
  <name>ncs-events</name>
  <description>NCS event according to tailf-ncs-devices.yang</description>
  <replay-support>true</replay-support>
  <builtin-replay-store>
    <enabled>true</enabled>
    <dir>./state</dir>
    <max-size>S10M</max-size>
    <max-files>50</max-files>
  </builtin-replay-store>
</stream>
```

In addition the commit queue needs to be enabled in the cluster configuration.

```
ncs(config)# cluster commit-queue enabled
ncs(config)# commit
```

For more detailed information on how to set up clustering, see Chapter 1, *LSA Overview* in *Layered Service Architecture*.

Error Recovery with Commit Queue

The goal of the commit queue is to increase the transactional throughput of NSO while keeping an eventual consistency view of the database. This means no matter if changes committed through the commit queue originate as pure device changes or as the effect of service manipulations the effects on the network should eventually be the same as if performed without a commit queue no matter if they succeed or not. This should be applicable to a single NSO node as well as NSO nodes in an LSA cluster.

Depending on the selected `error-option` NSO will store the reverse of the original transaction to be able to undo the transaction changes and get back to the previous state. This data is stored in the `/ncs:devices/commit-queue/completed` tree from where it can be viewed and invoked with the **rollback** action. When invoked the data will be removed.

Example 28. Viewing completed queue items

```
ncs# show devices commit-queue completed | notab
devices commit-queue completed queue-item 9494446997
 when      2015-02-09T16:48:17.915+00:00
 succeeded false
 devices   [ ce0 ce1 ce2 ]
 failed ce0
  reason "Failed to connect to device ce0: closed"
devices commit-queue completed queue-item 9494530158
 when      2015-02-09T16:48:17.915+00:00
 succeeded false
 devices   [ ce0 ]
 failed ce0
  reason "Deleted by user"
```

The error option can be configured under `/ncs:devices/global-settings/commit-queue/error-option`. Possible values are: *continue-on-error*, *rollback-on-error* and *stop-on-error*. The *continue-on-error* value means that the commit queue will continue on errors. No rollback data will be created. The *rollback-on-error* value means that the commit queue item will roll back on errors. The commit queue will place a lock on the failed queue item, thus blocking other queue items with overlapping devices to be executed. The **rollback** action will then automatically be invoked when the queue item has finished its execution. The lock will be removed as part of the rollback. The *stop-on-error* means that the commit queue will place a lock on the failed queue item, thus blocking other queue items with overlapping devices to be executed. The lock must then either manually be released when the error is fixed or the **rollback** action under `/devices/commit-queue/completed` be invoked. The **rollback** action is as:

Example 29. Execute rollback action

```
ncs(config)# devices commit-queue completed queue-item 9494446997 rollback
```

The error option can also be given as a commit parameter.

Note To guarantee service integrity NSO checks for overlapping service or device modifications against the items in the commit queue and returns an error if such exists. If a service instance does a shared set on the same data as a service instance in the queue actually changed, the reference count will be increased but no actual change is pushed to the device(s). This will give a false positive that the change is actually deployed in the network. The *rollback-on-error* and *stop-on-error* error options will automatically create a queue lock on the involved services and devices to prevent such a case.

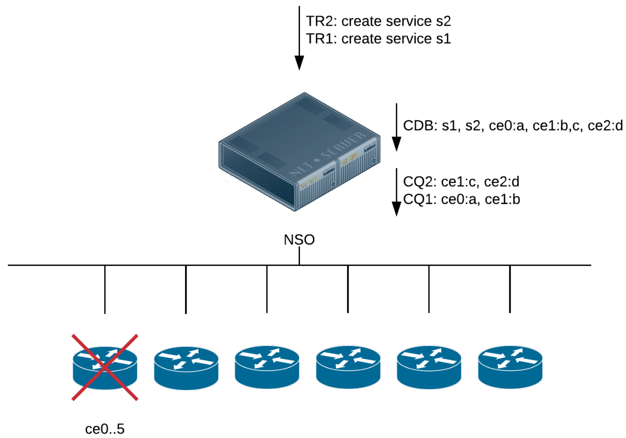
In a clustered environment, different parts of the resulting configuration change set will end up on different lower nodes. This means on some nodes the queue item could succeed and on others it could not.

The error option in a cluster environment will originate on the upper node. The reverse of the original transaction will be committed on this node and propagated through the cluster down to the lower nodes. The net effect of this is the state of the network will be the same as before the original change.

Note As the error option in a cluster environment will originate on the upper node, any configuration on the lower nodes will be meaningless.

When NSO is recovering from a failed commit, the rollback data of the failed queue items in the cluster, is applied and committed through the commit queue. In the rollback the no-networking flag will be set on the commits towards the failed lower nodes or devices to get CDB consistent with the network. Towards the successful nodes or devices the commit is done as before. This is what the `rollback` action in `/ncs:devices/commit-queue/completed/queue-item` does.

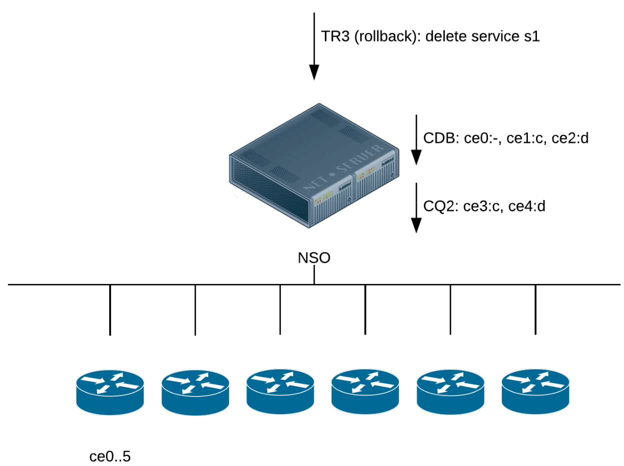
Example 30. Error recovery in a single node deployment



1 TR1; service *s1* creates *ce0:a* and *ce1:b*. The nodes *a* and *b* are created in CDB. In the changes of the queue item, CQ1, *a* and *b* are created.

2 TR2; service *s2* creates *ce1:c* and *ce2:d*. The nodes *c* and *d* are created in CDB. In the changes of the queue item, CQ2, *c* and *d* are created.

3 The queue item from TR1, CQ1, starts to execute. The node *a* cannot be created on the device. The node *b* was created on the device but that change is reverted as *a* failed to be created.

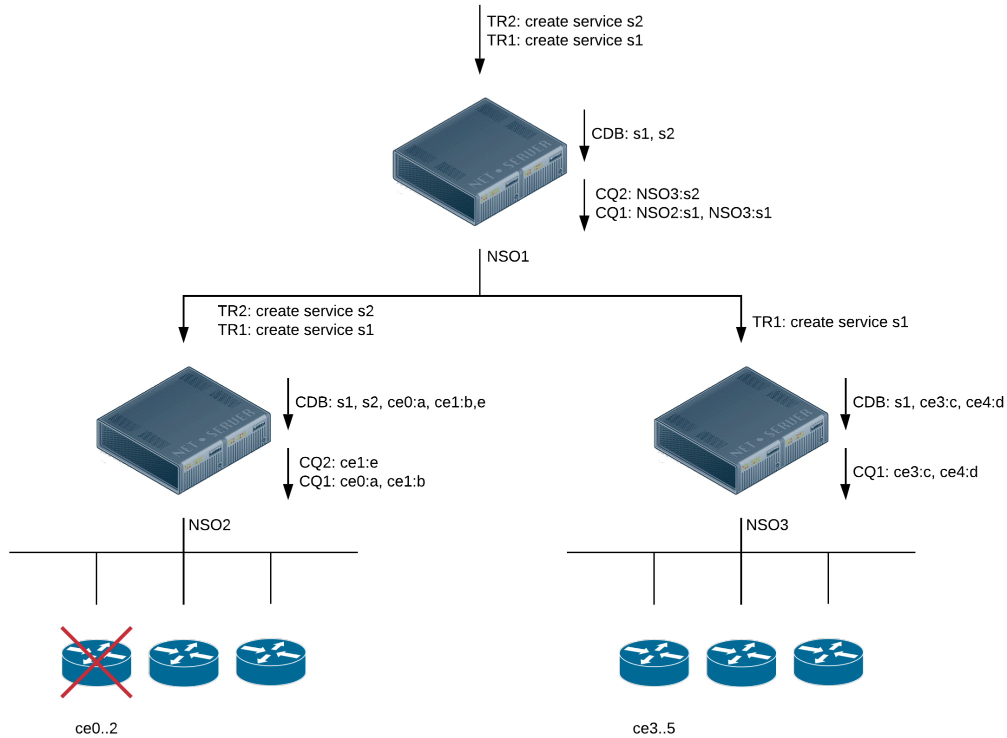


4 The reverse of TR1, rollback of CQ1, TR3, is committed.

5 TR3; service *s1* is applied with the old parameters. Thus the effect of TR1 is reverted. Nothing needs to be pushed towards the network, so no queue item is created.

6 TR2; as the queue item from TR2, CQ2, is not the same service instance and has no overlapping data on the *ce1* device, this queue item executes as normal.

Example 31. Error recovery in an LSA cluster

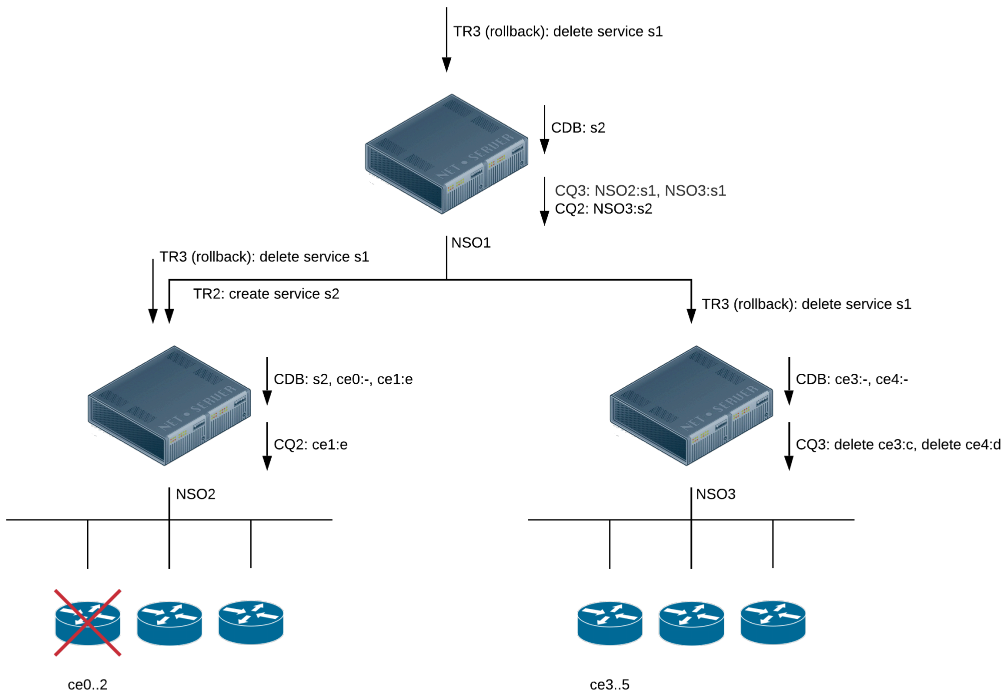


1 NSO1:TR1; service *s1* dispatches the service to NSO2 and NSO3 through the queue item NSO1:CQ1. In the changes of NSO1:CQ1, NSO2:*s1* and NSO3:*s1* are created.

2 NSO1:TR2; service *s2* dispatches the service to NSO2 through the queue item NSO1:CQ2. In the changes of NSO1:CQ2, NSO2:*s2* is created.

3 The queue item from NSO2:TR1, NSO2:CQ1, starts to execute. The node *a* cannot be created on the device. The node *b* was created on the device but that change is reverted as *a* failed to be created.

4 The queue item from NSO3:TR1, NSO3:CQ1, starts to execute. The changes in the queue item is committed successfully to the network.



5 The reverse of TR1, rollback of CQ1, TR3, is committed on all nodes part of TR1 that failed.

6 NSO2:TR3; service *s1* is applied with the old parameters. Thus the effect of NSO2:TR1 is reverted. Nothing needs to be pushed towards the network, so no queue item is created.

7 NSO1:TR3; service *s1* is applied with the old parameters. Thus the effect of NSO1:TR1 is reverted. A queue item is created to push the transaction changes to the lower nodes that didn't fail.

8 NSO3:TR3; service *s1* is applied with the old parameters. Thus the effect of NSO3:TR1 is reverted. Since the changes in the queue item NSO3:CQ1 was successfully committed to the network a new queue item NSO3:CQ3 is created to revert those changes.

If for some reason the rollback transaction would fail there are, depending on the failure, different techniques to reconcile the services involved:

- Make sure the commit queue is blocked to not interfere with the error recovery procedure. Do a sync-from on the non-completed device(s) and then re-deploy the failed service(s) with the `reconcile` option to reconcile original data, i.e., take control of that data. This option acknowledges other services controlling the same data. The reference count will indicate how many services control the data. Release any queue lock that was created.
- Make sure the commit queue is blocked to not interfere with the error recovery procedure. Use un-deploy with the no-networking option on the service and then do sync-from on the non-completed device(s). Make sure the error is fixed and then re-deploy the failed service(s) with the `reconcile` option. Release any queue lock that was created.

Commit Queue Tuning

As the goal of the commit queue is to increase the transactional throughput of NSO it means that we need to calculate the configuration change towards the device(s) outside of the transaction lock. To calculate a configuration change NSO needs a pre-commit running and a running view of the database. The key enabler to support this in the commit queue is to allow different views of the database to live beyond the commit. In NSO this is implemented by keeping a snapshot database of the configuration tree for devices and store configuration changes towards this snapshot database on a per device basis. The snapshot database is updated when a device in the queue has been processed. This snapshot database is stored on disk for persistence (the S.cdb file in the ncs-cdb directory).

The snapshot database could be populated in two ways. This is controlled by the `/ncs-config/cdb/snapshot/pre-populate` setting in the `ncs.conf` file. The parameter controls if the snapshot database should be pre-populated during upgrade or not. Switching this on or off implies different trade-offs.

If set to `false`, NSO is optimized for the default transaction behaviour. The snapshot database is populated in a lazy manner (when a device is committed through the commit queue for the first time after an upgrade). The drawback is that this commit will suffer performance wise, which is especially true for devices with large configurations. Subsequent commits on the same device will not have the same penalty.

If `true`, NSO is optimized for systems using the commit queue extensively. This will lead to better performance when committing using the commit queue with no additional penalty for the first time commits. The drawbacks are that the time to do upgrades will increase and also an almost twofold increase of NSO memory consumption.

NETCONF Call Home

The NSO device manager has built-in support for the NETCONF Call Home client protocol operations over SSH as defined in RFC 8071.

With NETCONF SSH Call Home, the NETCONF client listens for TCP connection requests from NETCONF servers. The SSH client protocol is started when the connection is accepted. The SSH client validates the server's presented host key with credentials stored in NSO. If no matching host key is found the TCP connection is closed immediately. Otherwise the SSH connection is established, and NSO is enabled to communicate with the device. The SSH connection is kept open until the devices itself terminates the connection, a NSO user disconnects the device, or the idle connection timeout is triggered (configurable in the `ncs.conf` file).

NSO will generate an asynchronous notification event whenever there is a connection request. An application can subscribe to these events and, for example, add an unknown device to the device tree with the information provided, or invoke actions on the device if it is known.

If an SSH connection is established, any outstanding configuration in the commit queue for the device will be pushed. Any notification stream for the device will also be reconnected.

NETCONF Call Home is enabled and configured under `/ncs-config/netconf-call-home` in the `ncs.conf` file. By default NETCONF Call Home is disabled.

A device can be connected through the NETCONF Call Home client only if `/devices/device/state/admin-state` is set to `call-home`. This state prevents any southbound communication to the device unless the connection has already been established through the NETCONF Call Home client protocol.

Notifications

The NSO device manager has built-in support for device notifications. Notifications are a means for the managed devices to send structured data asynchronously to the manager. NSO has native support for NETCONF event notifications (see RFC 5277) but could also receive notifications from other protocols implemented by the Network Element Drivers.

Notifications can be utilized in various different use case scenarios - It can be used to populate alarms in the Alarm manager, collect certain types of errors over time, build a network wide audit log, react on configuration changes etc.

The basic mode of operation is the manager subscribes to one or more *named* notification channels which are announced by the managed device. The manager keeps an open SSH channel towards the managed device, and then, the managed device may asynchronously send structured XML data on the ssh channel.

The notification support in NSO is usable as is without any further programming. However, NSO cannot understand any semantics contained inside the received XML messages, thus for example a notification with a content of "Clear Alarm 456" cannot be processed by NSO without any additional programming.

When you add programs to interpret and act upon notifications, make sure that resulting operations are idempotent. This means that they should be able to be called any number of times while guaranteeing that side effects only occur once. The reason for this is that, for example, replaying notifications can sometimes mean that your program will handle the same notifications multiple times.

In the `tailf-ncs.yang` data model you find a YANG data model which can be used to:

- Setup subscriptions. A subscription is configuration data from the point of view of NSO, thus if NSO is restarted, all configured subscriptions are automatically resumed.
- Inspect which named streams a managed device publishes.
- View all received notifications.

Note Notifications must be defined at the top level of a YANG module. NSO does currently not support defining notifications inside lists or containers as specified in section 7.16 in RFC 7950.

An Example Session

In this section we will use the `examples.ncs/web-server-farm/basic` example.

Let's dive into an example session with the NSO CLI. In the NSO example collection, the webserver publish two NETCONF notification structures, indicating what they intend to send to any interested listeners. They all have the YANG module:

Example 32. notif.yang

```
module notif {
  namespace "http://router.com/notif";
  prefix notif;

  import ietf-inet-types {
    prefix inet;
  }


  notification startUp {
    leaf node-id {
      type string;
    }
  }

  notification linkUp {
    leaf ifName {
      type string;
```

```
      mandatory true;
    }
    leaf extraId {
      type string;
    }
    list linkProperty {
      max-elements 64;
      leaf newlyAdded {
        type empty;
      }
      leaf flags {
        type uint32;
        default 0;
      }
      list extensions {
        max-elements 64;
        leaf name {
          type uint32;
          mandatory true;
        }
        leaf value {
          type uint32;
          mandatory true;
        }
      }
    }

    list address {
      key ip;
      leaf ip {
        type inet:ipv4-address;
      }
      leaf mask {
        type inet:ipv4-address;
      }
    }

    leaf-list iface-flags {
      type enumeration {
        enum UP;
        enum DOWN;
        enum BROADCAST;
        enum RUNNING;
        enum MULTICAST;
        enum LOOPBACK;
      }
    }
  }


  notification linkDown {
    leaf ifName {
      type string;
      mandatory true;
    }
  }
}
```

Follow the instructions in the README file if you want to run the example: build the example, start netsim, start ncs.

```
admin@ncs# show devices device pe2 notifications stream | notab
```

```
notifications stream NETCONF
 description    "default NETCONF event stream"
 replay-support false
notifications stream tailf-audit
 description    "Tailf Commit Audit events"
 replay-support true
notifications stream interface
 description              "Example notifications"
 replay-support           true
 replay-log-creation-time 2014-10-14T11:21:12+00:00
 replay-log-aged-time     2014-10-14T11:53:19.649207+00:00
```

The above shows how we can inspect - as status data - which named streams the managed device publishes. Each stream also has some associated data. The data model for that looks like:

Example 33. tailf-ncs.yang notification streams

```
module tailf-ncs {
  namespace "http://tail-f.com/ns/ncs";
  ...
  container devices {
     list device {
       ....
       container notifications {
          ....

          list stream {
             description "A list of the notification streams
                          provided by the device. NCS reads this list in
                          real time";

             config false;
             key name;
             leaf name {
               description "The name of the the stream";
               type string;
             }
             leaf description {
               description "A textual description of the stream";
               type string;
             }
             leaf replay-support {
               description "An indication of whether or not event replay
                             is available on this stream.";
               type boolean;
             }
             leaf replay-log-creation-time {
               description "The timestamp of the creation of the log
                            used to support the replay function on
                            this stream.
                            Note that this might be earlier then
                            the earliest available
                            notification in the log.  This object
                            is updated if the log resets
                            for some reason.";

               type yang:date-and-time;
             }
             leaf replay-log-aged-time {
               description "The timestamp of the last notification
                             aged out of the log";
               type yang:date-and-time;
```

```
        }
      }
```

Let's setup a subscription for the stream called *interface*. The subscriptions are NSO configuration data, thus to create a subscription we need to enter configuration mode:

Example 34. Configuring a Subscription

```
admin@ncs(config)# devices device www0..2 notifications \
      subscription mysub stream interface
admin@ncs(config-subscription-mysub)# commit
```

The above, created subscriptions for the *interface* stream on all web servers, i.e managed devices, www0, www1 and www2. Each subscription must have an associated stream to it, this is however not the key for an NSO notification, the key is a free form text string. This is since we can have multiple subscriptions to the same stream. More on this later when we describe the filter that can be associated to a subscription. Once the notifications start to arrive, they are read by NSO and stored in stable storage as CDB operational data. they are stored under each managed device - and we can view them as:

Example 35. Viewing the Received Notifications

```
admin@ncs# show devices device notifications | notab
devices device www0
 notifications subscription mysub
  local-user admin
  status     running
 notifications stream NETCONF
  description    "default NETCONF event stream"
  replay-support false
 notifications stream tailf-audit
  description    "Tailf Commit Audit events"
  replay-support true
 notifications stream interface
  description              "Example notifications"
  replay-support           true
  replay-log-creation-time 2014-10-14T11:21:12+00:00
  replay-log-aged-time     2014-10-14T11:56:45.755964+00:00
 notifications notification-name startUp
  uri http://router.com/notif
 notifications notification-name linkUp
  uri http://router.com/notif
 notifications notification-name linkDown
  uri http://router.com/notif
 notifications received-notifications notification 2014-10-14T11:54:43.692371+00:00 0
  user         admin
  subscription  mysub
  stream        interface
  received-time 2014-10-14T11:54:43.695191+00:00
  data linkUp ifName eth2
  data linkUp linkProperty
   newlyAdded
   flags      42
   extensions
    name  1
    value 3
   extensions
    name  2
    value 4668
  data linkUp address 192.168.128.55
   mask 255.255.255.0
```

Each received notification has some associated meta data, such as the time the event was received by NSO, which subscription and which stream is associated to the the notification and also which user created the subscription.

It is fairly instructive to inspect the XML that goes on the wire when we create a subscription and then also receive the first notification. We can do:

```
ncs(config)# devices global-settings trace pretty trace-dir ./logs
ncs(config)# commit

ncs(config)# devices disconnect

ncs(config)# devices device pe2 notifications \
     subscription foo stream interface
ncs(config-subscription-foo)# top
ncs(config)# exit

ncs# file show ./logs/netconf-pe2.trace
<<<<in 14-Oct-2014::13:59:52.295 device=pe2 session-id=14
<notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
  <eventTime>2014-10-14T11:58:51.816077+00:00</eventTime>
  <linkUp xmlns="http://router.com/notif">
    <ifName>eth2</ifName>
    <linkProperty>
      <newlyAdded/>
      <flags>42</flags>
      <extensions>
        <name>1</name>
        <value>3</value>
      </extensions>
      <extensions>
        <name>2</name>
        <value>4668</value>
      </extensions>
    </linkProperty>
    <address>
      <ip>192.168.128.55</ip>
      <mask>255.255.255.0</mask>
    </address>
  </linkUp>
</notification>
 .........
```

Thus, once the subscription has been configured, NSO continuously receives, and stored in CDB oper persistent storage, the notifications sent from the managed device. The notifications are stored in a circular buffer, to set the size of the buffer, we can do:

```
ncs(config)# devices device www0 notifications \
   received-notifications max-size 100
admin@ncs(config-device-www0)# commit
```

The default value is 200. Once the size of the circular buffer is exceeded, the oldes notification is removed.

Subscription Status

A running subscription can be in either of three states. The YANG model has:

```
module tailf-ncs {
  namespace "http://tail-f.com/ns/ncs";
  ...
  container devices {
```

```
list device {
  ....
  container notifications {
     ....
     list subscription {
        .....
       leaf status {
       description "Is this subscription currently running";
       config false;
       type enumeration {
         enum running {
           description "The subscription is established and we should
                        be receiving notifications";
         }
         enum connecting {
           description "Attempting to establish the subscription";
         }
         enum failed {
           description
           "The subscription has failed, unless the failure is
            in the connection establishing, i.e connect() failed
            there will be no automatic re-connect";
         }
       }
     }
```

If a subscription is in the *failed* state, an optional *failure-reason* field indicates the reason for the failure. If a subscription fails due to, not being able to connect to the managed device or if the managed device closed its end of the SSH socket, NSO will attempt to automatically reconnect. The re-connect attempt interval is configurable.

```
ncs# show devices device notifications subscription
             LOCAL            FAILURE  ERROR
NAME  NAME   USER   STATUS    REASON   INFO
---------------------------------------------
www0  foo    admin  running  -        -
      mysub  admin  running  -        -
www1  mysub  admin  running  -        -
www2  mysub  admin  running  -        -
```

SNMP Notifications

SNMP Notifications (v1, v2c, v3) can be received by NSO and acted upon. The SNMP receiver is a stand-alone process and by default all notifications are ignored. IP addresses must be opted in and a handler must be defined to take actions on certain notifications. This can be used to for example listen to configuration change notifications and trigger a log action or a resync for example

This actions are programmed in Java, see the Chapter 26, *SNMP Notification Receiver* in *Development Guide* in Development Guide how to do this.

Inactive configuration

NSO is able to configure inactive parameters on the devices that support inactive configuration. Currently these devices include Juniper devices and devices that announce 'http://tail-f.com/ns/netconf/inactive/1.0' capability. NSO itself implements 'http://tail-f.com/ns/netconf/inactive/1.0' capability which is formally defined in tailf-netconf-inactive YANG module.

To recap, a node that is marked as inactive exists in the datastore, but is not used by the server. The nodes announced as inactive by the device will also be inactive in the device's configuration in NSO, and activating/deactivating a node in NSO will push the corresponding change to the device. This also means that in order for NSO to be able to manage inactive configuration both /ncs-config/enable-inactive and /ncs-config/netconf-north-bound/capabilities/inactive need to be enabled in ncs.conf.

If the inactive feature is disabled in ncs.conf, NSO will still be able to manage devices that have inactive configuration in their datastore, but the inactive attribute will be ignored, so the data will appear as active in NSO and it would not be possible for NSO to activate/deactivate such nodes in the device.

CHAPTER 7

SSH Key Management

General

The SSH protocol uses public key technology for two distinct purposes:

Server authentication	This use is a mandatory part of the protocol. It allows an SSH client to authenticate the server, i.e. verify that it is really talking to the intended server and not some man-in-the-middle intruder. This requires that the client has a priori knowledge of the server's public keys, and the server proves its possession of one of the corresponding private keys by using it to sign some data. These keys are normally called "host keys", and the authentication procedure is typically referred to as "host key verification" or "host key checking".
Client authentication	This use is one of several possible client authentication methods, i.e. it is an alternative to the commonly used password authentication. The server is configured with one or more public keys which are authorized for authentication of a user. The client proves possession of one of the corresponding private keys by using it to sign some data - i.e. the exact reverse of the server authentication provided by host keys. The method is called "publickey" authentication in SSH terminology.

These two usages are fundamentally independent, i.e. host key verification is done regardless of whether the client authentication is via publickey, password, or some other method. However host key verification is of particular importance when client authentication is done via password, since failure to detect a man-in-the-middle attack in this case will result in the cleartext password being divulged to the attacker.

NSO as SSH Server

NSO can act as SSH server for northbound connections to the CLI or the NETCONF agent, and for connections from other nodes in an NSO cluster - cluster connections use NETCONF, and the server side setup used is the same as for northbound connections to the NETCONF agent. It is possible to use either the NSO built-in SSH server, or an external server such as OpenSSH, for all of these cases. When using an external SSH server, host keys for server authentication and authorized keys for client/user authentication need to be set up per the documentation for that server, and there is no NSO-specific key management in this case.

When the NSO built-in SSH server is used, the setup is very similar to the one OpenSSH uses:

Host Keys

The private host key(s) must be placed in the directory specified by `/ncs-config/aaa/ssh-server-key-dir` in `ncs.conf`, and named either `ssh_host_dsa_key` (for a DSA key) or `ssh_host_rsa_key` (for a RSA key). The key(s) must be in PEM format (e.g. as generated by the OpenSSH **ssh-keygen** command), and must not be encrypted - protection can be achieved by file system permissions (not enforced by NSO). The corresponding public key(s) is/are typically stored in the same directory with a `.pub` extension to the file name, but they are not used by NSO. The NSO installation creates a DSA private/public key pair in the directory specified by the default `ncs.conf`.

Publickey Authentication

The public keys that are authorized for authentication of a given user must be placed in the user's SSH directory. Please refer to the section called “Public Key Login” in *Administration Guide* for the details of how NSO searches for the keys to use.

NSO as SSH Client

NSO can act as SSH client for connections to managed devices that use SSH (this is always the case for devices accessed via NETCONF, typically also for devices accessed via CLI), and for connections to other nodes in an NSO cluster. In all cases a built-in SSH client is used. The `$NCS_DIR/examples.ncs/getting-started/using-ncs/8-ssh-keys` example in the NSO example collection has a detailed walk-through of the NSO functionality that is described in this section.

Host Key Verification

Verification level

The level of host key verification can be set globally via `/ssh/host-key-verification`. The possible values are:

`reject-unknown`
: The host key provided by the device or cluster node must be known by NSO for the connection to succeed.

`reject-mismatch`
: The host key provided by the device or cluster node may be unknown, but it must not be different from the "known" key for the same key algorithm, for the connection to succeed.

`none`
: No host key verification is done - the connection will never fail due to the host key provided by the device or cluster node.

The default is `reject-unknown`, and it is not recommended to use a different value, although it can be useful or needed in certain circumstances. E.g. `none` may be useful in a development scenario, and temporary use of `reject-mismatch` may be motivated until host keys have been configured for a set of existing managed devices.

Example 36. Allowing SSH Connections With Unknown Host Keys

```
admin@ncs(config)# ssh host-key-verification reject-mismatch
admin@ncs(config)# commit
Commit complete.
```

Connection to a Managed Device

The public host keys for a device that is accessed via SSH are stored in the `/devices/device/ssh/host-key` list. There can be several keys in this list, one each for the `ssh-ed25519` (ED25519 key), `ssh-dss` (DSA key) and `ssh-rsa` (RSA key) key algorithms. In case a device has entries in its `live-status-protocol` list that use SSH, the host keys for those can be stored in the `/devices/device/live-status-protocol/ssh/host-key` list, in the same way as the device keys - however if `/devices/device/live-status-protocol/ssh` does not exist, the keys from `/devices/device/ssh/host-key` are used for that protocol. The keys can be configured e.g. via input directly in the CLI, but in most cases it will be preferable to use the actions described below to retrieve keys from the devices. These actions will also retrieve any `live-status-protocol` keys for a device.

The level of host key verification can also be set per device, via `/devices/device/ssh/host-key-verification`. The default is to use the global value (or default) for `/ssh/host-key-verification`, but any explicitly set value will override the global value. The possible values are the same as for `/ssh/host-key-verification`.

There are several actions that can be used to retrieve the host keys from a device and store them in the NSO configuration:

`/devices/fetch-ssh-host-keys`	Retrieve the host keys for all devices. Successfully retrieved keys are committed to the configuration.
`/devices/device-group/fetch-ssh-host-keys`	Retrieve the host keys for all devices in a device group. Successfully retrieved keys are committed to the configuration.
`/devices/device/ssh/fetch-host-keys`	Retrieve the host keys for one or more devices. In the CLI, range expressions can be used for the device name, e.g. using '*' will retrieve keys for all devices etc. The action will commit the retrieved keys if possible, i.e. if the device entry is already committed, otherwise (i.e. if the action is invoked from "configure mode" when the device entry has been created but not committed), the keys will be written to the current transaction, but not committed.

The fingerprints of the retrieved keys will be reported as part of the result from these actions, but it is also possible to ask for the fingerprints of already retrieved keys by invoking the `/devices/device/ssh/host-key/show-fingerprint` action (`/devices/device/live-status-protocol/ssh/host-key/show-fingerprint` for live-status-protocols that use SSH).

Example 37. Retrieving SSH Host Keys for All Configured Devices

```
admin@ncs# devices fetch-ssh-host-keys
fetch-result {
    device c0
    result unchanged
    fingerprint {
        algorithm ssh-dss
        value 03:64:fc:b7:87:bd:34:5e:3b:6e:d8:71:4d:3f:46:76
    }
}
fetch-result {
    device h0
    result unchanged
    fingerprint {
        algorithm ssh-dss
        value 03:64:fc:b7:87:bd:34:5e:3b:6e:d8:71:4d:3f:46:76
```

```
    }
}
```

Connection to an NSO Cluster Node

This is very similar to the case of a connection to a managed device, it differs mainly in locations - and in the fact that SSH is always used for connection to a cluster node. The public host keys for a cluster node are stored in the `/cluster/remote-node/ssh/host-key` list, in the same way as the host keys for a device. The keys can be configured e.g. via input directly in the CLI, but in most cases it will be preferable to use the action described below to retrieve keys from the cluster node.

The level of host key verification can also be set per cluster node, via `/cluster/remote-node/ssh/host-key-verification`. The default is to use the global value (or default) for `/ssh/host-key-verification`, but any explicitly set value will override the global value. The possible values are the same as for `/ssh/host-key-verification`.

The `/cluster/remote-node/ssh/fetch-host-keys` action can be used to retrieve the host keys for one or more cluster nodes. In the CLI, range expressions can be used for the node name, e.g. using '*' will retrieve keys for all nodes etc. The action will commit the retrieved keys if possible, but if it is invoked from "configure mode" when the node entry has been created but not committed, the keys will be written to the current transaction, but not committed.

The fingerprints of the retrieved keys will be reported as part of the result from this action, but it is also possible to ask for the fingerprints of already retrieved keys by invoking the `/cluster/remote-node/ssh/host-key/show-fingerprint` action.

Example 38. Retrieving SSH Host Keys for All Cluster Nodes

```
admin@ncs# cluster remote-node * ssh fetch-host-keys
cluster remote-node ncs1 ssh fetch-host-keys
    result updated
    fingerprint {
        algorithm ssh-dss
        value 03:64:fc:b7:87:bd:34:5e:3b:6e:d8:71:4d:3f:46:76
    }
cluster remote-node ncs2 ssh fetch-host-keys
    result updated
    fingerprint {
        algorithm ssh-dss
        value 03:64:fc:b7:87:bd:34:5e:3b:6e:d8:71:4d:3f:46:76
    }
cluster remote-node ncs3 ssh fetch-host-keys
    result updated
    fingerprint {
        algorithm ssh-dss
        value 03:64:fc:b7:87:bd:34:5e:3b:6e:d8:71:4d:3f:46:76
    }
```

Publickey Authentication

Private Key Selection

The private key used for publickey authentication can be taken either from the SSH directory for the local user, or from a list of private keys in the NSO configuration. The user's SSH directory is determined according to the same logic as for the server-side public keys that are authorized for authentication of a given user, see the section called “Public Key Login” in *Administration Guide,* but of course different files in this directory are used, see below. Alternatively the key can be configured in the `/ssh/private-key` list, using an arbitrary name for the list key. In both cases the key must be in PEM format (e.g.

as generated by the OpenSSH **ssh-keygen** command), and it may be encrypted or not. Encrypted keys configured in `/ssh/private-key` must have the passphrase for the key configured via `/ssh/private-key/passphrase`.

Connection to a Managed Device

The specific private key to use is configured via the `authgroup` indirection and the `umap` selection mechanisms as for password authentication, just a different alternative. Setting `/devices/authgroups/group/umap/public-key` (or `default-map` instead of `umap` for users that are not in `umap`) without any additional parameters will select the default of using a file called `id_dsa` in the local user's SSH directory, which must have an unencrypted key. A different file name can be set via `/devices/authgroups/group/umap/public-key/private-key/file/name`. For an encrypted key, the passphrase can be set via `/devices/authgroups/group/umap/public-key/private-key/file/passphrase`, or `/devices/authgroups/group/umap/public-key/private-key/file/use-password` can be set to indicate that the password used (if any) by the local user when authenticating to NSO should also be used as passphrase for the key. To instead select a private key from the `/ssh/private-key` list, the name of the key is set via `/devices/authgroups/group/umap/public-key/private-key/name`.

Example 39. Configuring a Private Key File for Publickey Authentication to Devices

```
admin@ncs(config)# devices authgroups group default umap admin
admin@ncs(config-umap-admin)# public-key private-key file name /home/admin/.ssh/id-dsa
admin@ncs(config-umap-admin)# public-key private-key file passphrase
(<AES encrypted string>): *********
admin@ncs(config-umap-admin)# commit
Commit complete.
```

Connection to an NSO Cluster Node

This is again very similar to the case of a connection to a managed device, since the same `authgroup`/`umap` scheme is used. Setting `/cluster/authgroup/umap/public-key` (or `default-map` instead of `umap` for users that are not in `umap`) without any additional parameters will select the default of using a file called `id_dsa` in the local user's SSH directory, which must have an unencrypted key. A different file name can be set via `/cluster/authgroup/umap/public-key/private-key/file/name`. For an encrypted key, the passphrase can be set via `/cluster/authgroup/umap/public-key/private-key/file/passphrase`, or `/cluster/authgroup/umap/public-key/private-key/file/use-password` can be set to indicate that the password used (if any) by the local user when authenticating to NSO should also be used as passphrase for the key. To instead select a private key from the `/ssh/private-key` list, the name of the key is set via `/cluster/authgroup/umap/public-key/private-key/name`.

Example 40. Configuring a Private Key File for Publickey Authentication in Cluster

```
admin@ncs(config)# cluster authgroup default umap admin
admin@ncs(config-umap-admin)# public-key private-key file name /home/admin/.ssh/id-dsa
admin@ncs(config-umap-admin)# public-key private-key file passphrase
(<AES encrypted string>): *********
admin@ncs(config-umap-admin)# commit
Commit complete.
```

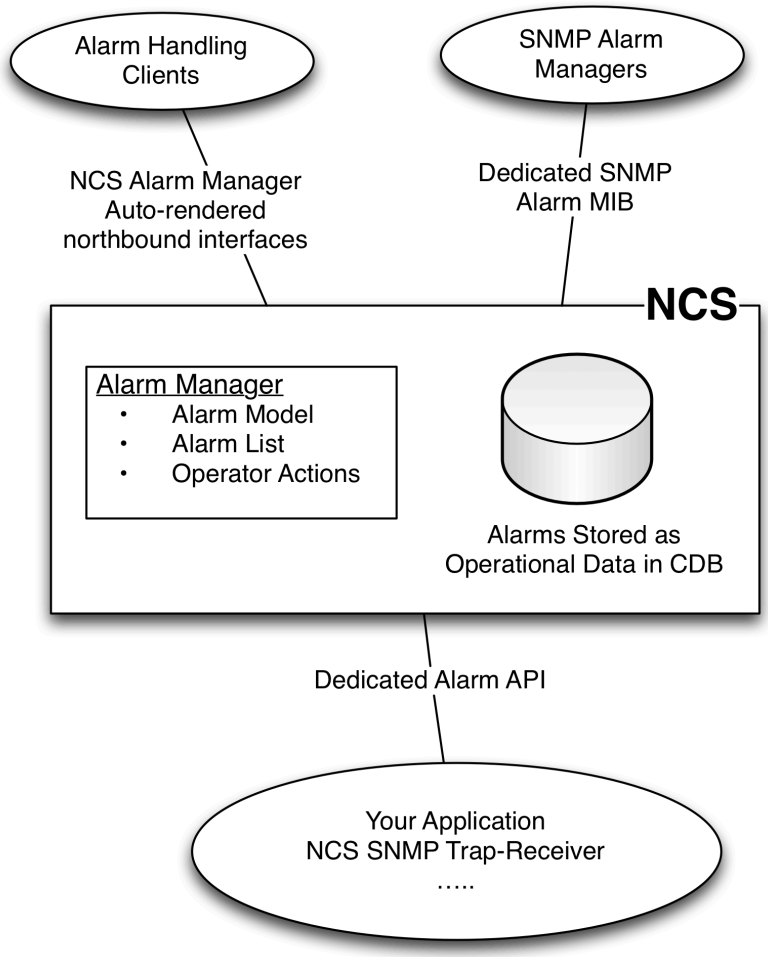
CHAPTER 8

Alarm Manager

Alarm Manager Introduction

NSO embeds a generic alarm manager. It is used for managing NSO native alarms and can easily be extended with application specific alarms. Alarm sources can be notifications from devices, undesired states on services detected or anything provided via the Java API.

Figure 41. The Alarm Manager



The Alarm Manager has three main components:

Alarm List	a list of alarms in NSO. Each list entry represents an alarm state for a specific device, object within the device and an alarm type
Alarm Model	for each alarm type, you can configure the mapping to for example X.733 alarm standard parameters that are sent as notifications northbound
Operator Actions	actions to set operator states on alarms such as acknowledgement, and also actions to administratively manage the alarm list such as deleting alarms

The alarm manager is accessible over all northbound interfaces. A read-only view including an SNMP alarm table and alarm notifications is available in an SNMP Alarm MIB. This MIB is suitable for integration to SNMP based alarm systems.

In order to populate the alarm list there is a dedicated Java API. This API lets a developer add alarms, change states on alarms etc. A common usage pattern is to use the SNMP notification receiver to map a subset of the device traps into alarms.

Overview of the alarm concepts

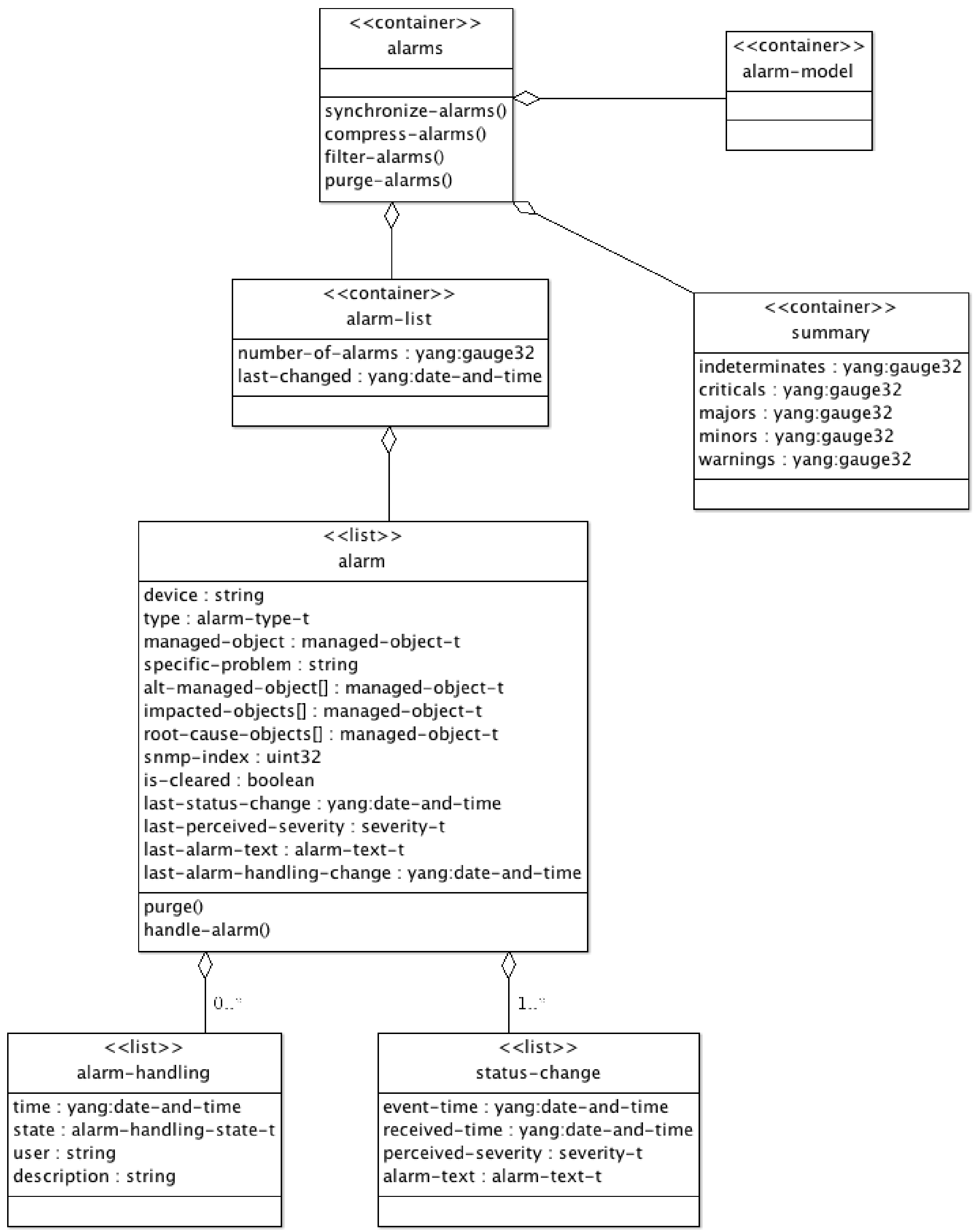
First of all it is important to clearly define what an alarm means. “An alarm denotes an undesirable state in a resource for which an operator action is required. ” Alarms are often confused with general logging and event mechanisms, thereby overflooding the operator with alarms. In NSO, the alarm manager shows undesired resource states that an operator should investigate. NSO contains other mechanisms for logging in general. Therefore, NSO does not naively populate the alarm list with traps received in the SNMP notification receiver.

Before looking into how NSO handles alarms it is important to define the fundamental concepts. We make a clear distinction between alarms and events in general. Alarms should be taken seriously and be investigated. Alarms have states, they go active with a specific severity, they change severity and they are cleared by the resource. The same alarm may become active again. A common mistake is to confuse the operator view with the resource view. The model described so far is the resource view. The resource itself may consider the alarm cleared. The alarm manager does not automatically delete cleared alarms. An alarm that has existed in the network may still need investigation. There are dedicated actions an operator can use to manage the alarm list, for example delete alarms based on criterias such as cleared and date. These actions can be performed over all north-bound interfaces.

Rather than viewing alarms as a list of alarm notifications NSO defines alarms as states on objects. The NSO alarm list uses four keys for alarms: the alarming *object* within a *device* and the *alarm type* and an optional *specific problem*. *Alarm types* are normally unique identifiers for a specific alarm state and are defined statically. An alarm type corresponds to the well-known X.733 alarm standard tuple event type, and probable cause. Specific problem is an optional key that is string based and can further redefine an alarm type at run-time. This is needed for alarms that are not known before a system is deployed. Imagine a system with general digital inputs. A MIB might specify traps called input-high, input-low. When defining the SNMP notification reception, an integrator might define an alarm type called "External-Alarm". `Input-high` might imply a major alarm and `input-low` might imply clear. At installation some detectors report "fire-alarm" and some "door-open" alarms. This is configured at the device and sent as free text in the SNMP var-binds. This is then managed by using the specific problem field of the NSO alarm manager to separate this different alarm types.

The data model for the alarm-manager is outlined in Figure 42, “Alarm Model”

Figure 42. Alarm Model



This means that we have a list with key: (managed device, managed object, alarm type, specific problem). In the example above we might have the following different alarms:

- Device : House1; Managed Object : Detector1; Alarm-Type : External Alarm; Specific Problem = Smoke;

- Device : House1; Managed Object : Detector2; Alarm-Type : External Alarm; Specific Problem = Door Open;

Each alarm entry shows the last status change for the alarm and also a child list with all status changes sorted in chronological order.

is-cleared	was the last state change *clear*?
last-status-change	time stamp for last status change
last-perceived-severity	last severity (not equal to clear)
last-alarm-text	the last alarm text (not equal to clear)
status-change, event-time	the time reported by the device
status-change, received-time	the time the state change was received by NSO
status-change, perceived-severity	the new perceived severity
status-change, alarm-text	descriptive text associated with the new alarm status

It is fundamental to define alarm types (specific problem) and managed objects with a fine-grained mechanism that still is extensible. For objects we allow YANG instance-identifiers to refer to a YANG instance identifier, an SNMP OID, or a string. Strings can be used when the underlying object is not modelled. We use YANG identities to define alarm types. This has the benefit that alarm types can be defined in a named hierarchy and thereby provide an extensible mechanism. In order to support "dynamic alarm types" so that alarms can be separated by information only available at run-time the string based field specific problem can also be used.

So far we have described the model based on the resource view. It is common practice to let operators manipulate the alarms corresponding to the operators investigation. We clearly separate the resource and the operator view, for example, there is no such thing as an operator "clearing an alarm". Rather the alarm entries can have a corresponding alarm handling state. Operators may want to acknowledge an alarm, set the alarm state to closed or similar.

We also support some alarm list administrative actions:

Synchronize alarms	try to read the alarm states in the underlying resources and update the alarm list accordingly (this action needs to be implemented by user code for specific applications)
Purge alarms	delete entries in the alarm list based on several different filter criteria
Filter alarms	with a XPATH as filter input, this action returns all alarms fulfilling the filter
Compress alarms	since every entry may contain a large amount of state change entries this action compresses the history to the latest state change

Alarms can be forwarded over NSO northbound interfaces. In many telecom environments alarms need to be mapped to X.733 parameters. We provide an *alarm model* where every alarm type is mapped to the corresponding X.733 parameters such as event type and probable cause. In this way, it is easy to integrate the NSO alarms into whatever X.733 enumerated values the upper fault management system is requiring.

The Alarm Model

The central part of the YANG Alarm model, `tailf-ncs-alarms.yang` has the following structure.

Example 43. tailf-ncs-alarms.yang

```
module tailf-ncs-alarms {

  namespace "http://tail-f.com/ns/ncs-alarms";
  prefix "al";
```

```
 ...
typedef managed-object-t {
   type union {
     type instance-identifier {
       require-instance false;
       }
     type yang:object-identifier;
     type string;
   }


 ...
 typedef event-type  {
   type enumeration {
     enum other {value 1;}
     enum communicationsAlarm {value 2;}
     enum qualityOfServiceAlarm {value 3;}
     enum processingErrorAlarm {value 4;}
     enum equipmentAlarm {value 5;}
     ...
   }
   description
   "...";
   reference
   "ITU Recommendation X.736, 'Information Technology - Open
    Systems Interconnection - System Management: Security
    Alarm Reporting Function', 1992";
 }

 typedef severity-t  {
   type enumeration {
     enum cleared {value 1;}
     enum indeterminate {value 2;}
     enum critical {value 3;}
     enum major {value 4;}
     enum minor {value 5;}
     enum warning {value 6;}
   }
   description
     "...";
 }
 ...
 identity alarm-type {
   description
   "Base identity for alarm types."
   ...
 }

 identity ncs-dev-manager-alarm {
   base alarm-type;
 }

 identity ncs-service-manager-alarm {
   base alarm-type;
 }

 identity connection-failure {
   base ncs-dev-manager-alarm;
   description
     "NCS failed to connect to a device";
 }
 ....
```

```
container alarm-model {
  list alarm-type {
    key "type";
      leaf type {
        type alarm-type-t;
      }

      uses alarm-model-parameters;
   }
}

    ...


  container alarm-list {
    config false;
    leaf number-of-alarms {
      type yang:gauge32;
    }

    leaf last-changed {
      type yang:date-and-time;
    }

    list alarm {
      key "device type managed-object specific-problem";
      uses common-alarm-parameters;
      leaf is-cleared {
        type boolean;
        mandatory true;
      }

      leaf last-status-change {
        type yang:date-and-time;
        mandatory true;
      }

      leaf last-perceived-severity {
        type severity-t;
      }

      leaf last-alarm-text {
        type alarm-text-t;
      }

      list status-change {
        key event-time;
        min-elements 1;
        uses alarm-state-change-parameters;
      }

      leaf last-alarm-handling-change {
        type yang:date-and-time;
      }

      list alarm-handling {
        key time;
        leaf time {
          tailf:info "Time stamp for operator action";
          type yang:date-and-time;
        }
        leaf state {
```

```
            tailf:info "The operators view of the alarm state";
            type alarm-handling-state-t;
            mandatory true;
            description
              "The operators view of the alarm state.";
          }
          ...
        }
        ...
        notification alarm-notification {
        ...
        rpc synchronize-alarms {
        ...
        rpc compress-alarms {
        ...
        rpc purge-alarms {
```

The first part of the YANG listing above shows the definition for `managed-object` type in order for alarms to refer to YANG, SNMP and other resources. We also see basic definitions from the X.733 standard for severity levels.

Note well the definition of alarm-type using YANG identities. In this way we can create a structured alarm type hierarchy all rooted at `alarm-type`. In order for you to add your specific alarm types, define your own alarm types YANG file and add identities using `alarm-type` as base.

The `alarm-model` container contains the mapping from alarm types to X.733 parameters used for north-bound interfaces.

The `alarm-list` container is the actual alarm list where we maintain a list mapping (device, managed-object, alarm-type, specific-problem) to the corresponding alarm state changes [(time, severity, text)].

Finally, we see the northbound alarm notification and alarm administrative actions.

Alarm handling

The NSO alarm manager has support for the operator to acknowledge alarms. We call this alarm handling. Each alarm has an associated list of alarm handling entries as:

```
container alarms {
  ....
  container alarm-list {
    config false;
    ....
    list alarm {
      key "device type managed-object specific-problem";

      .....

      list alarm-handling {
        key time;
        leaf time {
          type yang:date-and-time;
          description
            "Time-stamp for operator action on alarm.";
        }
        leaf state {
          mandatory true;
          type alarm-handling-state-t;
          description
            "The operators view of the alarm state";
```

```
        }
        leaf user {
          description "Which user has acknowledged this alarm";
          mandatory true;
          type string;
        }
        leaf description {
          description "Additional optional textual information regarding
            this new alarm-handling entry";
          type string;
        }
      }

        tailf:action handle-alarm {
          tailf:info "Set the operator state of this alarm";
          description
            "An action to allow the operator to add an entry to the
             alarm-handling list. This is a means for the operator to indicate
             the level of human intervention on an alarm.";
          input {
            leaf state {
              type alarm-handling-state-t;
              mandatory true;
            }
          }
        }
      }
```

The following typedef defines the different states an alarm can be set into.

Example 44. Alarm state

```
  typedef alarm-handling-state-t  {
    type enumeration {
      enum none {
        value 1;
      }
      enum ack {
        value 2;
      }
      enum investigation {
        value 3;
      }
      enum observation {
        value 4;
      }
      enum closed {
        value 5;
      }
    }
    description
      "Operator actions on alarms";
  }
```

It is of course also possible to manipulate the alarm handling list from either Java code or Javascript code running in the web browser using the `js_maapi` library.

Below follows a simple scenario to illustrate the alarm concepts. The example can be found in `examples.ncs/service-provider/simple-mpls-vpn`

```
$ make stop clean all start
$ ncs-netsim stop pe0
```

```
$ ncs-netsim stop pe1
$ ncs_cli -u admin -C
admin connected from 127.0.0.1 using console on host
admin@ncs# devices connect
...
connect-result {
    device pe0
    result false
    info Failed to connect to device pe0: connection refused
}
connect-result {
    device pe1
    result false
    info Failed to connect to device pe1: connection refused
}
...
admin@ncs# show alarms alarm-list
alarms alarm-list number-of-alarms 2
alarms alarm-list last-changed 2015-02-18T08:02:49.162436+00:00
alarms alarm-list alarm pe0 connection-failure /devices/device[name='pe0'] ""
 is-cleared              false
 last-status-change      2015-02-18T08:02:49.162734+00:00
 last-perceived-severity major
 last-alarm-text         "Failed to connect to device pe0: connection refused"
 status-change 2015-02-18T08:02:49.162734+00:00
  received-time      2015-02-18T08:02:49.162734+00:00
  perceived-severity major
  alarm-text         "Failed to connect to device pe0: connection refused"
alarms alarm-list alarm pe1 connection-failure /devices/device[name='pe1'] ""
 is-cleared              false
 last-status-change      2015-02-18T08:02:49.162436+00:00
 last-perceived-severity major
 last-alarm-text         "Failed to connect to device pe1: connection refused"
 status-change 2015-02-18T08:02:49.162436+00:00
  received-time      2015-02-18T08:02:49.162436+00:00
  perceived-severity major
  alarm-text         "Failed to connect to device pe1: connection refused"
```

In the above scenario we stop two of the devices and then ask NSO to connect to all devices. This results in two alarms for pe0 and pe1. Note that the key for the alarm is the devicename, the alarm-type and the full path to the object (in this case the device and not an object within the device) and finally an empty string for specific problem.

In the next command sequence we start the device and request NSO to connect. This will clear the alarms.

```
admin@ncs# exit
$ ncs-netsim start pe0
DEVICE pe0 OK STARTED
$ ncs-netsim start pe1
DEVICE pe1 OK STARTED
$ ncs_cli -u admin -C
$ admin@ncs# devices connect
...
connect-result {
    device pe0
    result true
    info (admin) Connected to pe0 - 127.0.0.1:10028
}
connect-result {
    device pe1
    result true
    info (admin) Connected to pe1 - 127.0.0.1:10029
```

```
}
...
admin@ncs# show alarms alarm-list
alarms alarm-list number-of-alarms 2
alarms alarm-list last-changed 2015-02-18T08:05:04.942637+00:00
alarms alarm-list alarm pe0 connection-failure /devices/device[name='pe0'] ""
 is-cleared              true
 last-status-change      2015-02-18T08:05:04.942637+00:00
 last-perceived-severity major
 last-alarm-text         "Failed to connect to device pe0: connection refused"
 status-change 2015-02-18T08:02:49.162734+00:00
  received-time      2015-02-18T08:02:49.162734+00:00
  perceived-severity major
  alarm-text         "Failed to connect to device pe0: connection refused"
 status-change 2015-02-18T08:05:04.942637+00:00
  received-time      2015-02-18T08:05:04.942637+00:00
  perceived-severity cleared
  alarm-text         "Connected as admin"
alarms alarm-list alarm pe1 connection-failure /devices/device[name='pe1'] ""
 is-cleared              true
 last-status-change      2015-02-18T08:05:04.84115+00:00
 last-perceived-severity major
 last-alarm-text         "Failed to connect to device pe1: connection refused"
 status-change 2015-02-18T08:02:49.162436+00:00
  received-time      2015-02-18T08:02:49.162436+00:00
  perceived-severity major
  alarm-text         "Failed to connect to device pe1: connection refused"
 status-change 2015-02-18T08:05:04.84115+00:00
  received-time      2015-02-18T08:05:04.84115+00:00
  perceived-severity cleared
  alarm-text         "Connected as admin"
```

Note that there are two status-change entries for the alarm and that the alarm is cleared. In the following scenario we will state that the alarm is closed and finally purge (delete) all alarms that are cleared and closed. (Again note the distinction between operator states and the states from the underlying resources.)

```
admin@ncs# alarms alarm-list alarm pe0 connection-failure /devices/device[name='pe0']
          "" handle-alarm state closed description Fixed

admin@ncs# show alarms alarm-list alarm alarm-handling

DEVICE  TYPE                STATE   USER   DESCRIPTION
---------------------------------------------------------
pe0     connection-failure  closed  admin  Fixed

admin@ncs# alarms purge-alarms alarm-handling-state-filter { state closed }
Value for 'alarm-status' [any,cleared,not-cleared]: cleared
purged-alarms 1
```

Assume you need to configure the northbound parameters. This is done using the alarm-model. A logical mapping of the connection problem above is to map it to X.733 probable cause `connectionEstablishmentError (22)` . This is done in the NSO CLI in the following way:

```
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# alarms alarm-model alarm-type connection-failure probable-cause 22
admin@ncs(config-alarm-type-connection-failure/*)# commit
Commit complete.
admin@ncs(config-alarm-type-connection-failure/*)# show full-configuration
alarms alarm-model alarm-type connection-failure *
 event-type     communicationsAlarm
 has-clear      true
```

```
kind-of-alarm  root-cause
probable-cause 22
```

CHAPTER 9

Plug-and-play Scripting

Introduction

This chapter defines a scripting mechanism to be used together with the CLI (scripting is not available for any other northbound interfaces). The chapter is intended for users that are familiar with UNIX shell scripting and/or programming. With the scripting mechanism it is possible for an end-user to add new functionality to NSO in a plug-and-play like manner. No special tools are needed. There are three categories of scripts:

`command scripts`	used to add new commands to the CLI.
`policy scripts`	invoked at validation time and may control the outcome of a transaction. Policy scripts have the mandate to cause a transaction to abort.
`post-commit scripts`	invoked when a transaction has been committed. Post-commit scripts can for example be used for logging, sending external events etc.

The terms "script" and "scripting" used throughout this description refer to how functionality can be added without a requirement for integration using the NSO programming APIs. NSO will only run the scripts as UNIX executables. Thus they may be written as shell scripts, or using some other scripting language that is supported by the OS, e.g., Python, or even be compiled code. The scripts are run with the same user id as NSO. The examples in this section are written using shell scripts as a least common denominator, but they could have been written in whatever is suitable, e.g., Python or C.

Script storage

Scripts are stored in a directory tree with a predefined structure where there is a sub-directory for each script category:

```
scripts/
```

```
command/
policy/
post-commit/
```

For all script categories it suffices to just add a valid script in the correct sub-directory in order to enable the script. See the details for each script category for how a valid script of that category is defined. Scripts with a name beginning with a dot character ('.') are ignored.

The directory path to the location of the scripts is configured with the */ncs-config/scripts/dir* configuration parameter. It is possible to have several scripts directories. The sample `ncs.conf` file that comes with the NSO release specifies two script directories: `./scripts` and `${NCS_DIR}/scripts`.

Script interface

All scripts are required to provide a formal description of their interface. When the scripts are loaded, NSO will invoke the scripts with (one of)

```
--command
--policy
--post-commit
```

as argument depending of the script category.

The script must respond by writing its formal interface description on `stdout` and exit normally. Such a description consists of one or more sections. Which sections that are required depends on the category of the script.

The sections do however have a common syntax. Each section begins with the keyword "begin" followed by the type of section. After that one or more lines of settings follows. Each such setting begins with a name, followed by a colon character (':') and after that the value is stated. The section ends with the keyword "end". Empty lines and spaces may be used to improve the readability.

For examples see each corresponding section below.

Loading of scripts

Scripts are automatically loaded at startup and may also be manually reloaded with the CLI command **script reload**. The command takes an optional *verbosity* parameter which may have one of the following values:

`diff` Shows info about those scripts that have been changed since the latest (re)load. This is the default.

`all` Shows info about all scripts regardless of whether they have been changed or not.

`errors` Shows info about those scripts that are erroneous, regardless of whether they have been changed or not. Typical errors are invalid file permissions and syntax errors in the interface description.

Yet another parameter may be useful when debugging reload of scripts:

`debug` Shows additional debug info about the scripts.

An example session reloading scripts:

```
admin@ncs# script reload all
$NCS_DIR/examples.ncs/getting-started/using-ncs/7-scripting/scripts:
ok
command:
```

```
    add_user.sh: unchanged
    echo.sh: unchanged
policy:
    check_dir.sh: unchanged
post-commit:
    show_diff.sh: unchanged
/opt/ncs/scripts: ok
command:
    device_brief.sh: unchanged
    device_brief_c.sh: unchanged
    device_list.sh: unchanged
    device_list_c.sh: unchanged
    device_save.sh: unchanged
```

Command scripts

Command scripts are used to add new commands to the CLI. The scripts are executed in the context of a transaction. When the script is run in `oper` mode, this is a read-only transaction, when it is run in `config` mode, it is a read-write transaction. In that context the script may make use of the environment variables `NCS_MAAPI_USID` and `NCS_MAAPI_THANDLE` in order to attach to the active transaction. This makes it simple to make use of the **ncs-maapi** command (see the ncs-maapi(1) in *Manual Pages* manual page) for various purposes.

Each command script must be able to handle the argument `--command` and, when invoked, write a `command` section to `stdout`. If the CLI command is intended to take parameters, one `param` section per CLI parameter must also be emitted.

The command is not paginated by default in the CLI and will only do so if it is piped to more.

```
joe@io> example_command_script | more
```

Command section

The following settings can be used to define a command:

`modes` Defines in which CLI mode(s) that the command should be available. The value can be `oper`, `config` or both (separated with space).

`styles` Defines in which CLI styles that the command should be available. The value can be one or more of `c`, `i` and `j` (separated with space). `c` means Cisco style, `i`, means Cisco IOS and `j` for J-style.

`cmdpath` Is the full CLI command path. For example the command path `my script echo` implies that the command will be called `my script echo` in the CLI.

`help` Command help text.

An example of a command section is:

```
begin command
  modes: oper
  styles: c i j
  cmdpath: my script echo
  help: Display a line of text
end
```

Param section

In this section various aspects of a parameter is specified. This may both affect the parameter syntax for the end-user in the CLI as well as what the command script will get as arguments. The following settings can be used to customize each CLI parameter:

`name`	Optional name of the parameter. If provided, the CLI will prompt for this name before the value. By default the name is not forwarded to the script. See `flag` and `prefix`.
`type`	The type of the parameter. By default each parameter has a value, but by setting the type to `void` the CLI will not prompt for a value. In order to be useful the `void` type must be combined with `name` and either `flag` or `prefix`.
`presence`	Controls whether the parameter must be present in the CLI input or not. Can be set to `optional` or `mandatory`.
`words`	Controls the number of words that the parameter value may consist of. By default the value must consist of just one word (possibly quoted if it contains spaces). If set to `any`, the parameter may consist of any number of words. This setting is only valid for the last parameter.
`flag`	Extra word added before the parameter value. For example if set to `-f` and the user enters `logfile`, the script will get `-f logfile` as arguments.
`prefix`	Extra string prepended to the parameter value (as a single word). For example if set to `--file=` and the user enters `logfile`, the script will get `--file=logfile` as argument.
`help`	Parameter help text.

If the command takes a parameter to redirect the output to a file, a param section might look like this:

```
begin param
 name: file
 presence: optional
 flag: -f
 help: Redirect output to file
end
```

Full command example

A command denying changes the configured trace-dir for a set of devices it can use the `check_dir.sh` script.

```
#!/bin/bash

set -e

while [ $# -gt 0 ]; do
    case "$1" in
        --command)
            # Configuration of the command
            #
            # modes   - CLI mode (oper config)
            # styles  - CLI style (c i j)
            # cmdpath - Full CLI command path
            # help    - Command help text
            #
            # Configuration of each parameter
            #
            # name     - (optional) name of the parameter
            # more     - (optional) true or false
            # presence - optional or mandatory
            # type     - void - A parameter without a value
            # words    - any - Multi word param. Only valid for the last param
            # flag     - Extra word added before the parameter value
            # prefix   - Extra string prepended to the parameter value
            # help     - Command help text
            cat << EOF

begin command
```

```
  modes: config
  styles: c i j
  cmdpath: user-wizard
  help: Add a new user
end
EOF
            exit
            ;;
        *)
            break
            ;;
    esac
    shift
done

## Ask for user name
while true; do
    echo -n "Enter user name: "
    read user

    if [ ! -n "${user}" ]; then
        echo "You failed to supply a user name."
    elif ncs-maapi --exists "/aaa:aaa/authentication/users/user{${user}}"; then
        echo "The user already exists."
    else
        break
    fi
done

## Ask for password
while true; do
    echo -n "Enter password: "
    read -s pass1
    echo

    if [ "${pass1:0:1}" == "$" ]; then
        echo -n "The password must not start with $. Please choose a "
        echo    "different password."
    else
        echo -n "Confirm password: "
        read -s pass2
        echo

        if [ "${pass1}" != "${pass2}" ]; then
            echo "Passwords do not match."
        else
            break
        fi
    fi
done

groups=`ncs-maapi --keys "/nacm/groups/group"`
while true; do
    echo "Choose a group for the user."
    echo -n "Available groups are: "
    for i in ${groups}; do echo -n "${i} "; done
    echo
    echo -n "Enter group for user: "
    read group

    if [ ! -n "${group}" ]; then
        echo "You must enter a valid group."
```

```
    else
        for i in ${groups}; do
            if [ "${i}" == "${group}" ]; then
                # valid group found
                break 2;
            fi
        done
        echo "You entered an invalid group."
    fi
    echo
done

echo "Creating user"

ncs-maapi --create "/aaa:aaa/authentication/users/user{${user}}"
ncs-maapi --set "/aaa:aaa/authentication/users/user{${user}}/password" \
                "${pass1}"

echo "Setting home directory to: /homes/${user}"
ncs-maapi --set "/aaa:aaa/authentication/users/user{${user}}/homedir" \
            "/homes/${user}"

echo "Setting ssh key directory to: /homes/${user}/ssh_keydir"
ncs-maapi --set "/aaa:aaa/authentication/users/user{${user}}/ssh_keydir" \
            "/homes/${user}/ssh_keydir"

ncs-maapi --set "/aaa:aaa/authentication/users/user{${user}}/uid" "1000"
ncs-maapi --set "/aaa:aaa/authentication/users/user{${user}}/gid" "100"

echo "Adding user to the ${group} group."
gusers=`ncs-maapi --get "/nacm/groups/group{${group}}/user-name"`

for i in ${gusers}; do
    if [ "${i}" == "${user}" ]; then
        echo "User already in group"
        exit 0
    fi
done

ncs-maapi --set "/nacm/groups/group{${group}}/user-name" "${gusers} ${user}"
```

Calling `$NCS_DIR/examples.ncs/getting-started/using-ncs/7-scripting/scripts/command/echo.sh` with the argument `--command` argument produces a command section and a couple of param sections:

```
$ ./echo.sh --command
begin command
  modes: oper
  styles: c i j
  cmdpath: my script echo
  help: Display a line of text
end

begin param
 name: nolf
 type: void
 presence: optional
 flag: -n
 help: Do not output the trailing newline
end

begin param
```

```
 name: file
 presence: optional
 flag: -f
 help: Redirect output to file
end

begin param
 presence: mandatory
 words: any
 help: String to be displayed
end
```

In the complete example `$NCS_DIR/examples.ncs/getting-started/using-ncs/7-scripting` there is a `README` file and a simple command script `scripts/command/echo.sh`.

Policy scripts

Policy scripts are invoked at validation time, before a change is committed. A policy script can reject the data, accept it, or accept it with a warning. If a warning is produced, it will be displayed for interactive users (e.g. through the CLI or Web UI). The user may choose to abort or continue to commit the transaction.

Policy scripts are typically assigned to individual leafs or containers. In some cases it may be feasible to use a single policy script, e.g. on the top level node of the configuration. In such a case, this script is responsible for the validation of all values and their relationships throughout the configuration.

All policy scripts are invoked on every configuration change. The policy scripts can be configured to depend on certain subtrees of the configuration, which can save time but it is very important that all dependencies are stated and also updated when the validation logic of the policy script is updated. Otherwise an update may be accepted even though a dependency should have denied it.

There can be multiple dependency declarations for a policy script. Each declaration consists of a dependency element specifying a configuration subtree that the validation code is dependent upon. If any element in any of the subtrees is modified, the policy script is invoked. A subtree is specified as an absolute path.

If there are no declared dependencies, the root of the configuration tree (/) is used, which means that the validation code is executed when any configuration element is modified. If dependencies are declared on a leaf element, an implicit dependency on the leaf itself is added.

Each policy script must handle the argument `--policy` and, when invoked, write a `policy` section to `stdout`. The script must also perform the actual validation when invoked with the argument `--keypath`.

Policy section

The following settings can be used to configure a policy script:

`keypath`	Mandatory. Keypath is a path to a node in the configuration data tree. The policy script will be associated with this node. The path must be absolute. A keypath can for example be `/devices/device/c0`. The script will be invoked if the configuration node, referred to by by the keypath, is changed or if any node in the subtree under the node (if the node is a container or list) is changed.
`dependency`	Declaration of a dependency. The dependency must be an absolute keypath. Multiple dependency settings can be declared. Default is `/`.

`priority`	An optional integer parameter specifying the order policy scripts will be evaluated, in order of increasing priority, where lower value is higher priority. The default priority is `0`.
`call`	This optional setting can only be used if the associated node, declared as `keypath`, is a list. If set to `once`, the policy script is only called once even though there exists many list entries in the data store. This is useful if we have a huge amount of instances or if values assigned to each instance have to be validated in comparison with its siblings. Default is `each`.

A policy that will be run for every change on or under /devices/device.

```
begin policy
 keypath: /devices/device
 dependency: /devices/global-settings
 priority: 4
 call: each
end
```

Validation

When NSO has come to the conclusion that the policy script should be invoked to perform its validation logic, the script is invoked with the option `--keypath`. If the registered node is a leaf, its value will be given with the `--value` option. For example `--keypath /devices/device/c0` or if the node is a leaf `--keypath /devices/device/c0/address --value 127.0.0.1`.

Once the script has performed its validation logic it must exit with a proper status. The following exit statuses are valid:

`0` Validation ok. Vote for commit.

`1` When the outcome of the validation is dubious it is possible for the script to issue a warning message. The message is extracted from the script output on stdout. An interactive user has the possibility to choose to abort or continue to commit the transaction. Non-interactive users automatically vote for commit.

`2` When the validation fails it is possible for the script to issue an error message. The message is extracted from the script output on stdout. The transaction will be aborted.

Full policy example

A policy denying changes the configured trace-dir for a set of devices it can use the `check_dir.sh` script.

```
#!/bin/sh

usage_and_exit() {
    cat << EOF
Usage: $0 -h
       $0 --policy
       $0 --keypath <keypath> [--value <value>]

  -h                     display this help and exit
  --policy               display policy configuration and exit
  --keypath <keypath>    path to node
  --value <value>        value of leaf

Return codes:

  0 - ok
  1 - warning message is printed on stdout
```

```
  2 - error message   is printed on stdout
EOF
    exit 1
}

while [ $# -gt 0 ]; do
    case "$1" in
        -h)
            usage_and_exit
            ;;
        --policy)
            cat << EOF
begin policy
  keypath: /devices/global-settings/trace-dir
  dependency: /devices/global-settings
  priority: 2
  call: each
end
EOF
            exit 0
            ;;
        --keypath)
            if [ $# -lt 2 ]; then
                echo "<ERROR> --keypath <keypath> - path omitted"
                usage_and_exit
            else
                keypath=$2
                shift
            fi
            ;;
        --value)
            if [ $# -lt 2 ]; then
                echo "<ERROR> --value <value> - leaf value omitted"
                usage_and_exit
            else
                value=$2
                shift
            fi
            ;;
        *)
            usage_and_exit
            ;;
    esac
    shift
done

if [ -z "${keypath}" ]; then
    echo "<ERROR> --keypath <keypath> is mandatory"
    usage_and_exit
fi

if [ -z "${value}" ]; then
    echo "<ERROR> --value <value> is mandatory"
    usage_and_exit
fi

orig="./logs"
dir=${value}
# dir=`ncs-maapi --get /devices/global-settings/trace-dir`
if [ "${dir}" != "${orig}" ] ; then
    echo "/devices/global-settings/trace-dir: must retain it original value (${orig})"
    exit 2
```

```
fi
```

Trying to change that parameter would result in an aborted transaction

```
admin@ncs(config)# devices global-settings trace-dir ./testing
admin@ncs(config)# commit
Aborted: /devices/global-settings/trace-dir: must retain it original
value (./logs)
```

In the complete example `$NCS_DIR/examples.ncs/getting-started/using-ncs/7-scripting/` there is a `README` file and a simple policy script `scripts/policy/check_dir.sh`.

Post-commit scripts

Post-commit scripts are run when a transaction has been committed, but before any locks have been released. The transaction hangs until the script has returned. The script cannot change the outcome of the transaction. Post-commit scripts can for example be used for logging, sending external events etc. The scripts run as the same user id as NSO.

The script is invoked with *`--post-commit`* at script (re)load. In future releases it is possible that the `post-commit` section will be used for control of post-commit scripts behavior.

At post-commit, the script is invoked without parameters. In that context the script may make use of the environment variables `NCS_MAAPI_USID` and `NCS_MAAPI_THANDLE` in order to attach to the active (read-only) transaction.

This makes it simple to make use of the **ncs-maapi** command. Especially the command **ncs-maapi --keypath-diff /** may turn out to be useful, as it provides a listing of all updates within the transaction on a format that is easy to parse.

Post-commit section

All post-commit scripts must be able to handle the argument `--post-commit` and, when invoked, write an empty `post-commit` section to `stdout`:

```
begin post-commit
end
```

Full post-commit example

Assume the administrator of a system would want to have a mail each time a change is performed on the system, a script such as `mail_admin.sh`:

```
#!/bin/bash

set -e

if [ $# -gt 0 ]; then
    case "$1" in
        --post-commit)
            cat &lt;&lt; EOF
begin post-commit
end
EOF
            exit 0
            ;;
        *)
            echo
            echo "Usage: $0 [--post-commit]"
```

```
            echo
            echo "  --post-commit Mandatory for post-commit scripts"
            exit 1
            ;;
    esac
else
    file="mail_admin.log"
    NCS_DIFF=$(ncs-maapi --keypath-diff /)
    mail -s "NCS Mailer" admin@example.com &lt;&lt;EOF
AutoGenerated mail from NCS

$NCS_DIFF
EOF
fi
```

If the admin then loads this script

```
admin@ncs# script reload debug
$NCS_DIR/examples.ncs/getting-started/using-ncs/1-simulated-cisco-ios/scripts:
ok
    post-commit:
        mail_admin.sh: new
--- Output from
$NCS_DIR/examples.ncs/getting-started/using-ncs/1-simulated-cisco-ios/scripts/post-commit/mai
--post-commit ---
1: begin post-commit
2: end
3:
---
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# devices global-settings trace-dir ./again
admin@ncs(config)# commit
Commit complete.
```

This configuration change will produce an email to admin@example.com with subject `NCS Mailer` and body

```
AutoGenerated mail from NCS
value set  : /devices/global-settings/trace-dir
```

In the complete example `$NCS_DIR/examples.ncs/getting-started/using-ncs/7-scripting/` there is a `README` file and a simple post-commit script `scripts/post-commit/show_diff.sh`.

CHAPTER 10

Compliance reporting

Introduction

When the network configuration is broken there is a need to gather information and verify the network.

NSO has numerous functions to show different aspects of such a network configuration 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 a correctly configured network? Where is the authoritative configuration kept? Naturally, NSO, with the configurations stored in CDB, is the authority. Checking the live devices against the NSO stored device configuration is a fundamental part of compliance reporting. Compliance reporting can also be based on one or a number of stored templates which the live devices are compared against. The compliance reports can also be a combination of both approaches.

Compliance reporting can be configured to check the current situation or checking historic events, or both. To assemble historic events, rollback files are used. Therefore this functionality must have been enabled in NSO prior to report execution or else no history view can be presented.

The reports can be created in either plain text, HTML, or DocBook XML format. In addition, the data can also be exported to a SQLite database file. The DocBook XML format allows you to use the report in further post-processing, such as creating a PDF using Apache FOP and your own custom styling.

Reports can be generated using the CLI or Web UI. In the NSO Web UI, compliance reporting options are available under the 'Compliance reporting' card. The CLI options are described in the sections below.

Creating compliance report definitions

It is possible to create several named compliance report definitions. Each named report defines the devices, services and/or templates that should be part of the network configuration verification.

Let us walk through a simple compliance report definition. This example is based on the `examples.ncs/service-provider/mpls-vpn` example. For the details of the included services and devices in this example see the `README` file.

First of all the reports has a name which is key in the *report* list. Furthermore the report has a *device-check* and a *service-check* container for specifying devices and services to check. The *compare-template* list allows for specifying templates to compare device configurations against. A report definition can specify all containers at the same time:

```
$ ncs_cli -u admin -C
admin connected from 127.0.0.1 using console on ncs
ncs# config
Entering configuration mode terminal
ncs(config)# compliance reports report gold-check
Possible completions:
  compare-template   Diff devices against templates
  device-check       Report on devices
  run                Run this compliance report
  service-check      Report on services out of sync
  <cr>
```

We will first use the *device-check* container to specify which devices to check. Devices can be defined in one of 4 different ways:

- *all-devices* - Check all defined devices
- *device-group* - Specified list of device groups
- *device* - Specified list of devices
- *select-devices* - Specified by an XPath expression

Furthermore, for a *device-check* the behavior or the verification can be specified. The default behavior for device verification is the following:

- To request a check-sync action to verify that the device is currently in sync. This behavior is controlled by the leaf *current-out-of-sync* (default true).
- To scan commit log (i.e. rollback files) for changes on the devices and report these. This behavior is controlled by the leaf *historic-changes* (default true).

```
ncs(config)# compliance reports report gold-check
ncs(config-report-gold-check)# device-check
Possible completions:
  all-devices           Report on all devices
  current-out-of-sync   Should current check-sync action be performed?
  device                Report on specific devices
  device-group          Report on specific device groups
  historic-changes      Include commit log events from within the report
                        interval
  select-devices        Report on devices selected by an XPath expression
  <cr>
```

We will choose the default behaviour and check all devices:

```
ncs(config-report-gold-check)# device-check all-devices
```

In our example we also use the *service-check* container to specify which services to check. Services can be defined in one of 3 ways:

- *all-services* - Check all defined services

- *service* - Specified list of services
- *select-services* - Specified by an XPath expression

Also for the *service-check* the verification behavior can be specified. The default behavior for service verification is the following:

- To request a check-sync action to verify that the service is currently in sync. This behavior is controlled by the leaf *current-out-of-sync* (default true).
- To scan commit log (i.e. rollback files) for changes on the services and report these. This behavior is controlled by the leaf *historic-changes* (default true).

```
ncs(config-report-gold-check)# service-check
Possible completions:
  all-services          Report on all services
  current-out-of-sync   Should current check-sync action be performed?
  historic-changes      Include commit log events from within the report
                        interval
  select-services       Report on services selected by an XPath expression
  service               Report on specific services
  <cr>
```

In our report we choose the default behavior and to check the l3vpn service:

```
ncs(config-report-gold-check)# service-check select-services /l3vpn:vpn/l3vpn:l3vpn
ncs(config-report-gold-check)# commit
Commit complete.
ncs(config-report-gold-check)# show full-configuration
compliance reports report gold-check
 device-check all-devices
 service-check select-services /l3vpn:vpn/l3vpn:l3vpn
!
```

Our next example will illustrate how to add a device template in the compliance report. This template will be used to compare some part of the device configuration against. First we define the device template:

```
ncs(config-report-gold-check)# top
ncs(config)# devices template gold-conf config
ncs(config-config)# ios:snmp-server community {$COMMUNITY}
ncs(config-community-{$COMMUNITY})# commit
Commit complete.
ncs(config-community-{$COMMUNITY})# show full-configuration
devices template gold-conf
 config
  ios:snmp-server community {$COMMUNITY}
  !
 !
!
```

We will also need a device group which will be used later in the report definition. For the sake of simplicity, in this example we will just choose some of the ce devices:

```
ncs(config-community-{$COMMUNITY})# top
ncs(config)# devices device-group mygrp device-name
(list): [ce0 ce1 ce2 ce3]
ncs(config-device-group-mygrp)# commit
Commit complete.
```

Now we add the template to the already defined report `gold-check`. An entry in the *compare-template* list contain the combination of a template and a device-group which imply that the template will be applied to all devices in the device group and the difference (if any) will be reported as a compliance violation. Note, that no data will be changed on the device. Since the device template can contain variables, each *compare-template* also has a *variable* list.

In our example report we use the `gold-conf` template and the `mygrp` group:

```
ncs(config-device-group-mygrp)# top
ncs(config)# compliance reports report gold-check
ncs(config-report-gold-check)# compare-template gold-conf mygrp
```

Since the `gold-conf` template uses variables, we will set the values for this variable in the report:

```
ncs(config-compare-template-gold-conf/mygrp)# variable COMMUNITY
ncs(config-variable-COMMUNITY)# value 'public'
ncs(config-variable-COMMUNITY)# show configuration
compliance reports report gold-check
 compare-template gold-conf mygrp
  variable COMMUNITY
   value 'public'
  !
 !
!
ncs(config-variable-COMMUNITY)# commit
Commit complete.
```

Running compliance reports

Compliance reporting is a read-only operation. When running a compliance report the result is stored in a file located in a sub directory `compliance-reports` under the NSO `state` directory. NSO has operational data for managing this report storage which makes it possible to list existing reports. Here is an example of such report listing:

```
ncs(config-variable-COMMUNITY)# top
ncs(config)# exit
ncs# show compliance report-results
compliance report-results report 1
 name              gold-check
 title             "GOLD NW 1"
 time              2015-02-04T18:48:57+00:00
 who               admin
 compliance-status violations
 location          http://.../report_1_admin_1_2015-2-4T18:48:57:0.xml
compliance report-results report 2
 name              gold-check
 title             "GOLD NW 2"
 time              2015-02-04T18:51:48+00:00
 who               admin
 compliance-status violations
 location          http://.../report_2_admin_1_2015-2-4T18:51:48:0.text
compliance report-results report 3
 name              gold-check
 title             "GOLD NW 3"
 time              2015-02-04T19:11:43+00:00
 who               admin
 compliance-status violations
 location          http://.../report_3_admin_1_2015-2-4T19:11:43:0.text
```

There is also a *remove* action to remove report results (and the corresponding file):

```
ncs# compliance report-results report 2..3 remove
ncs# show compliance report-results
```

```
compliance report-results report 1
 name             gold-check
 title            "GOLD NW 1"
 time             2015-02-04T18:48:57+00:00
 who              admin
 compliance-status violations
 location         http://.../report_1_admin_1_2015-2-4T18:48:57:0.xml
```

When running the report there are a number of parameters that can be be specified with the specific *run* action.

The parameters that are possible to specify for a report *run* action are:

- *title* - The title in the resulting report
- *from* - The date and time from which the report should start the information gathering. If not set, the oldest available information is implied.
- *to* - The date and time to where the information gathering should stop. If not set, the current date and time is implied. If set, no new check-syncs of devices and/or services will be attempted.
- *outformat* - One of xml, html, text, or sqlite. If xml is specified the report will formatted using the docbook schema.

We will request a report run with a title and formatted as text.

```
ncs# compliance reports report gold-check run \
> title "My First Report" outformat text
```

In the above command the report was run without a *from* or a *to* argument. This implies that historical information gathering will be based on all available information. This include information gathered from rollback files.

When a *from* argument is supplied to a compliance report run action, this implies that only historical information younger than the *from* date and time is checked.

```
ncs# compliance reports report gold-check run \
> title "First check" from 2015-02-04T00:00:00
```

When a *to* argument is supplied this implies that historic information will be gathered for all logged information up to the date and time of the *to* argument.

```
ncs# compliance reports report gold-check run \
> title "Second check" to 2015-02-05T00:00:00
```

The *from* and a *to* arguments can be combined to specify a fixed historic time interval.

```
ncs# compliance reports report gold-check run \
> title "Third check" from 2015-02-04T00:00:00 to 2015-02-05T00:00:00
```

When a compliance report is run the action will respond with a flag indicating if any discrepancies were found. Also it reports how many devices and services have been verified in total by the report.

```
ncs# compliance reports report gold-check run \
> title "Fourth check" outformat text
time 2015-2-4T20:42:45.019012+00:00
compliance-status violations
info Checking 17 devices and 2 services
location http://.../report_7_admin_1_2015-2-4T20:42:45.019012+00:00.text
```

Below is an example of a compliance report result (in text format):

Example 45. Compliance report result

```
$ cat ./state/compliance-reports/report_7_admin_1_2015-2-4T20\:42\:45.019012+00\:00.text
reportcookie : g2gCbQAAAAtGaWZ0aCBjaGVja20AAAAKZ29sZC1jaGVjaw==

Compliance report : Fourth check

        Publication date : 2015-2-4 20:42:45
        Produced by user : admin

Chapter : Summary

        Compliance result titled "Fourth check" defined by report "gold-check"
        Resulting in violations
        Checking 17 devices and 2 services
        Produced 2015-2-4 20:42:45
        From : Oldest available information
        To : 2015-2-4 20:42:45

Devices out of sync

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


Template discrepancies

gold-conf

        Discrepancies in device
        ce0
        ce1
        ce2
        ce3
```

```
Chapter : Details


Commit list

        SeqNo   ID      User    Client  Timestamp            Label  Comment
        0       10031   admin   cli     2015-02-04 20:31:42
        1       10030   admin   cli     2015-02-04 20:03:41
        2       10029   admin   cli     2015-02-04 19:54:40
        3       10028   admin   cli     2015-02-04 19:45:20
        4       10027   admin   cli     2015-02-04 18:38:05


Service commit changes

        No service data commits saved for the time interval


Device commit changes

        No device data commits saved for the time interval


Service differences

        No service data diffs found


Template discrepancies details

gold-conf

Device ce0

 config {
     ios:snmp-server {
+        community public {
+        }
     }
 }

Device ce1

 config {
     ios:snmp-server {
+        community public {
+        }
     }
 }

Device ce2

 config {
     ios:snmp-server {
+        community public {
+        }
     }
 }

Device ce3

 config {
```

```
     ios:snmp-server {
+        community public {
+        }
     }
 }
```

Additional configuration checks

In some cases it is insufficient to only check that the required configuration is present, as it is possible for other configuration on the device to interfere with the desired functionality. For example, a service may configure a routing table entry for the 198.51.100.0/24 network. If someone also configures a more specific entry, say 198.51.100.0/28, that entry will take precedence and may interfere with the way the service requires the traffic to be routed. In effect, this additional configuration can render the service inoperable.

To help operators ensure there is no such extraneous configuration on the managed devices, compliance reporting feature supports so-called *strict* mode. This mode not only checks the required configuration is present but also reports any configuration present on the device that is not part of the template.

You can configure this mode in the report definition, when specifying the device template to check against, for example:

```
ncs(config)# compliance reports report gold-check
ncs(config-report-gold-check)# compare-template gold-conf mygrp strict
```

However, in practice, using the strict mode with device templates may prove challenging. Often, each device will have its own set of IP addresses configured. While you can supply variable values to the template, you likely need to maintain a separate set for each device (since each uses its own unique IPs).

One way to overcome this problem is to use services to configure all aspects of the managed device. But perhaps you only use NSO to configure a subset of all the services (configuration) on the devices. In this case, you can still perform generic configuration validation with the help of compliance templates, which are similar to, but separate from device templates.

With compliance templates you use regular expressions to check compliance, instead of simple fixed values or variables, and they can be use with or without strict mode.

You can create a compliance template from scratch, similarly to how you create a device template.
To check that the router uses only internal DNS servers from the 10.0.0.0/8 range, you might create a compliance template such as:

```
admin@ncs(config)# compliance template internal-dns
admin@ncs(config-template-internal-dns)# ned-id router-nc-1.0 config sys dns server 10\\\\..+
```

Here, the value for the `/sys/dns/server` must start with "10.", followed by any string (regular expression ".+"). Since dot has special meaning with regular expressions (any character), it must be escaped with backslash to match only the actual dot character. But note the required multiple escaping ("\\\
\") in this case.

As these expressions can be non-trivial to construct, the templates have a **check** command that allows you to quickly check compliance for a set of devices, which is a great development aid.

```
admin@ncs(config)# show full-configuration devices device ex0 config sys dns server
devices device ex0
 config
  sys dns server 10.2.3.4
  !
  sys dns server 192.168.100.10
```

```
  !
 !
!
admin@ncs(config)# compliance template internal-dns
admin@ncs(config-template-internal-dns)# check device ex0
check-result {
    device ex0
    result violations
    diff  config {
     sys {
         dns {
+            # after server 10.2.3.4
+            /* No match of 10\\..+ */
+            server 192.168.100.10;
         }
     }
 }

}
```

Alternatively, you can use the `/compliance/create-template` action when you already have existing device templates that you would like to use as a starting point for a compliance template.

Finally, to use compliance templates in a report, reference them from `device-check/template`:

```
admin@ncs(config-report-gold-check)# device-check template internal-dns
```

CHAPTER 11

NSO Packages

Overview

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 upgraded in run-time.

Listing Packages

The currently loaded packages can be viewed with the following command:

Example 46. Show Currently Loaded Packages

```
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

 build-info date "2015-01-29 23:40:12"
 build-info file ncs-3.4_HEAD-cisco-ios-3.0.tar.gz
 build-info arch linux.x86_64
 build-info java "compiled Java class data, version 50.0 (Java 1.6)"
 build-info package name cisco-ios
 build-info package version 3.0
 build-info package ref 3.0
 build-info package sha1 a8f1329
 build-info ncs version 3.4_HEAD
 build-info ncs sha1 81a1e4c
```

```
 build-info dev-support version 0.99
 build-info dev-support branch e4d3fa7
 build-info dev-support sha1 e4d3fa7
 oper-status up
```

Thus the above command shows that NSO currently has only one package loaded, the NED package for Cisco IOS. The output includes the name and version of the package, the minimum required NSO version, the Java components included, package build details, and finally the operational status of the package. The operational status is of particular importance - if it is anything other than `up`, it indicates that there was a problem with the loading or the initialization of the package. In this case an item `error-info` may also be present, giving additional information about the problem. To show only the operational status for all loaded packages, this command can be used:

```
admin@ncs# show packages package * oper-status
packages package cisco-ios
 oper-status up
```

CHAPTER 12

Life-cycle Operations - Manipulating Existing Services and Devices

Commit flags and device & service actions

Devices and services are the most important entities in NSO. Once created they may be manipulated in several different ways. The three main categories of operations that affect the state of services and devices are

Commit flags	Commit flags modifies the transaction semantics
Device actions	Explicit actions that modifies the devices
Service actions	Explicit actions that modifies the services

The purpose of this chapter is more of a quick reference guide, an enumeration of commonly used commands. The context in which these commands should be used is found in other parts of the documentation.

Commit Flags

Commit flags may be present when issuing a **commit** command:

```
commit <flag>
```

Some of these flags may be configured to apply globally for all commits, under `/devices/global-settings`, or per device profile, under `/devices/profiles`.

Some of the more important flags are:

`and-quit`	Exit to (CLI operational mode) after commit.
`check`	Validate the pending configuration changes. Equivalent to **validate** command (See Chapter 5, *NSO CLI*).
`comment | label`	Add a commit comment/label visible in compliance reports, rollback files etc.

`dry-run`
: Validate and display the configuration changes but do not perform the actual commit. *Neither* CDB *nor* the devices are affected. Instead the effects that would have taken place is shown in the returned output. The output format can be set with the `outformat` option. Possible output formats are: *xml*, *cli* and *native*.

The *xml* format displays all changes in the whole data model. The changes will be displayed in NETCONF XML edit-config format, i.e., the edit-config that would be applied locally (at NCS) to get a config that is equal to that of the managed device.

The *cli* format displays all changes in the whole data model. The changes will be displayed in CLI curly bracket format.

The *native* format displays only changes under `/devices/device/config`. The changes will be displayed in native device format.

The *native* format can be used with the `reverse` option to display the device commands for getting back to the current running state in the network if the commit is successfully executed. Beware that if any changes are done later on the same data the reverse device commands returned are invalid.

`no-networking`
: Validate the configuration changes, update the CDB but *do not* update the actual devices.

This is equivalent to first setting the admin-state to southbound locked, then issuing a standard commit. In both cases the configuration changes are prevented from being sent to the actual devices.

Warning If the commit implies changes, it will make the device out-of-sync.

The **sync-to** command can then be used to push the change to the network.

`no-out-of-sync-check`
: Commit even if the device is out-of-sync. This can be used in scenarios where you know that the change you are doing is not in conflict with what is on the device and do not want to perform the action **sync-from** first. Verify result by using the action **compare-config**

Warning The device's sync state is assumed to be unknown after such commit and the stored last-transaction-id value is cleared.

`no-overwrite`
: NSO will check that the data that should be modified has not changed on the device compared to NSO's view of the data. This is a fine-granular sync check; NSO verifies that NSO and the device is in sync regarding the data that will be modified. If they are not in sync, the transaction is aborted.

This parameter is particularly useful in brown field scenarios where the device always is out of sync due to being directly modified by operators or other management systems.

Warning The device's sync state is assumed to be unknown after such commit and the stored last-transaction-id value is cleared.

no-revision-drop
: Fail if one or more devices have obsolete device models.

  When NSO connects to a managed device the version of the device data model is discovered. Different devices in the network might have different versions. When NSO is requested to send configuration to devices, NSO defaults to drop any configuration that only exists in later models than the device supports. This flag forces NSO to never silently drop any data set operations towards a device.

no-deploy
: Commit without invoking the service create method, i.e, write the service instance data without activating the service(s). The service(s) can later be re-deployed to write the changes of the service(s) to the network.

reconcile
: Reconcile the service data. All data which existed before the service was created will now be owned by the service. When the service is removed that data will also be removed. In technical terms the reference count will be decreased by one for everything which existed prior to the service.

  If manually configured data exists below in the configuration tree that data is kept unless the option *discard-non-service-config* is used.

use-lsa
: Force handling of the LSA nodes as such. This flag tells NSO to propagate applicable commit flags and actions to the LSA nodes without applying them on the upper NSO node itself. The commit flags affected are *dry-run*, *no-networking*, *no-out-of-sync-check*, *no-overwrite* and *no-revision-drop*.

no-lsa
: Do not handle any of the LSA nodes as such. These nodes will be handled as any other device.

commit-queue
: Commit through the commit queue (the section called “Commit Queue”).

  While the configuration change is committed to CDB immediately it is not committed to the actual device but rather queued for eventual commit in order to increase transaction throughput.

  This enables use of the commit queue feature for individual **commit** commands without enabling it by default.

  Possible operation modes are: *async*, *sync* and *bypass*.

  If the *async* mode is set the operation returns successfully if the transaction data has been successfully placed in the queue.

  The *sync* mode will cause the operation to not return until the transaction data has been sent to all devices, or a timeout occurs. If the timeout occurs the transaction data stays in the queue and the operation returns successfully. The timeout value can be specified with the *timeout* or *infinity* option. By default the timeout value is determined by what is configured in `/devices/global-settings/commit-queue/sync`.

  The *bypass* mode means that if `/devices/global-settings/commit-queue/enabled-by-default` *true* the data in this transaction will bypass the commit queue. The data will be written directly to the devices. The operation will still fail if the commit queue contains one or more entries affecting the same device(s) as the transaction to be committed.

In addition the *`commit-queue`* flag has a number of other useful options that affects the resulting queue item.

The *`tag`* option sets a user defined opaque tag that is present in all notifications and events sent referencing the queue item.

The *`block-others`* option will cause the resulting queue item to block subsequent queue items which use any of the devices in this queue item, from being queued.

The *`lock`* option will place a lock on the resulting queue item. The queue item will not be processed until it has been unlocked, see the actions *`unlock`* and *`lock`* in `/devices/commit-queue/queue-item`. No following queue items, using the same devices, will be allowed to execute as long as the lock is in place.

The *`atomic`* option sets the atomic behaviour of the resulting queue item. If this is set to *false*, the devices contained in the resulting queue item can start executing if the same devices in other non-atomic queue items ahead of it in the queue are completed. If set to *true*, the atomic integrity of the queue item is preserved.

Depending on the selected *`error-option`* NSO will store the reverse of the original transaction to be able to undo the transaction changes and get back to the previous state. This data is stored in the `/devices/commit-queue/completed` tree from where it can be viewed and invoked with the **rollback** action. When invoked the data will be removed. Possible values are: *continue-on-error*, *rollback-on-error* and *stop-on-error*. The *continue-on-error* value means that the commit queue will continue on errors. No rollback data will be created. The *rollback-on-error* value means that the commit queue item will roll back on errors. The commit queue will place a lock on the failed queue item, thus blocking other queue items with overlapping devices to be executed. The **rollback** action will then automatically be invoked when the queue item has finished its execution. The lock will be removed as part of the rollback. The *stop-on-error* means that the commit queue will place a lock on the failed queue item, thus blocking other queue items with overlapping devices to be executed. The lock must then either manually be released when the error is fixed or the **rollback** action under `/devices/commit-queue/completed` be invoked.

Note Read about error recovery in the section called “Commit Queue” for a more detailed explanation.

`trace-id` Use the provided trace id as part of the log messages emitted while processing. If no trace id is given, NSO is going to generate and assign a trace id to the processing.

All commands in NSO can also have pipe commands. A useful pipe command for commit is **details**:

```
ncs% commit | details
```

This will give feedback on the steps performed in the commit.

When working with templates there is a pipe command **debug** which can be used to troubleshoot templates. To enable debugging on all templates use:

```
ncs% commit | debug template
```

When configuring using many templates the debug output can be overwhelming. For this reason there is an option to only get debug information for one template, in this example a template named l3vpn:

```
ncs% commit | debug template l3vpn
```

Device Actions

Actions for devices can be performed globally on the `/devices` path, and for individual devices on `/devices/device/name`. Many actions are also available on device-groups as well as device ranges.

add-capability

This action adds a capability to the list of capabilities. If `uri` is specified, then it is parsed as YANG capability string and `module`, `revision`, `feature` and `deviation` parameters are derived from the string. If `module` is specified, then the namespace is looked up in the list of loaded namespaces and capability string constructed automatically. If the `module` is specified and the attempt to look it up failed, then the action does nothing. If `module` is specified or can be derived from capability string, then the `module` is also added/replaced in the list of modules. This action is only intended to be used for pre-provisioning: it is not possible to override capabilities and modules provided by the NED implementation using this action.

apply-template

Take a named template and apply its configuration here.

If the `accept-empty-capabilities` parameter is included, the template is applied to devices even if the capability of the device is unknown.

This action will behave differently depending on if it is invoked with a transaction or not. When invoked with a transaction (such as via the CLI) it will apply the template to it and leave it to the user to commit or revert the resulting changes. If invoked without a transaction (for example when invoked via RESTCONF), the action will automatically create one and commit the resulting changes. An error will be returned and the transaction aborted if the template failed to apply on any of the devices.

check-sync

Check if the NSO copy of the device configuration is in sync with the actual device configuration, using device-specific mechanisms. This operation is usually cheap as it only compares a signature of the configuration from the device rather than comparing the entire configuration.

Depending on the device the signature is implemented as a transaction-id, timestamp, hash-sum or not at all. The capability must be supported by the corresponding NED. The output might say unsupported, and then the only way to perform this would be to do a full **compare-config** command.

Warning As some NEDs implements the signature as an hash-sum of the entire configuration, this operation might for some devices be just as expensive as performing a full **compare-config** command.

check-yang-modules
: Check if the device YANG modules loaded by NSO have revisions that are compatible with the ones reported by the device.

  This can indicate for example that the device has a YANG module of later revision than the corresponding NED.

clear-trace
: Clear all trace files for all active traces for all managed devices.

compare-config
: Retrieve the config from the device and compare to the NSO locally stored copy.

connect
: Set up a session to the unlocked device. This is not used in real operational scenarios. NSO automatically establishes connections on demand. However it is useful for test purposes when installing new NEDs, adding devices etc.

  When a device is southbound locked, all southbound communication is turned off. The `override-southbound-locked` flag overrides the southbound lock for connection attempts. Thus, this is a way to update the capabilities including revision information for a managed device although the device is southbound locked.

copy-capabilities
: This action copies the list of capabilities and the list of modules from another device or profile. When used on a device, this action is only intended to be used for pre-provisioning: it is not possible to override capabilities and modules provided by the NED implementation using this action.

  Note This action overwrites existing list of capabilities.

delete-config
: Delete the device configuration in NSO without executing the corresponding delete on the managed device.

disconnect
: Close all sessions to the device.

fetch-ssh-host-keys
: Retrieve the SSH host keys from all devices, or all devices in the given device group, and store them in each device's `ssh/host-key` list. Successfully retrieved new or updated keys are always committed by the action.

find-capabilities
: This action populates the list of capabilities based on the configured ned-id for the device, if possible. NSO will look up the package corresponding to the ned-id and add all the modules from these packages to the list of device capabilities and list of modules. It is the responsibility of the caller to verify that the automatically populated list of capabilities matches actual device's capabilities. The list of capabilities can then be fine-tuned using **add-capability** and **capability/remove** actions. Currently this approach will only work for CLI and generic devices. This action is only intended to be used for pre-provisioning: it is not possible to override capabilities and modules provided by the NED implementation using this action.

Note This action overwrites existing list of capabilities.

instantiate-from-other-device	Instantiate the configuration for the device as a copy of the configuration of some other already working device.
load-native-config	Load configuration data in native format into the transaction. This action is only applicable for devices with NETCONF, CLI and generic NEDs. The action can load the configuration data either from a file in the local filesystem or as a string through the northbound client. If loading XML the data must be a valid XML document, either with a single namespace or wrapped in a config node with the http://tail-f.com/ns/config/1.0 namespace. The `verbose` option can be used to show additional parse information reported by the NED. By default the behaviour is to merge the configuration that is applied. This can be changed by setting the `mode` option to replace. This will replace the entire device configuration. This action will behave differently depending on if it is invoked with a transaction or not. When invoked with a transaction (such as via the CLI) it will load the configuration into it and leave it to the user to commit or revert the resulting changes. If invoked without a transaction (for example when invoked via RESTCONF), the action will automatically create one and commit the resulting changes.
migrate	Change the NED identity and migrate all data. As a side-effect reads and commits the actual device configuration. The action reports what paths have been modified and the services affected by those changes. If the `verbose` option is used, all service instances are reported instead of just the service points. If the `dry-run` option is used, the action simply reports what it would do. If the `no-networking` option is used, no southbound traffic is generated towards the devices. Only the device configuration in CDB is used for the migration. If used, NSO can not know if the device is in sync. To determine this, the **compare-config** or the **sync-from** action must be used.
partial-sync-from	Synchronize parts of the devices' configuration by pulling from the network.
ping	ICMP ping the device.
scp-from	Secure copy file from the device. The `port` option specifies the port to connect to on the device. If this leaf is not configured, NSO will use the port for the management interface of the device. The `preserve` option preserves modification times, access times, and modes from the original file. This is not always supported by the device.
scp-to	Secure copy file to the device.

	The `port` option specifies the port to connect to on the device. If this leaf is not configured, NSO will use the port for the management interface of the device. The `preserve` option preserves modification times, access times, and modes from the original file. This is not always supported by the device.
sync-from	Synchronize the NSO copy of the device configuration by reading the actual device configuration. The change will be immediately committed to NSO. If the `dry-run` option is used, the action simply reports (in different formats) what it would do. The `verbose` option can be used to show additional parse information reported by the NED. **Note** If you have any services that has created configuration on the device the corresponding service might be out-of-sync. Use the commands **check-sync** and **re-deploy** to reconcile this.
sync-to	Synchronize the device configuration by pushing the NSO copy to the device. NSO pushes a minimal diff to the device. The diff is calculated by reading the configuration from the device and comparing with the configuration in NSO. If the `dry-run` option is used, the action simply reports (in different formats) what it would do.

Some of the operations above can't be performed while the device is being committed to (or waiting in the commit queue). This is to avoid getting inconsistent data when reading the configuration. The `wait-for-lock` option in these specifies a timeout to wait for a device lock to be placed in the commit queue. The lock will be automatically released once the action has been executed. If the `no-wait-for-lock` option is specified, the action will fail immediately for the device if the lock is taken for the device or if the device is placed in the commit queue. The `wait-for-lock` and the `no-wait-for-lock` options are device settings as well, they can be set as a device profile, device and global setting. The `no-wait-for-lock` option is set in the global settings by default. If neither `wait-for-lock` and the `no-wait-for-lock` options are provided together with the action, the device setting is used.

Service Actions

Service actions are performed on the service instance.

check-sync	Check if the service has been undermined, i.e., if the service was to be re-deployed, would it do anything. This action will invoke the FASTMAP code to create the change set that is compared to the existing data in CDB locally. If `outformat` is boolean, `true` is returned if the service is in sync, i.e., a re-deploy would do nothing. If outformat is `cli`, `xml` or `native`, the changes that the service would do to the network if re-deployed are returned. If configuration changes has been made out-of-band then **deep-check-sync** is needed to detect a out-of-sync condition.

The `deep` option is used to recursively **check-sync** stacked services. The `shallow` option only **check-sync** the topmost service.

deep-check-sync

Check if the service has been undermined on the device itself. The action **check-sync** compares the output of the service code to what is stored in CDB locally. This action retrieves the configuration from the devices touched by the service and compares the forward diff set of the service to the retrieved data. This is thus a fairly heavy weight operation. As opposed to the **check-sync** action that invokes the FASTMAP code, this action re-applies the forward diff-set. This is the same output you see when inspecting the **get-modifications** operational field in the service instance.

If the device is in sync with CDB, the output of this action is identical to the output of the cheaper **check-sync** action.

get-modifications

Returns the data the service modified, either in CLI curly bracket format, or NETCONF XML edit-config format. The modifications are shown as if the service instance was the only instance that modifies the data. This data is only available if the parameter `/services/global-settings/collect-forward-diff` is set to true.

If the parameter `reverse` is given the modifications needed to reverse the effect of the service is shown. The modifications are shown as if this service instance was the last service instance. This will be applied if the service is deleted. This data is always available.

The `deep` option is used to recursively **get-modifications** for stacked services. The `shallow` option only **get-modifications** for the topmost service.

re-deploy

Run the service code again, possibly writing the changes of the service to the network once again. There are several reasons for performing this operation such as:

- a **device sync-from** action has been performed in order to incorporate an out-of-band change.
- data referenced by the service has changed such as topology information, QoS policy definitions etc.

The `deep` option is used to recursively **re-deploy** stacked services. The `shallow` option only **re-deploy** the topmost service.

If the `dry-run` option is used, the action simply reports (in different formats) what it would do.

Use the option `reconcile` if the service should reconcile original data, i.e., take control of that data. This option acknowledges other services controlling the same data. All data which existed before the service was created will now be owned by the service. When the service is removed that data will also be removed. In technical terms the reference count will be decreased by one for everything which existed prior to the service. If manually configured data exists below in the configuration tree that data is kept unless the option `discard-non-service-config` is used.

> **Note** The action is idempotent. If no configuration diff exists then nothing needs to be done.

> **Note** The NSO general principle of minimum change applies.

reactive-re-deploy	This is a tailored **re-deploy** intended to be used in the reactive FASTMAP scenario. It differs from the ordinary **re-deploy** in that this action does not take any commit parameters. This action will **re-deploy** the services as an shallow depth **re-deploy**. It will be performed with the same user as the original commit. Also, the commit parameters will be identical to the latest commit involving this service. By default this action is asynchronous and returns nothing. Use the `sync` leaf to get synchronous behaviour and block until the service **re-deploy** transaction is committed. The `sync` leaf also means that the action will possibly return a commit result, such as commit queue id if any, or an error if the transaction failed.
touch	This action marks the service as changed. Executing the action **touch** followed by a commit is the same as executing the action **re-deploy shallow**. By using the action **touch** several re-deploys can be performed in the same transaction.
un-deploy	Undo the effects of the service instance but keep the service itself. The service can later be re-deployed. This is a means to deactivate a service, but keeping it in the system.

CHAPTER 13

Web User Interface

Overview

The NSO Web UI consists of a suit of web based applications. Each application has it's own distinct concern, for instance handle configuration, transaction handling, manage devices, manage services or monitor the system. The different applications can be accessed from the application hub, which is shown directly after authentication.

The Web UI is a mix of custom built applications and auto-rendering from the underlying device and service models. The latter gives the benefit that a Web UI is immediately updated when new devices or services are added to the system. So, say you add support for a new device vendor. Without any programming is the NSO Web UI capable of configuring those devices.

All modern web browsers are supported and no plug-ins are required. The interface is a pure JavaScript Client.

Using the Web UI

The Web UI is available on port 8080 on the NSO server. The port can be changed in the `ncs.conf` file. A NSO user must exist.

More help how to use the Web UI is present in the Web UI applications. The help is located in the user menu, which can be found to the right in the application header.

Transactions and Commit

Take special notice to the Commit Manager application, whenever a transaction has started, the active changes can be inspected and evaluated before they are commited and pushed to the network. The data is saved to NSO datastore and pushed to the network when a user presses "Commit".

Any network-wide change can be picked up as a rollback-file. That rollback can then be applied to undo whatever happened to the network.

CHAPTER 14

Network Simulator

Overview

The **ncs-netsim** program is a useful tool to simulate a network of devices to be managed by NSO. It makes it easy to test NSO packages towards simulated devices. All you need is the NSO NED packages for the devices that you need to simulate. The devices are simulated with the Tail-f ConfD product.

All the NSO examples use **ncs-netsim** to simulate the devices. A good way to learn how to use **ncs-netsim** is to study these.

Using Netsim

The **ncs-netsim** tool takes any number of NED packages as input. The user can specify the number of device instances per package (device type) and a string that is used as prefix for the name of the devices. The command takes the following parameters:

```
admin$ ncs-netsim --help
Usage ncs-netsim  [--dir <NetsimDir>]
            create-network <NcsPackage> <NumDevices> <Prefix> |
            create-device <NcsPackage> <DeviceName> |
            add-to-network <NcsPackage> <NumDevices> <Prefix> |
            add-device <NcsPackage> <DeviceName> |
            delete-network                     |
            [-a | --async]  start [devname]    |
            [-a | --async ] stop [devname]     |
            [-a | --async ] reset [devname]    |
            [-a | --async ] restart [devname]  |
            list                      |
            is-alive [devname]        |
            status [devname]          |
            whichdir                  |
            ncs-xml-init [devname]    |
            ncs-xml-init-remote <RemoteNodeName> [devname] |
            [--force-generic]         |
            packages                  |
            netconf-console devname [XpathFilter] |
```

```
        [-w | --window] [cli | cli-c | cli-i] devname
```

Assume you have prepared a NSO package for a device called `router`. (See the `examples.ncs/getting-started/developing-with-ncs/0-router-network` example). Also assume the package is in `./packages/router`. At this point you can create the simulated network by:

```
$ ncs-netsim create-network ./packages/router 3 device --dir ./netsim
```

This creates three devices; `device0`, `device1`, and `device2`. The simulated network is stored in the `./netsim` directory. The output structure is:

```
./netsim/device/
     device0/<ConfD files>, <log files>
     device1/
     ....
```

There is one separate directory for every ConfD simulating the devices.

The network can be started with

```
$ ncs-netsim start
```

You can add more devices to the network in a similiar way as it was created. E.g. if you created a network with some Juniper devices and want to add some Cisco IOS devices. Point to the NED you want to use (See `{NCS_DIR}/packages/neds/`) and run the command. Remember to start the new devices after they have been added to the network.

```
$ ncs-netsim add-to-network ${NCS_DIR}/packages/neds/cisco-ios 2 c-device --dir ./nets
```

To extract the device data from the simulated network to a file in XML format.

```
$ ncs-netsim ncs-xml-init > devices.xml
```

This data is usually used to load the simulated network into NSO. Putting the xml file in the **./ncs-cdb** folder will load it when NSO starts. If NSO is already started it can be reloaded while running.

```
$ ncs_load -l -m devices.xml
```

The generated device data creates devices of the same type as the device being simulated. This is true for netconf, cli and snmp devices. When simulating generic devices the simulated device will run as a netconf device.

Under very special circumstances one can choose to force running the simulation as a generic device with the option `--force-generic`

The simulated network device info can be shown with:

```
 $ ncs-netsim list
...
 name=device0 netconf=12022 snmp=11022 ipc=5010 cli=10022 dir=examples.ncs/getting-started/devel
with-ncs/0-router-network/netsim/device/device0
...
```

Here you can see the device name, the working directory and port number for different services to be accessed on the simulated device (NETCONF SSH, SNMP, IPC and direct access to the CLI).

You can reach the CLI of individual devices with:

```
$ ncs-netsim cli-c device0
```

The simulated devices actually provides three different styles of CLI:

- cli : J-Style
- cli-c : Cisco XR Style
- cli-i : Cisco IOS Style

Individual devices can be started and stopped with:

```
$ ncs-netsim start device0
$ ncs-netsim stop device0
```

You can check the status of the simulated network. Either a short version just to see if the device is running or a more verbose with all information.

```
$ ncs-netsim is-alive device0
$ ncs-netsim status device0
```

View which packages are used in the simulated network.

```
$ ncs-netsim packages
```

It is also possible to reset the network back to state of initialization.

```
$ ncs-netsim reset
```

When you are done, remove the network.

```
$ ncs-netsim delete-network
```

Using ConfD Tools with Netsim

The netsim tool includes a standard ConfD distribution and the ConfD C API library (libconfd) that the ConfD tools use. The library is built with default settings where the values for MAXDEPTH and MAXKEYLEN are 20 and 9, respectively. These values define the size of confd_hkeypath_t struct and this size is related to the size of data models in terms of depth and key lengths. Default values should be big enough even for very large and complex data models. But in some rare cases, one or both of these values might not be large enough for a given data model.

One might observe a limitation when the data models that are used by simulated devices exceed these limits. Then it would not be possible to use the ConfD tools that are provided with the netsim. To overcome this limitation, it is advised to use the corresponding NSO tools to perform desired tasks on devices.

NSO and ConfD tools and Python APIs are basically the same except for naming, the default IPC port and the MAXDEPTH and MAXKEYLEN values, where for NSO tools, the values are set to 60 and 18, respectively. Thus, the advised solution is to use the NSO tools and NSO Python API with netsim.

e.g. Instead of using below cmd

```
$ CONFD_IPC_PORT=5010 ${NCS_DIR}/netsim/confd/bin/confd_load -m -l *.xml
```

One may use:

```
$ NCS_IPC_PORT=5010 ncs_load -m -l *.xml
```

Learn more

The README file in `examples.ncs/getting-started/developing-with-ncs/0-router-network` gives a good introduction on how to use **ncs-netsim**.