



NSO 5.7 Development Guide

Release: NSO 5.7.1

Published: May 17, 2010

Last Modified: January 26, 2022

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

THE SPECIFICATIONS AND INFORMATION REGARDING THE PRODUCTS IN THIS MANUAL ARE SUBJECT TO CHANGE WITHOUT NOTICE. ALL STATEMENTS, INFORMATION, AND RECOMMENDATIONS IN THIS MANUAL ARE BELIEVED TO BE ACCURATE BUT ARE PRESENTED WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. USERS MUST TAKE FULL RESPONSIBILITY FOR THEIR APPLICATION OF ANY PRODUCTS.

THE SOFTWARE LICENSE AND LIMITED WARRANTY FOR THE ACCOMPANYING PRODUCT ARE SET FORTH IN THE INFORMATION PACKET THAT SHIPPED WITH THE PRODUCT AND ARE INCORPORATED HEREIN BY THIS REFERENCE. IF YOU ARE UNABLE TO LOCATE THE SOFTWARE LICENSE OR LIMITED WARRANTY, CONTACT YOUR CISCO REPRESENTATIVE FOR A COPY.

The Cisco implementation of TCP header compression is an adaptation of a program developed by the University of California, Berkeley (UCB) as part of UCB's public domain version of the UNIX operating system. All rights reserved. Copyright © 1981, Regents of the University of California.

NOTWITHSTANDING ANY OTHER WARRANTY HEREIN, ALL DOCUMENT FILES AND SOFTWARE OF THESE SUPPLIERS ARE PROVIDED "AS IS" WITH ALL FAULTS. CISCO AND THE ABOVE-NAMED SUPPLIERS DISCLAIM ALL WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THOSE OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OR ARISING FROM A COURSE OF DEALING, USAGE, OR TRADE PRACTICE.

IN NO EVENT SHALL CISCO OR ITS SUPPLIERS BE LIABLE FOR ANY INDIRECT, SPECIAL, CONSEQUENTIAL, OR INCIDENTAL DAMAGES, INCLUDING, WITHOUT LIMITATION, LOST PROFITS OR LOSS OR DAMAGE TO DATA ARISING OUT OF THE USE OR INABILITY TO USE THIS MANUAL, EVEN IF CISCO OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Any Internet Protocol (IP) addresses and phone numbers used in this document are not intended to be actual addresses and phone numbers. Any examples, command display output, network topology diagrams, and other figures included in the document are shown for illustrative purposes only. Any use of actual IP addresses or phone numbers in illustrative content is unintentional and coincidental.

Cisco and the Cisco logo are trademarks or registered trademarks of Cisco and/or its affiliates in the U.S. and other countries. To view a list of Cisco trademarks, go to this URL: <https://www.cisco.com/go/trademarks>. Third-party trademarks mentioned are the property of their respective owners. The use of the word partner does not imply a partnership relationship between Cisco and any other company. (1110R)

This product includes software developed by the NetBSD Foundation, Inc. and its contributors.

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com).

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit <http://www.openssl.org/>.

This product includes software written by Tim Hudson (tjh@cryptsoft.com).

U.S. Pat. No. 8,533,303 and 8,913,519

Copyright © 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022 Cisco Systems, Inc. All rights reserved.



CONTENTS

Preface **xv**

CHAPTER 1

The Configuration Database and YANG	1
Key Features of the CDB	1
Compilation and Loading of YANG Modules	2
Showcase: Extending the CDB with Packages	2
Data Modeling Basics	4
Showcase: Building and Testing a Model	6
Initialization Files	8

CHAPTER 2

Basic Automation with Python	11
Setup	11
Transactions	12
Read and Write Values	12
Lists	13
Showcase: Configuring DNS with Python	14
The Magic Behind the API	16

CHAPTER 3

Creating a Service	19
Why services?	19
The Service Package	20
Service Parameters	22
Showcase: A Simple DNS Configuration Service	23
Service Templates	27
Showcase: DNS Configuration Service with Templates	29

CHAPTER 4

Applications in NSO	33
NSO Architecture	33
Callbacks as Extension Mechanism	34

Actions	36
Showcase: Implementing Device Count Action	37
Running Application Code	41
<hr/>	
CHAPTER 5	The NSO Java VM 43
Overview	43
YANG model	45
Java Packages and the Class Loader	45
The NED component type	47
The callback component type	47
The application component type	47
The Resource Manager	48
The Alarm centrals	50
Embedding the NSO Java VM	50
JMX interface	51
Logging	54
The NSO Java VM timeouts	54
Debugging Startup	54
<hr/>	
CHAPTER 6	The NSO Python VM 57
Introduction	57
YANG model	57
Structure of the User provided code	59
The application component	60
The upgrade component	61
Debugging of Python packages	62
Using non-standard Python	63
Configure NSO to use a custom start command (recommended)	63
Changing the path to <i>python3</i> or <i>python</i>	64
Updating the default start command (not recommended)	64
<hr/>	
CHAPTER 7	Embedded Erlang applications 65
Introduction	65
Erlang API	65
Application configuration	66
Example	67

CHAPTER 8

The YANG Data Modeling Language	69
The YANG Data Modeling Language	69
YANG in NSO	69
YANG Introduction	70
Modules and Submodules	70
Data Modeling Basics	70
Leaf Nodes	70
Leaf-list Nodes	71
Container Nodes	71
List Nodes	71
Example Module	72
State Data	73
Built-in Types	73
Derived Types (typedef)	74
Reusable Node Groups (grouping)	75
Choices	75
Extending Data Models (augment)	76
RPC Definitions	77
Notification Definitions	77
Working With YANG Modules	78
Integrity Constraints	79
The when statement	81
Using the Tail-f Extensions with YANG	81
Using a YANG annotation file	82
Custom Help Texts and Error Messages	83
Custom Help Texts	83
Custom Help Text in a Typedef	84
Custom Error Messages	84
An Example: Modeling a List of Interfaces	84
Modeling Relationships	87
Ensuring Uniqueness	89
Default Values	89
The Final Interface YANG model	90
More on leafrefs	92
Using Multiple Namespaces	94

Module Names, Namespaces and Revisions	95
Hash Values and the id-value Statement	96
NSO caveats	96
The union type and value conversion	96
User defined types	97

CHAPTER 9

Using CDB	99
Introduction	99
The NSO Data Model	100
Addressing Data Using Keypaths	102
Subscriptions	103
Sessions	104
Loading Initial Data Into CDB	105
Operational Data in CDB	106
Subscriptions	106
Example	107
Operational Data	114
Automatic Schema Upgrades and Downgrades	118
Using Initialization Files for Upgrade	120
Writing an Upgrade Package Component	123

CHAPTER 10

Java API Overview	135
Introduction	135
MAAPI	138
CDB API	140
DP API	144
Transaction and Data Callbacks	145
Service and Action Callbacks	153
Validation Callbacks	154
Transforms	155
Hooks	155
NED API	156
NAVU API	156
ALARM API	164
NOTIF API	166

HA API	168
Java API Conf Package	168
Namespace classes and the loaded schema	171
CHAPTER 11	
Python API Overview	173
Introduction	173
Python API overview	174
Python scripting	175
High-level MAAP API	175
Maagic API	176
Namespaces	177
Reading data	177
Writing data	178
Deleting data	178
Containers	178
Presence containers	178
Choices	178
Lists and List elements	179
Unions	179
Enumeration	179
Leafref	180
Identityref	180
Instance-identifier	180
Leaf-list	180
Binary	181
Bits	181
Empty leaf	182
Maagic examples	182
Action requests	182
PlanComponent	185
Python packages	186
Action handler	186
Service handler	187
Low-level APIs	189
Low-level MAAP API	190



Low-level CDB API 191

CHAPTER 12

NSO Packages	193
Package Overview	193
An Example Package	195
The package-meta-data.xml file	195
Components	198
Creating Packages	201
Creating a NETCONF NED Package	201
Creating an SNMP NED Package	202
Creating a CLI NED Package or a Generic NED Package	202
Creating a Service Package or a Data Provider Package	202

CHAPTER 13

Package Development	203
Developing a Service Package	203
ncs-setup	205
The netsim Part of a NED Package	206
Plug-and-play scripting	208
Creating a Service Package	208
Java Service Implementation	209
Developing our First Service Application	209
Tracing Within the NSO Service Manager	212
Controlling error messages info level from Java	215
Loading Packages	216
Debugging the Service and Using Eclipse IDE	216
Running the NSO Java VM Standalone	218
Using Eclipse to Debug the Package Java Code	219
Remote Connecting with Eclipse to the NSO Java VM	220
Working with ncs-project	220
Create a new project	220
Project setup	221
Export	223
NSO Project man pages	224
The project-meta-data.xml file	225

CHAPTER 14

Developing NSO Services	229
--------------------------------	------------

Introduction	229
Definitions	230
The Fundamentals	231
Mapping	231
FASTMAP and Transactions	232
Auto-rendering from the Service Model	232
Writing the Service Model	232
Finding the Mapping	232
Mapping Iterations	233
Strategies to Implement the Mapping	234
Creating an NSO Service Application	234
Mapping using Service Templates	235
Preparation	235
Defining the Service Model	236
Defining the Template	239
Mapping using Java	245
Overview	245
Setting up the environment	245
Creating a service package	247
The Service Model	248
Managing the NSO Java VM	249
A first look at Java Development	250
Using Eclipse	251
Writing the service code	256
Mapping using Java combined with Templates	260
Overview	260
The VLAN Feature Template	261
The VLAN Java Logic	262
Steps to Build a Java and Template Solution	263
Service Mapping: Putting Things Together	263
Auxiliary Service Data	263
Topology	264
Creating a Multi-Vendor Service	266
Defining the Mapping	270
FASTMAP Description	273

**CHAPTER 15**

Reactive FASTMAP	275
Progress reporting using plan-data	278
Service Progress Monitoring	320
Performance Considerations	327
Services that involve virtual devices, NFV	328
Starting a virtual machine	328
Stopping a virtual machine	329
How to handle Licenses	330
Advanced Mapping Techniques	330
Create Methods	330
Persistent FASTMAP Properties	332
Pre and post hooks	333
Stacked Services and Shared Structures	336
FASTMAP pre-lock create option	336
Service Caveats	337
Service Discovery	339
Discovery basics	339
Reconciliation caveats	344
Reconciling in bulk	344
Partial sync	345
Partial sync-from	345
Templates	347
Introduction	347
Config templates	347
Create a Config-template from a Device-template	348
Create a Config-template from a device configuration	349
Basic principles	350
Values in a template	351
Processing instructions	352
XPath Context in config-templates	354
Conditional Statements	355
Loop Statements	356
Capabilities and Namespaces	357
Template tag operations	357

	Operations on ordered-by user lists and leaf-lists	358
	Debugging templates	360
	Service Templates	362
	Macros in Service Templates	367
	Pre- and Post-Modification Templates	369
	Templates applied from an API	370
	Feature Template	370
	Passing deep structures from an API	371
	Service and API Templates in multi-NED environment	373
<hr/> CHAPTER 16	Developing Alarm Applications	375
	Introduction	375
	Using the Alarms Sink	376
	Using the Alarms Source	378
	Extending the alarm manager, adding user defined alarm types and fields	379
	Mapping alarms to objects	381
<hr/> CHAPTER 17	SNMP Notification Receiver	383
	Introduction	383
	Configuring NSO to Receive SNMP Notifications	384
	Built-in Filters	384
	Notification Handlers	385
<hr/> CHAPTER 18	The web server	389
	Introduction	389
	Web server capabilities	389
	CGI support	390
	Storing TLS data in database	391
<hr/> CHAPTER 19	Kicker	395
	Introduction	395
	Kicker action invocation	395
	Data Kicker Concepts	396
	Generalized Monitors	396
	Kicker Constraints/Filters	398
	Variable Bindings	399



CHAPTER 20**Scheduler** **411**

Introduction	411
Scheduling Periodic Work	411
Schedule Expression	412
Scheduling Non-recurring Work	412
Scheduling in a HA Cluster	412
Troubleshooting	413
History log	413
XPath log	413
Devel Log	413
Suspending the Scheduler	413

CHAPTER 21**Progress Trace** **415**

Introduction	415
Configuring Progress Trace	416
Unhide Progress Trace	416
Log to File	416
Log as Operational Data	416
Log as Notification Events	417
Verbosity	417
Using Filters	417

Report Progress Events from User Code **417**

CHAPTER 22**Nano Services for Staged Provisioning **419****

Basic Concepts **420**

 Plan Outline **420**

 Per-State Configuration **421**

 Link Plan Outline to Service **422**

 Service Instantiation **424**

 Benefits and Use Cases **425**

 Backtracking and Staged Delete **427**

 Managing Side Effects **429**

 Multiple and Dynamic Plan Components **431**

 Netsim Router Provisioning Example **434**

 Zombie Services **436**

 Using Notifications to Track the Plan and its Status **437**

 The `plan-state-change` Notification **437**

 The `service-commit-queue-event` Notification **438**

 Examples of `service-state-changes` Stream Subscriptions **438**

 The `trace-id` in the Notification **440**

 Developing and Updating a Nano Service **440**

 Adding Components **441**

 Removing Components **441**

 Replacing Components **441**

 Adding and Removing States **441**

 Modifying States **442**

 Implementation Reference **442**

 Back-Tracking **442**

 Behavior Tree **445**

 Nano Service Pre-Condition **447**

 Nano Service Opaque and Component Properties **449**

 Nano Service Callbacks **450**

 Generic Service Callbacks **452**

 Nano Service Pre/Post Modifications **452**

 Forced Commits **453**

 Plan Location **453**



CHAPTER 23

Nano Services and Commit Queue	454
Graceful Link Migration Example	455
<hr/>	
Encryption Keys	463
Introduction	463
Reading encryption keys using an external command	463

CHAPTER 24

External Logging	465
Introduction	465
Enabling external log processing	465
Processing logs using an external command	466

Preface

Cisco Network Services Orchestrator (NSO) is a versatile tool. Its design acknowledges that it is impossible to support every specific use case out of the box in today's diverse networks. Instead, NSO provides the tooling and support systems that enable fast and cost-effective development of custom, bespoke solutions.

As the product name suggests, a lot revolves around services in NSO. When implemented as services, network applications can offload many common tasks to NSO and focus on use-case-specific logic, allowing you to manage configuration in an intelligent way.

The following guide primarily targets software developers, system integrators, and network automation engineers who need to extend NSO with service packages or add other custom functionality. The content leads you through the process of developing applications in NSO and shows you how to use various NSO features to simplify network automation.

We recommend readers have at least some understanding of the Python or Java programming language. The required level depends on the complexity of the actual use case. You can implement many automation scenarios by simply modifying the provided examples. However, prior knowledge of NSO basics, as described in the NSO User Guide, is necessary for the best experience.



CHAPTER 1

The Configuration Database and YANG

Cisco NSO is a network automation platform that supports a variety of uses. This can be as simple as configuration of a standard-format hostname, which can be implemented in minutes. Or it could be an advanced MPLS VPN with custom traffic-engineered paths in a Service Provider network, which might take weeks to design and code.

Regardless of complexity, any network automation solution must keep track of two things: intent and network state. The Configuration Database (CDB) built into NSO was designed for this exact purpose. Firstly, the CDB will store the intent, which describes what you want from the network. Traditionally we call this intent a network service, since this is what the network ultimately provides to its users. Secondly, the CDB also stores a copy of the configuration of the managed devices, that is, the network state. Knowledge of the network state is essential in order to correctly provision new services. It also enables faster diagnosis of problems and is required for advanced functionality, such as self-healing.

This chapter will describe the main features of the CDB and explain how NSO stores data there. To help you better understand the structure of the CDB, you will learn how to add your own data to it.

- [Key Features of the CDB, page 1](#)
- [Compilation and Loading of YANG Modules, page 2](#)
- [Showcase: Extending the CDB with Packages, page 2](#)
- [Data Modeling Basics, page 4](#)
- [Showcase: Building and Testing a Model, page 6](#)
- [Initialization Files, page 8](#)

Key Features of the CDB

The CDB is a dedicated built-in storage for data in NSO. It was built from the ground up to efficiently store and access network configuration data, such as device configurations, service parameters, and even configuration for NSO itself. Unlike traditional SQL databases that store data as rows in a table, the CDB is a hierarchical database, with a structure resembling a tree. You could think of it as somewhat like a big XML document that can store all kinds of data.

There are a number of other features that make the CDB an excellent choice for a configuration store:

- Fast lightweight database access through a well-defined API.
- Subscription (“push”) mechanism for change notification.
- Transaction support for ensuring data consistency.
- Rich and extensible schema based on YANG.

- Built-in support for schema and associated data upgrade.
- Close integration with NSO for low-maintenance operation.

To speed up operations, CDB keeps a copy of configuration data in RAM, in addition to persisting it to disk using journal files. However, this means the amount of RAM needed is proportional to the number of managed devices and services. When NSO is used to manage a large network the amount of needed RAM can be quite large. The CDB also stores transient operational data, such as alarms and traffic statistics. By default, this operational data is only kept in RAM and is reset during restarts, however, the CDB can be instructed to persist it if required.


Important

For reliable storage of the configuration on disk, the CDB requires that the file system correctly implements the standard primitives for file synchronization and truncation. For this reason (as well as for performance), NFS or other network file systems are unsuitable for use with the CDB - they may be acceptable for development, but using them in production is *unsupported* and *strongly discouraged*.

The automatic schema update feature is useful not only when performing an actual upgrade of NSO itself, it also simplifies the development process. It allows individual developers to add and delete items in the configuration independently.

Additionally, the schema for data in the CDB is defined with a standard modeling language called YANG. YANG (RFC 7950, <https://tools.ietf.org/html/rfc7950>) describes constraints on the data and allows the CDB to store values more efficiently.

Compilation and Loading of YANG Modules

All of the data stored in the CDB follows the data model provided by various YANG modules. Each module usually comes as one or more files with a .yang extension and declares a part of the overall model.

NSO provides a base set of YANG modules out of the box. They are located in \$NCS_DIR/src/ncs/yang if you wish to inspect them. These modules are required for proper system operation.

All other YANG modules are provided by packages and extend the base NSO data model. For example, each Network Element Driver (NED) package adds the required nodes to store the configuration for that particular type of device. In the same way, you can store your custom data in the CDB by providing a package with your own YANG module.

However, the CDB can't use the YANG files directly. The bundled compiler, **ncsc**, must first transform a YANG module into a final schema (.fxs) file. The reason is that internally and in the programming APIs NSO refers to YANG nodes with integer values instead of names. This conserves space and allows for more efficient operations, such as switch statements in the application code. The .fxs file contains this mapping and needs to be recreated if any part of the YANG model changes. The compilation process is usually started from the package Makefile by the **make** command.

Showcase: Extending the CDB with Packages

Prerequisites:

- No previous NSO or netsim processes are running. Use the **ncs --stop** and **ncs-netsim stop** commands to stop them if necessary.
- NSO local install with a fresh runtime directory has been created by the **ncs-setup --dest ~/nso-lab-rundir** or similar command.

- The environment variable NSO_RUNDIR points to this runtime directory, such as set by the **export NSO_RUNDIR=~/nso-lab-rundir** command. It enables the below commands to work as-is, without additional substitution needed.

Step 1: Create a package

The easiest way to add your own data fields to the CDB is by creating a service package. The package includes a YANG file for the service-specific data, which you can customize. You can create the initial package by simply invoking the **ncs-make-package** command. This command also sets up a Makefile with the code for compiling the YANG model.

Use the following command to create a new package:

```
$ ncs-make-package --service-skeleton python --build \
--dest $NSO_RUNDIR/packages/my-data-entries my-data-entries
mkdir -p ..load-dir
mkdir -p java/src/
/nso/bin/ncsc `ls my-data-entries-ann.yang > /dev/null 2>&1 && echo "-a my-data-entries-ann
-c -o ..load-dir/my-data-entries.fxs yang/my-data-entries.yang
$
```

The command line switches instruct the command to compile the YANG file and place the package in the right location.

Step 2: Add package to NSO

Now start the NSO process if it is not running already and connect to the CLI:

```
$ cd $NSO_RUNDIR ; ncs ; ncs_cli -Cu admin
admin connected from 127.0.0.1 using console on nso
admin@ncs#
```

Next, instruct NSO to load the newly created package:

```
admin@ncs# packages reload
>>> System upgrade is starting.
>>> Sessions in configure mode must exit to operational mode.
>>> No configuration changes can be performed until upgrade has completed.
>>> System upgrade has completed successfully.
reload-result {
    package my-data-entries
    result true
}
```

Once the package loading process completes, you can verify the data model from your package was incorporated into NSO. Use the **show** command, which now supports an additional parameter:

```
admin@ncs# show my-data-entries
% No entries found.
admin@ncs#
```

This command tells you that NSO knows about the extended data model but there is no actual data configured for it yet.

Step 3: Set data

More interestingly, you are now able to add custom entries to the configuration. First enter the CLI configuration mode:

Step 4: Inspect the YANG module

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

Then add an arbitrary entry under my-data-entries:

```
admin@ncs(config)# my-data-entries "entry number 1"
admin@ncs(config-my-data-entries-entry number 1)#
```

What is more, you can also set a dummy IP address:

```
admin@ncs(config-my-data-entries-entry number 1)# dummy 0.0.0.0
admin@ncs(config-my-data-entries-entry number 1)#
```

However, if you try to use something different from dummy, you will get an error. Likewise, if you try to assign dummy a value that is not an IP address. How did NSO learn about this dummy value?

If you assumed from the YANG file, you are correct. YANG files provide the schema for the CDB and that dummy value comes from the YANG model in your package. Let's take a closer look.

Step 4: Inspect the YANG module

Exit the configuration mode and discard the changes by typing **abort**:

```
admin@ncs(config-my-data-entries-entry number 1)# abort
admin@ncs#
```

Open the YANG file in an editor or list its contents from the CLI with the following command:

```
admin@ncs# file show packages/my-data-entries/src/yang/my-data-entries.yang
module my-data-entries {
< ... output omitted ... >
  list my-data-entries {
    < ... output omitted ... >
    leaf dummy {
      type inet:ipv4-address;
    }
  }
}
```

At the start of the output you can see the module **my-data-entries**, which contains your data model. By default, the **ncs-make-package** gives it the same name as the package. You can check that this module is indeed loaded:

```
admin@ncs# show ncs-state loaded-data-models data-model my-data-entries
```

NAME	REVISION	NAMESPACE	PREFIX	EXPORTED TO ALL	EXPORTED TO
<hr/>					
my-data-entries	-	http://com/example/mydataentries	my-data-entries	X	-
<hr/>					

admin@ncs#

The **list my-data-entries** statement, located a bit further down in the YANG file, allowed you to add custom entries before. And near the end of the output you can find the **leaf dummy** definition, with IPv4 as the type. This is the source of information that enables NSO to enforce a valid IP address as the value.

Data Modeling Basics

NSO uses YANG to structure and enforce constraints on data that it stores in the CDB. YANG was designed to be extensible and handle all kinds of data modeling, which resulted in a number of language

features that help achieve this goal. However, there are only four fundamental elements (node types) for describing data:

- leaf nodes
- leaf-list nodes
- container nodes
- list nodes

You can then combine these elements into a complex, tree-like structure, which is why we refer to individual elements as nodes (of the data tree). In general, YANG separates nodes into those that hold data (leaf, leaf-list) and those that hold other nodes (container, list).

A *leaf* contains simple data such as an integer or a string. It has one value of a particular type, and no child nodes. For example:

```
leaf host-name {
    type string;
    description "Hostname for this system";
}
```

This code describes the structure that can hold a value of a hostname (of some device). A leaf node is used because hostname only has a single value, that is, device has one (canonical) hostname. In the NSO CLI, you set a value of a leaf simply as:

```
admin@ncs(config)# host-name "server-NY-01"
```

A *leaf-list* is a sequence of leaf nodes of the same type. It can hold multiple values, very much like an array. For example:

```
leaf-list domains {
    type string;
    description "My favourite internet domains";
}
```

This code describes a data structure that can hold many values, such as a number of domain names. In the CLI, you can assign multiple values to a leaf-list with the help of square bracket syntax:

```
admin@ncs(config)# domains [ cisco.com tail-f.com ]
```

Leaf and leaf-list describe nodes that hold simple values. As a model keeps expanding, having all data nodes on the same (top) level can quickly become unwieldy. A *container* node is used to group related nodes into a subtree. It has only child nodes and no value. Container may contain any number of child nodes of any type (including leafs, lists, containers, and leaf-lists). For example:

```
container server-admin {
    description "Administrator contact for this system";
    leaf name {
        type string;
    }
}
```

This code defines a concept of a server administrator. In the CLI, you first select the container before you access the child nodes:

```
admin@ncs(config)# server-admin name "Ingrid"
```

Similarly, a *list* defines a collection of container-like list entries that share the same structure. Each entry is like a record or a row in a table. It is uniquely identified by the value of its key leaf (or leaves). A list

definition may contain any number of child nodes of any type (leafs, containers, other lists, and so on). For example:

```
list user-info {
    description "Information about team members";
    key "name";
    leaf name {
        type string;
    }
    leaf expertise {
        type string;
    }
}
```

This code defines a list of users (of which there can be many), where each user is uniquely identified by their name. In the CLI, lists take an additional parameter, the key value, in order to select a single entry:

```
admin@ncs(config)# user-info "Ingrid"
```

To set a value of a particular list entry, first specify the entry, then the child node, like so:

```
admin@ncs(config)# user-info "Ingrid" expertise "Linux"
```

Combining just these four fundamental YANG node types, you can build a very complex model that describes your data. As an example, the model for configuration of a Cisco IOS-based network device, with its myriad features, is created with YANG. However, it makes sense to start with some simple models, to learn what kind of data they can represent and how to alter that data with the CLI.

Showcase: Building and Testing a Model

Prerequisites:

- No previous NSO or netsim processes are running. Use the **ncs --stop** and **ncs-netsim stop** commands to stop them if necessary.
- NSO local install with a fresh runtime directory has been created by the **ncs-setup --dest ~/nso-lab-rundir** or similar command.
- The environment variable **NSO_RUNDIR** points to this runtime directory, such as set by the **export NSO_RUNDIR=~/nso-lab-rundir** command. It enables the below commands to work as-is, without additional substitution needed.

Step 1: Create a model skeleton

You can add custom data models to NSO by using packages. So, you will build a package to hold the YANG module that represents your model. Use the following command to create a package (if you are building on top of the previous showcase, the package may already exist and will be updated):

```
$ ncs-make-package --service-skeleton python \
--dest $NSO_RUNDIR/packages/my-data-entries my-data-entries
$
```

Change the working directory to the directory of your package:

```
$ cd $NSO_RUNDIR/packages/my-data-entries
```

You will place the YANG model into the `src/yang/my-test-model.yang` file. In a text editor, create a new file and add the following text at the start:

```
module my-test-model {
```

```
namespace "http://example.tail-f.com/my-test-model";
prefix "t";
```

The first line defines a new module and gives it a name. In addition, there are two more statements required: the namespace and prefix. Their purpose is to help avoid name collisions.

Step 2: Fill out the model

Add a statement for each of the four fundamental YANG node types (leaf, leaf-list, container, list) to the my-test-model.yang model.

```
leaf host-name {
    type string;
    description "Hostname for this system";
}
leaf-list domains {
    type string;
    description "My favourite internet domains";
}
container server-admin {
    description "Administrator contact for this system";
    leaf name {
        type string;
    }
}
list user-info {
    description "Information about team members";
    key "name";
    leaf name {
        type string;
    }
    leaf expertise {
        type string;
    }
}
```

Also add the closing bracket for module at the end:

```
}
```

Remember to finally save the file as my-test-model.yang in the src/yang/ directory of your package. It is a best practice for the name of the file to match the name of the module.

Step 3: Compile and load the model

Having completed the model, you must compile it into an appropriate (.fxs) format. From the text editor first return to the shell and then run the **make** command in the src/ subdirectory of your package:

```
$ make -C src/
make: Entering directory 'nso-run/packages/my-data-entries/src'
/nso/bin/ncsc `ls my-test-model-ann.yang > /dev/null 2>&1 && echo "-a my-test-model-ann.yang
               -c -o ../load-dir/my-test-model.fxs yang/my-test-model.yang
make: Leaving directory 'nso-run/packages/my-data-entries/src'
$
```

The compiler will report if there are errors in your YANG file, and you must fix them before continuing.

Next, start the NSO process and connect to the CLI:

```
$ cd $NSO_RUNDIR && ncs && ncs_cli -C -u admin
```

Step 4: Test the model

```
admin connected from 127.0.0.1 using console on nso
admin@ncs#
```

Finally, instruct NSO to reload the packages:

```
admin@ncs# packages reload

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

reload-result {
    package my-data-entries
    result true
}
admin@ncs#
```

Step 4: Test the model

Enter the configuration mode by using the **config** command and test out how to set values for the data nodes you have defined in the YANG model:

- host-name leaf
- domains leaf-list
- server-admin container
- user-info list

Use the **?** and **TAB** keys to see the possible completions.

Now feel free to go back and experiment with the YANG file to see how your changes affect the data model. Just remember to rebuild and reload the package after you make any changes.

Initialization Files

Adding a new YANG module to the CDB enables it to store additional data, however there is nothing in the CDB for this module yet. While you can add configuration with the CLI, for example, there are situations where it makes sense to start with some initial data in the CDB already. This is especially true when a new instance starts for the first time and the CDB is completely empty.

In such cases you can bootstrap the CDB data with XML files. There are various uses for this feature. For example, you can implement some default “factory settings” for your module or you might want to pre-load data when creating a new instance for testing.

In particular, some of the provided examples use the CDB init files mechanism to save you typing out all of the initial configuration commands by hand. They do so by creating a file with the configuration encoded in the XML format.

When starting empty, the CDB will try to initialize the database from all XML files found in the directories specified by the `init-path` and `db-dir` settings in `ncs.conf` (please see `ncs.conf(5)` in *NSO 5.7 Manual Pages* for exact details). The loading process scans the files with the `.xml` suffix and adds all the data in a single transaction. In other words, there is no specified order in which the files are processed. This happens early during start up, during the so-called *start phase 1*, described in the section called “Starting NSO” in *NSO 5.7 Administration Guide*.

The content of the init file does not need to be a complete instance document but can specify just a part of the overall data, very much like the contents of the NETCONF `edit-config` operation. However, the end result of applying all the files must still be valid according to the model.

It is a good practice to wrap the data inside a `config` element, as it gives you the option to have multiple top-level data elements in a single file while it remains a valid XML document. Otherwise, you would have to use separate files for each of them. The following example uses the `config` element to fit all the elements into a single file.

Example 1. A sample CDB init file `my-test-data.xml`

```
<config xmlns="http://tail-f.com/ns/config/1.0">
  <host-name xmlns="http://example.tail-f.com/my-test-model">server-NY-01</host-name>

  <server-admin xmlns="http://example.tail-f.com/my-test-model">
    <name>Ingrid</name>
  </server-admin>
</config>
```

There are many ways to generate the XML data. A common approach is to dump existing data with the `ncs_load` utility or the `display xml` filter in the CLI. In fact, all of the data in the CDB can be represented (or exported, if you will) in XML. This is no coincidence. XML was the main format for encoding data with NETCONF when YANG was originally created and you can trace the origin of some YANG features back to XML.

Example 2. Creating init XML file with the `ncs_load` command

```
$ ncs_load -F p -p /domains > cdb-init.xml
$ cat cdb-init.xml
<config xmlns="http://tail-f.com/ns/config/1.0">
  <domains xmlns="http://example.tail-f.com/my-test-model">cisco.com</domains>
  <domains xmlns="http://example.tail-f.com/my-test-model">tail-f.com</domains>
</config>
$
```




CHAPTER 2

Basic Automation with Python

You can manipulate data in the CDB with the help of XML files or the UI, however, these approaches are not well suited for programmatic access. NSO includes libraries for multiple programming languages, providing a simpler way for scripts and programs to interact with it. The Python Application Programming Interface (API) is likely the easiest to use.

This chapter will show you how to read and write data using the Python programming language. With this approach, you will learn how to do basic network automation in just a few lines of code.

- [Setup, page 11](#)
- [Transactions, page 12](#)
- [Read and Write Values, page 12](#)
- [Lists, page 13](#)
- [Showcase: Configuring DNS with Python, page 14](#)
- [The Magic Behind the API, page 16](#)

Setup

The environment setup that happens during the sourcing of the `ncsrc` file also configures the `PYTHONPATH` environment variable. It allows the Python interpreter to find the NSO modules, which are packaged with the product. This approach also works with Python virtual environments and does not require installing any packages.

Since the `ncsrc` file takes care of setting everything up, you can directly start the Python interactive shell and import the main `ncs` module. This module is a wrapper around a low-level `C_ncs` module that you may also need to reference occasionally. Documentation for both of the modules is available through the built-in `help()` function or separately in the HTML format.

If the `import ncs` statement fails, please verify that you are using a supported Python version and that you have sourced the `ncsrc` beforehand.

Generally, you can run the code from the Python interactive shell but we recommend against it. The code uses nested blocks, which are hard to edit and input interactively. Instead, we recommend you save the code to a file, such as `script.py`, which you can then easily run and rerun with the `python3 script.py` command. If you would still like to interactively inspect or alter the values during the execution, you can use the `import pdb; pdb.set_trace()` statements at the location of interest.

Transactions

With NSO, data reads and writes normally happen inside a transaction. Transactions ensure consistency and avoid race conditions, where simultaneous access by multiple clients could result in data corruption, such as reading half-written data. To avoid this issue, NSO requires you to first start a transaction with a call to `ncs.maapi.single_read_trans()` or `ncs.maapi.single_write_trans()`, depending on whether you want to only read data or read and write data. Both of them require you to provide the following two parameters:

user	Username (string) of the user you wish to connect as
context	Method of access (string), allowing NSO to distinguish between CLI, web UI, and other types of access, such as Python scripts

These parameters specify security-related information that is used for auditing, access authorization, and so on. Please refer to Chapter 9, *The AAA infrastructure in NSO 5.7 Administration Guide* for more details.

As transactions use up resources, it is important to clean up after you are done using them. Using a Python `with` code block will ensure that clean up is automatically performed after a transaction goes out of scope. For example:

```
with ncs.maapi.single_read_trans('admin', 'python') as t:  
    ...
```

In this case, the variable `t` stores the reference to a newly started transaction. Before you can actually access the data, you also need a reference to the root element in the data tree for this transaction. That is, the top element, under which all of the data is located. The `ncs.maagic.get_root()` function, with transaction `t` as a parameter, achieves this goal.

Read and Write Values

Once you have the reference to the root element, say in a variable named `root`, navigating the data model becomes straightforward. Accessing a property on `root` selects a child data node with the same name as the property. For example, `root.nacm` gives you access to the `nacm` container, used to define fine-grained access control. Since `nacm` is itself a container node, you can select one of its children using the same approach. So, the code `root.nacm.enable_nacm` refers to another node inside `nacm`, called `enable-nacm`. This node is a leaf, holding a value, which you can print out with the Python `print()` function. Doing so is conceptually the same as using the `show running-config nacm enable-nacm` command in the CLI.

There is a small difference, however. Notice that in the CLI the `enable-nacm` is hyphenated, as this is the actual node name in YANG. But names must not include the hyphen (minus) sign in Python, so the Python code uses an underscore instead.

The following is the full source code that prints the value:

Example 3. Reading a value in Python

```
import ncs  
  
with ncs.maapi.single_read_trans('admin', 'python') as t:  
    root = ncs.maagic.get_root(t)  
    print(root.nacm.enable_nacm)
```

As you can see in this example, it is necessary to import only the `ncs` module, which automatically imports all the submodules. Depending on your NSO instance, you might also notice that the value printed

is `True`, without any quotation marks. As a convenience, the value gets automatically converted to the best-matching Python type, which in this case is a boolean value (`True` or `False`).

Moreover, if you start a read/write transaction instead of a read-only one, you can also assign a new value to the leaf. Of course the same validation rules apply as using the CLI and you need to explicitly commit the transaction if you want the changes to persist. A call to the `apply()` method on the transaction object `t` performs this function. Here is an example:

Example 4. Writing a value in Python

```
import ncs

with ncs.maapi.single_write_trans('admin', 'python') as t:
    root = ncs.maagic.get_root(t)
    root.nacm.enable_nacm = True
    t.apply()
```

Lists

You can access a YANG list node in a manner similar to how you access a leaf. However, working with a list more resembles working with a Python `dict` than a list, even though the name would suggest otherwise. The distinguishing feature is that YANG lists have keys that uniquely identify each list item. So, lists are more naturally represented as a kind of a dictionary in Python.

Let's say there is a list of customers defined in NSO, with a YANG schema such as:

```
container customers {
    list customer {
        key "id";
        leaf id {
            type string;
        }
    }
}
```

To simplify the code, you might want to assign the value of `root.customers.customer` to a new variable `our_customers`. Then you can easily access individual customers (list items) by their `id`. For example, `our_customers['ACME']` would select the customer with `id` equal to ACME. You can check for the existence of an item in a list using the Python `in` operator, for example, '`'ACME'` in `our_customers`. Having selected a specific customer using the square bracket syntax, you can then access the other nodes of this item.

Compared to dictionaries, making changes to YANG lists is quite a bit different. You cannot just add arbitrary items because they must obey the YANG schema rules. Instead, you call the `create()` method on the list object and provide the value for the key. This method creates and returns a new item in the list if it doesn't exist yet. Otherwise, the method returns the existing item. And for item removal, use the Python built-in `del` function with the list object and specify the item to delete. For example, `del our_customers['ACME']` deletes the ACME customer entry.

In some situations, you might want to enumerate all of the list items. Here, the list object can be used with the Python `for` syntax, which iterates through each list item in turn. Note that this differs from standard Python dictionaries, which iterate through the keys. The following example demonstrates this behavior.

Example 5. Using lists with Python

```
import ncs
```

```

with ncs.maapi.single_write_trans('admin', 'python') as t:
    root = ncs.maagic.get_root(t)
    our_customers = root.customers.customer

    new_customer = our_customers.create('ACME')
    new_customer.status = 'active'

    for c in our_customers:
        print(c.id)

    del our_customers['ACME']
    t.apply()

```

Now let's see how you can use this knowledge for network automation.

Showcase: Configuring DNS with Python

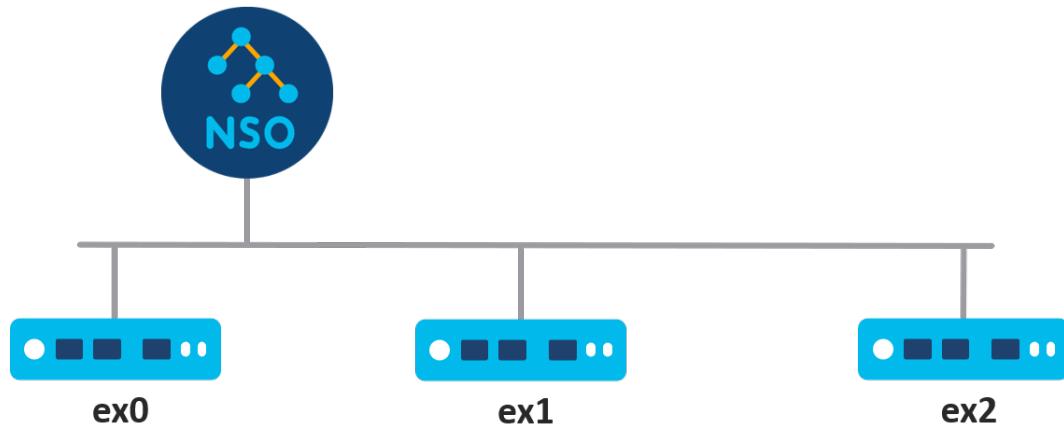
Prerequisites:

- No previous NSO or netsim processes are running. Use the **ncs --stop** and **ncs-netsim stop** commands to stop them if necessary.

Step 1: Start the routers

Leveraging one of the examples included with the NSO installation allows you to quickly gain access to an NSO instance with a few devices already onboarded. The “getting-started/developing-with-ncs” set of examples contains three simulated routers that you can configure.

Figure 6. The lab topology



Navigate to the 0-router-network directory with the following command:

```
$ cd ${NCS_DIR}/examples.ncs/getting-started/developing-with-ncs/0-router-network
```

You can prepare and start the routers by running the **make** and **netsim** commands from this directory.

```
$ make clean all && ncs-netsim start
```

With the routers running, you should also start the NSO instance that will allow you to manage them.

```
$ ncs
```

In case the `ncs` command reports an error about an address already in use, you have another NSO instance already running that you must stop first (`ncs --stop`).

Step 2: Inspect the device data model

Before you can use Python to configure the router, you need to know what to configure. The simplest way to find out how to configure the DNS on this type of a router is by using the NSO CLI.

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

In the CLI, you can verify that the NSO is managing three routers and check their names with the following command:

```
admin@ncs# show devices list
```

To make sure that the NSO configuration matches the one deployed on routers, also perform a sync-from action.

```
admin@ncs# devices sync-from
```

Let's say you would like to configure the DNS server 192.0.2.1 on the ex1 router. To do this by hand, first enter the configuration mode.

```
admin@ncs# config
```

Then navigate to the NSO copy of the ex1 configuration, which resides under the `devices device ex1 config` path, and use the `?` and `TAB` keys to explore the available configuration options. You are looking for the DNS configuration.

```
admin@ncs(config)# devices device ex1 config
```

Once you have found it, you see the full DNS server configuration path is:

```
devices device ex1 config sys dns server
```

As an alternative to using the CLI approach to find this path, you could also consult the data model of the router in the `packages/router/src/yang/` directory.

As you won't be configuring ex1 manually at this point, exit the configuration mode.

```
admin@ncs(config)# abort
```

Instead, you will create a Python script to do it, so exit the CLI as well.

```
admin@ncs# exit
```

Step 3: Create the script

You will place the script into the `ex1-dns.py` file. In a text editor, create a new file and add the following text at the start:

```
import ncs

with ncs.maapi.single_write_trans('admin', 'python') as t:
    root = ncs.maagic.get_root(t)
```

The `root` variable allows you to access configuration in the NSO, much like entering the configuration mode on the CLI does.

Next, you will need to navigate to the ex1 router. It makes sense to assign it to the `ex1_device` variable, which makes it more obvious what it refers to and easier to access in the script.

Step 4: Run and verify the script

```
ex1_device = root.devices.device['ex1']
```

In NSO, each managed device, such as the ex1 router, is an entry inside the `device` list. The list itself is located in the `devices` container, which is a common practice for lists.

The list entry for ex1 includes another container, `config`, where the copy of ex1 configuration is kept. Assign it to the `ex1_config` variable.

```
ex1_config = ex1_device.config
```

Alternatively, you can assign to `ex1_config` directly, without referring to `ex1_device`, like so:

```
ex1_config = root.devices.device['ex1'].config
```

This is the equivalent of using `devices device ex1 config` on the CLI.

For the last part, keep in mind the full configuration path you found earlier. You have to keep navigating to reach the `server` list node. You can do this through the `sys` and `dns` nodes on the `ex1_config` variable.

```
dns_server_list = ex1_config.sys.dns.server
```

DNS configuration typically allows specifying multiple servers for redundancy and is therefore modeled as a list. You add a new DNS server with the `create()` method on the list object.

```
dns_server_list.create('192.0.2.1')
```

Having made the changes, do not forget to commit them with a call to `apply()` or they will be lost.

```
t.apply()
```

Lastly, add a simple print statement to notify you when the script completes.

```
print('Done!')
```

Step 4: Run and verify the script

Save the script file as `ex1-dns.py` and run it with the `python3` command:

```
$ python3 ex1-dns.py
```

You should see `Done!` printed out. Then start the NSO CLI to verify the configuration change.

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

Finally, you can check the configured DNS servers on ex1 by using the `show running-config` command.

```
admin@ncs# show running-config devices device ex1 config sys dns server
```

If you see the 192.0.2.1 address in the output, you have successfully configured this device using Python!

The Magic Behind the API

Perhaps you've wondered about the unusual name of the Python `ncs.maagic` module? It is not a typo but a portmanteau of the words Management Agent API (MAAPI) and magic. The latter is used in the context of so-called magic methods in Python. The purpose of magic methods is to allow custom code to play nicely with the Python language. An example you might have come across in the past is the `__init__()` method in a class, which gets called whenever you create a new object. This one and similar methods are called magic because they are invoked automatically and behind-the-scenes (implicitly).

The NSO Python API makes extensive use of such magic methods in the `ncs.maagic` module. Magic methods help this module translate an object-based, user-friendly programming interface into low-level function calls. In turn, the high-level approach to navigating the data hierarchy with `ncs.maagic` objects is called the *Python Maagic API*.



CHAPTER 3

Creating a Service

The device YANG models contained in the Network Element Drivers (NEDs) enable NSO to store device configurations in the CDB and expose a uniform API to the network for automation, such as by Python scripts. The concept of NSO services builds on top of this network API and adds the ability to store service-specific parameters with each service instance.

In this chapter, you will learn about the parts that make up a service and how to build one yourself.

- [Why services?, page 19](#)
- [The Service Package, page 20](#)
- [Service Parameters, page 22](#)
- [Showcase: A Simple DNS Configuration Service, page 23](#)
- [Service Templates, page 27](#)
- [Showcase: DNS Configuration Service with Templates, page 29](#)

Why services?

Network automation includes provisioning and deprovisioning configuration, even though the deprovisioning part often doesn't get as much attention. It is nevertheless significant since leftover, residual configuration can cause hard-to-diagnose operational problems. Even more importantly, without proper deprovisioning, seemingly trivial changes may prove hard to implement correctly.

Consider the following example. You create a simple script that configures a DNS server on a router, by adding the IP address of the server to the DNS server list. This should work fine for initial provisioning. However, when the IP address of the DNS server changes, the configuration on the router should be updated as well.

Can you still use the same script in this case? Most likely not, since you need to remove the old server from configuration and add the new one. The original script would just add the new IP address after the old one, resulting in both entries on the device. In turn, the device may experience slow connectivity as the system periodically retries the old DNS IP address and eventually times out.

The following figure illustrates this process, where a simple script first configures the IP address 192.0.2.1 (“.1”) as the DNS server, then later configures 192.0.2.8 (“.8”), resulting in a leftover old entry (“.1”).

Figure 7. DNS configuration with a simple script



In such situation the script could perhaps simply replace the existing configuration, by removing all existing DNS server entries before adding the new one. But is this a reliable practice? What if a device requires an additional DNS server that an administrator configured manually? It would be overwritten and lost.

In general, the safest approach is to keep track of the previous changes and only replace the parts that have actually changed. This, however, is a lot of work and nontrivial to implement yourself. Fortunately, NSO provides such functionality through the FASTMAP algorithm, which is used when deploying services.

The other major benefit of using NSO services for automation is service interface definition using YANG, which specifies the name and format of the service parameters. Many new NSO users wonder why use a service YANG model when they could just use the Python code or templates directly. While it might be difficult to see the benefits without much prior experience, YANG allows you to write better, more maintainable code, which simplifies the solution in the long run.

Many, if not most, security issues and provisioning bugs stem from unexpected user input. You must always validate user input (service parameter values) and YANG compels you to think about that when writing the service model. It also makes it easy to write the validation rules by using a standardized syntax, specifically designed for this purpose.

Moreover, separation of concerns into the user interface, validation, and provisioning code allows for better organization, which becomes extremely important as the project grows. It also gives NSO the ability to automatically expose the service functionality through its APIs for integration with other systems.

For these reasons, services are the preferred way of implementing network automation in NSO.

The Service Package

As you may already know, services are added to NSO with packages. Therefore, you need to create a package if you want to implement a service of your own. NSO ships with an **ncs-make-package** utility that makes creating packages effortless. Adding the `--service-skeleton` python option creates a service skeleton, that is, an empty service, which you can tailor to your needs. As the last argument you must specify the package name, which in this case is the service name. The command then creates a new directory with that name and places all the required files in the appropriate subdirectories.

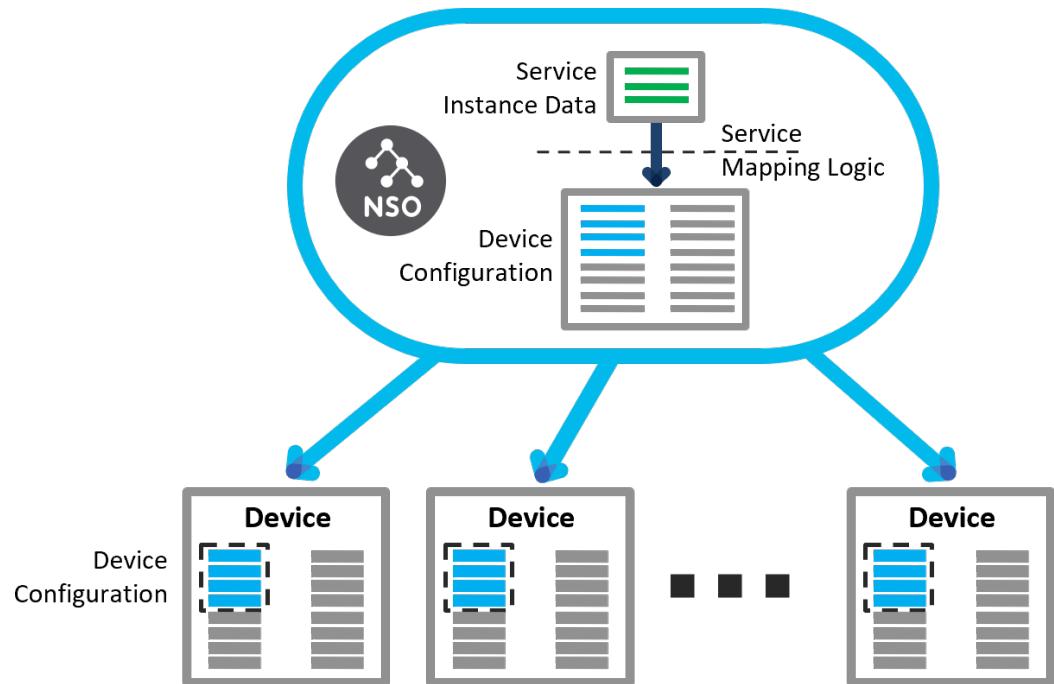
The package contains the two most important parts of the service:

- the service YANG model and
- the service provisioning code, also called the mapping logic.

Let's first look at the provisioning part. This is the code that performs the network configuration necessary for your service. The code often includes some parameters, for example, the DNS server IP address or addresses to use if your service is in charge of DNS configuration. So, we say that the code *maps* the

service parameters into the device parameters, which is where the term *mapping logic* originates from. NSO, with the help of the NED, then translates the device parameters to the actual configuration. This simple tree-to-tree mapping describes how to create the service and NSO automatically infers how to update, remove or re-deploy the service, hence the name FASTMAP.

Figure 8. Transformation of service parameters into device configurations



How do you create the provisioning code and where do you place it? Is it similar to a stand-alone Python script? Indeed, the code is mostly the same. The main difference is that now you don't have to create a session and a transaction yourself because NSO already provides you with one. Through this transaction the system tracks the changes to configuration made by your code.

The package skeleton contains a directory called `python`. It holds a Python package named after your service. In the package, the `ServiceCallbacks` class (the `main.py` file) is used for provisioning code. The same file also contains the `Main` class, which is responsible for registering the `ServiceCallbacks` class as service provisioning code with NSO.

Of the most interest is the `cb_create()` method of the `ServiceCallbacks` class:

```
def cb_create(self, tctx, root, service, proplist)
```

NSO calls this method for service provisioning. Now, let's see how to evolve a stand-alone automation script into a service. Suppose you have Python code for DNS configuration on a router, similar to the following:

```
with ncs.maapi.single_write_trans('admin', 'python') as t:
    root = ncs.maagic.get_root(t)

    ex1_device = root.devices.device['ex1']
    ex1_config = ex1_device.config
    dns_server_list = ex1_config.sys.dns.server
    dns_server_list.create('192.0.2.1')
```

```
t.apply()
```

Taking into account the `cb_create()` signature and the fact that the NSO manages the transaction for a service, you won't be needing the transaction and `root` variable setup. The NSO service framework already takes care of setting up the `root` variable with the right transaction. There is also no need to call `apply()` because NSO does that automatically.

You only have to provide the core of the code (the middle portion in the above stand-alone script) to the `cb_create()`:

```
def cb_create(self, tctx, root, service, proplist):
    ex1_device = root.devices.device['ex1']
    ex1_config = ex1_device.config
    dns_server_list = ex1_config.sys.dns.server
    dns_server_list.create('192.0.2.1')
```

You can run this code by adding the service package to NSO and provisioning a service instance. It will achieve the same effect as the stand-alone script but with all the benefits of a service, such as tracking changes.

Service Parameters

In practice, all services have some variable parameters. Most often parameter values change from service instance to service instance, as the desired configuration is a little bit different for each of them. They may differ in the actual IP address that they configure or in whether the switch for some feature is on or off. Even the DNS configuration service requires a DNS server IP address, which may be the same across the whole network but could change with time if the DNS server is moved elsewhere. Therefore, it makes sense to expose the variable parts of the service as service parameters. This allows a service operator to set the parameter value without changing the service provisioning code.

With NSO, service parameters are defined in the service model, written in YANG. The YANG module describing your service is part of the service package, located under the `src/yang` path, and customarily named the same as the package. In addition to the module-related statements (`description`, `revision`, `imports`, and so on), a typical service module includes a YANG `list`, named after the service. Having a list allows you to configure multiple service instances with slightly different parameter values. For example, in a DNS configuration service, you might have multiple service instances with different DNS servers. The reason being, some devices, such as those in the Demilitarized Zone (DMZ), might not have access to the internal DNS servers and would need to use a different set.

The service model skeleton already contains such a list statement. The following is another example, similar to the one in the skeleton:

```
list my-svc {
    description "This is an RFS skeleton service";

    key name;
    leaf name {
        tailf:info "Unique service id";
        tailf:cli-allow-range;
        type string;
    }

    uses ncs:service-data;
    ncs:servicepoint my-svc-servicepoint;

    // Devices configured by this service instance
    leaf-list device {
        type leafref {
```

```

        path "/ncs:devices/ncs:device/ncs:name";
    }

    // An example generic parameter
    leaf server-ip {
        type inet:ipv4-address;
    }
}

```

Along with the description, the service specifies a key, name, to uniquely identify each service instance. This can be any free-form text, as denoted by its type (string). The statements starting with `tailf:` are NSO-specific extensions for customizing the user interface NSO presents for this service. After that come two lines, the `uses` and `ncs:servicepoint`, that tell NSO this is a service and not just some ordinary list. At the end, there are two parameters defined, `device` and `server-ip`.

NSO then allows you to add the values for these parameters when configuring a service instance, as shown in the following CLI transcript:

```

admin@ncs(config)# my-svc instance1 ?
Possible completions:
  check-sync           Check if device config is according to the service
  commit-queue
  deep-check-sync     Check if device config is according to the service
  device
  < ... output omitted ... >
  server-ip
  < ... output omitted ... >

```

Finally, your Python script can read the supplied values inside the `cb_create()` method via the provided `service` variable. This variable points to the currently-provisioning service instance, allowing you to use code such as `service.server_ip` for the value of the `server-ip` parameter.

Showcase: A Simple DNS Configuration Service

Prerequisites:

- No previous NSO or netsim processes are running. Use the `ncs --stop` and `ncs-netsim stop` commands to stop them if necessary.
- NSO local install with a fresh runtime directory has been created by the `ncs-setup --dest ~/nso-lab-rundir` or similar command.
- The environment variable `NSO_RUNDIR` points to this runtime directory, such as set by the `export NSO_RUNDIR=~/nso-lab-rundir` command. It enables the below commands to work as-is, without additional substitution needed.

Step 1: Prepare simulated routers

The “getting-started/developing-with-ncs” set of examples contains three simulated routers that you can use for this scenario. The `0-router-network` directory holds the data necessary for starting the routers and connecting them to your NSO instance. First, change the current working directory:

```
$ cd $NCS_DIR/examples.ncs/getting-started/developing-with-ncs/0-router-network
```

From this directory, you can start a fresh set of routers by running the following `make` command:

```
$ make showcase-clean-start
< ... output omitted ... >
DEVICE ex0 OK STARTED
```

Step 2: Create a service package

```
DEVICE ex1 OK STARTED
DEVICE ex2 OK STARTED
make: Leaving directory 'examples.ncs/getting-started/developing-with-ncs/0-router-network'
```

The routers are now running. The required NED package and a CDB initialization file, ncs-cdb/ncs_init.xml, were also added to your NSO instance. The latter contains connection details for the routers and will be automatically loaded on the first NSO start.

In case you're not using a fresh working directory, you may need to use the **ncs_load** command to load the file manually. Older versions of the system may also be missing the above **make** target, which you can add to the **Makefile** yourself:

```
showcase-clean-start:
    $(MAKE) clean all
    cp ncs-cdb/ncs_init.xml ${NSO_RUNDIR}/ncs-cdb/
    cp -a ../packages/router ${NSO_RUNDIR}/packages/
    ncs-netsim start
```

Step 2: Create a service package

You create a new service package with the **ncs-make-package** command. Without the **--dest** option, the package is created in the current working directory. Normally you run the command without this option, as it is shorter. For NSO to find and load this package, it has to be placed (or referenced via a symbolic link) in the **packages** subfolder of the NSO running directory. So, change the current working directory before creating the package:

```
$ cd ${NSO_RUNDIR}/packages
```

You need to provide two parameters to **ncs-make-package**. The first is the **--service-skeleton** python option, which selects the Python programming language for scaffolding code. The second parameter is the name of the service. As you are creating a service for DNS configuration, **dns-config** is a fitting name for it. Run the final, full command:

```
$ ncs-make-package --service-skeleton python dns-config
```

If you look at the file structure of the newly created package, you will see it contains a number of files.

```
dns-config/
+-- package-meta-data.xml
+-- python
|   '-- dns_config
|       '-- __init__.py
|       '-- main.py
+-- README
+-- src
|   '-- Makefile
|   '-- yang
|       '-- dns-config.yang
+-- templates
'-- test
    '-- < ... output omitted ... >
```

The **package-meta-data.xml** describes the package and tells NSO where to find the code. Inside the **python** folder is a service-specific Python package, where you add your own Python code (to **main.py** file). There is also a **README** file that you can update with the information relevant to your service. The **src** folder holds the source code that you must compile before you can use it with NSO. That's why there is also a **Makefile** that takes care of the compilation process. In the **yang** subfolder is the service YANG module. The **templates** folder can contain additional XML files, discussed later. Lastly, there's the **test** folder where you can put automated testing scripts, which won't be discussed here.

Step 3: Add the DNS server parameter

While you can always hard-code the desired parameters, such as the DNS server IP address, in the Python code, it means you have to change the code every time the parameter value (the IP address) changes. Instead, you can define it as an input parameter in the YANG file. Fortunately, the skeleton already has a leaf called dummy that you can rename and use for this purpose.

Open the `dns-config.yang`, located inside `dns-config/src/yang/`, in a text or code editor and find the following line:

```
leaf dummy {
```

Replace the word “dummy” with the word “dns-server”, save the file and return to the shell. Run the `make` command in the `dns-config/src` folder to compile the updated YANG file.

```
$ make -C dns-config/src
make: Entering directory 'dns-config/src'
mkdir -p ..load-dir
mkdir -p java/src/
bin/ncsc `ls dns-config-ann.yang > /dev/null 2>&1 && echo "-a dns-config-ann.yang"` \
         -c -o ..load-dir/dns-config.fxs yang/dns-config.yang
make: Leaving directory 'dns-config/src'
```

Step 4: Add Python code

In a text or code editor open the `main.py` file, located inside `dns-config/python/dns_config/`. Find the following snippet:

```
@Service.create
def cb_create(self, tctx, root, service, proplist):
    self.log.info('Service create(service=', service._path, ')')
```

Right after the `self.log.info()` call, read the value of the `dns-server` parameter into a `dns_ip` variable:

```
dns_ip = service.dns_server
```

Mind the 8 spaces in front to make sure the line is correctly aligned. After that, add the code that configures the ex1 router:

```
ex1_device = root.devices.device['ex1']
ex1_config = ex1_device.config
dns_server_list = ex1_config.sys.dns.server
if dns_ip not in dns_server_list:
    dns_server_list.create(dns_ip)
```

Here, you are using the `dns_ip` variable that contains the operator-provided IP address instead of a hard-coded value. In the end, the `cb_create()` method should look like the following:

```
@Service.create
def cb_create(self, tctx, root, service, proplist):
    self.log.info('Service create(service=', service._path, ')')
    dns_ip = service.dns_server
    ex1_device = root.devices.device['ex1']
    ex1_config = ex1_device.config
    dns_server_list = ex1_config.sys.dns.server
    if dns_ip not in dns_server_list:
        dns_server_list.create(dns_ip)
```

Save the file and let's see the service in action!

Step 5: Deploy the service

Start the NSO from the running directory:

```
$ cd $NSO_RUNDIR; ncs
```

Then start the NSO CLI:

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

If you have started a fresh NSO instance, the packages are loaded automatically. Still, there's no harm in requesting a package reload anyway:

```
admin@ncs# packages reload
reload-result {
    package dns-config
    result true
}
reload-result {
    package router-nc-1.0
    result true
}
```

As you will be making changes on the simulated routers, make sure NSO has their current configuration with the **devices sync-from** command.

```
admin@ncs# devices sync-from
sync-result {
    device ex0
    result true
}
sync-result {
    device ex1
    result true
}
sync-result {
    device ex2
    result true
}
```

Now you can test out your service package by configuring a service instance. First, enter the configuration mode.

```
admin@ncs# config
```

Configure a test instance and specify the DNS server IP address:

```
admin@ncs(config)# dns-config test dns-server 192.0.2.1
```

The easiest way to see configuration changes from the service code is to use the **commit dry-run** command.

```
admin@ncs(config-dns-config-test)# commit dry-run
cli {
    local-node {
        data devices {
            device ex1 {
                config {
                    sys {
                        dns {
                            + # after server 10.2.3.4
```

```
+           server 192.0.2.1;
+       }
+   }
}
}
+dns-config test {
+   dns-server 192.0.2.1;
+}
}
```

The output tells you the new DNS server is being added in addition to an existing one already there. Commit the changes:

```
admin@ncs(config-dns-config-test)# commit
```

Finally, change the IP address of the DNS server:

```
admin@ncs(config-dns-config-test)# dns-server 192.0.2.8
```

With the help of **commit dry-run** observe how the old IP address gets replaced with the new one, without any special code needed for provisioning.

```
admin@ncs(config-dns-config-test)# commit dry-run
cli {
  local-node {
    data devices {
      device ex1 {
        config {
          sys {
            dns {
              -         server 192.0.2.1;
              +         # after server 10.2.3.4
              +         server 192.0.2.8;
            }
          }
        }
      }
    }
  }
  dns-config test {
    -   dns-server 192.0.2.1;
    +   dns-server 192.0.2.8;
  }
}
```

Service Templates

The DNS configuration example intentionally performs very little configuration, a single line really, in order to focus on the service concepts. In practice, services can become more complex in two different ways. First, the DNS configuration service takes the IP address of the DNS server as an input parameter, supplied by the operator. Instead, the provisioning code could leverage another system, such as an IP Address Management (IPAM), to get the required information. In such cases, you have to add additional logic to your service code to generate the parameters (variables) to be used for configuration.

Second, generating the configuration from the parameters can become more complex when it touches multiple subsystems or spans across multiple devices. An example would be a service that adds a new VLAN, configures an IP address and a DHCP server, and adds the new route to a routing protocol. Or perhaps the service has to be duplicated on two separate devices for redundancy.

An established approach to the second challenge is to use a templating system for configuration generation. Templates separate the process of constructing parameter values from how they are used, adding a degree of flexibility and decoupling. NSO uses XML-based *configuration (config) templates*, which you can invoke from provisioning code or link directly to services. In the latter case, you don't even have to write any Python code.

XML templates are snippets of configuration, similar to the CDB init files, but more powerful. Let's see how you could implement the DNS configuration service using a template instead of navigating the data model with Python.

While you are free to write an XML template by hand, it has to follow the target data model. Fortunately, the NSO CLI can help you and do most of the hard work for you. First, you'll need a sample instance with the desired configuration. As you are configuring the DNS server on a router and the ex1 device already has one configured, you can just reuse that one. Otherwise, you might configure one by hand, using the CLI. You do that by displaying the existing configuration in the XML format and saving it to a file, by piping it through the **display xml** and **save** filters, as shown here:

```
admin@ncs# show running-config devices device ex1 config sys dns | display xml
<config xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>ex1</name>
      <config>
        <sys xmlns="http://example.com/router">
          <dns>
            <server>
              <address>192.0.2.1</address>
            </server>
          </dns>
        </sys>
      </config>
    </device>
  </devices>
</config>
admin@ncs# show running-config devices device ex1 config sys dns | \
  display xml | save template.xml
```

The file structure of a package usually contains a `templates` folder and that is where the template belongs. When loading packages, NSO will scan this folder and process any `.xml` files it finds as templates.

Of course, a template with hard-coded values is of limited use, as it would always produce the exact same configuration. It becomes a lot more useful with variable substitution. In its simplest form, you define a variable value in the provisioning (Python) code and reference it from the XML template, by using curly braces and a dollar sign: `{ $VARIABLE }`. Also, many users prefer to keep the variable name uppercased to make it stand out more from the other XML elements in the file. For example, in the template XML file for the DNS service, you would likely replace the IP address `192.0.2.1` with the variable `{ $DNS_IP }` to control its value from the Python code.

You apply the template by creating a new `ncs.template.Template` object and calling its `apply()` method. This method takes the name of the XML template as the first parameter, without the trailing `.xml` extension, and an object of type `ncs.template.Variables` as the second parameter. Using the `Variables` object, you provide values for the variables in the template.

```
template_vars = ncs.template.Variables()
template_vars.add('VARIABLE', 'some value')

template = ncs.template.Template(service)
```

```
template.apply('template', template_vars)
```

In fact, variables in a template can take a more complex form of an XPath expression, where the parameter for the `Template` constructor comes into play. This parameter defines the root node (starting point) when evaluating XPath paths. Use the provided `service` variable, unless you specifically need a different value. It is what the so-called template-based services use as well.

Template-based services are no-code, pure template services that only contain a YANG model and an XML template. Since there is no code to set the variables, they must rely on XPath for the dynamic parts of the template. Such services still have a YANG data model with service parameters, that XPath can access. For example, if you have a parameter leaf defined in the service YANG file by the name `dns-server`, you can refer to its value with the `{ /dns-server }` code in the XML template.

Likewise, you can use the same XPath in a template of a Python service. Then you don't have to add this parameter to the `variables` object but can still access its value in the template, saving you a little bit of Python code.

Showcase: DNS Configuration Service with Templates

Prerequisites:

- No previous NSO or netsim processes are running. Use the `ncs --stop` and `ncs-netsim stop` commands to stop them if necessary.
- NSO local install with a fresh runtime directory has been created by the `ncs-setup --dest ~/nso-lab-rundir` or similar command.
- The environment variable `NSO_RUNDIR` points to this runtime directory, such as set by the `export NSO_RUNDIR=~/nso-lab-rundir` command. It enables the below commands to work as-is, without additional substitution needed.

Step 1: Prepare simulated routers

The “getting-started/developing-with-ncs” set of examples contains three simulated routers that you can use for this scenario. The `0-router-network` directory holds the data necessary for starting the routers and connecting them to your NSO instance. First, change the current working directory:

```
$ cd $NCS_DIR/examples.ncs/getting-started/developing-with-ncs/0-router-network
```

From this directory, you can start a fresh set of routers by running the following `make` command:

```
$ make showcase-clean-start
< ... output omitted ... >
DEVICE ex0 OK STARTED
DEVICE ex1 OK STARTED
DEVICE ex2 OK STARTED
make: Leaving directory 'examples.ncs/getting-started/developing-with-ncs/0-router-network'
```

The routers are now running. The required NED package and a CDB initialization file, `ncs-cdb/ncs_init.xml`, were also added to your NSO instance. The latter contains connection details for the routers and will be automatically loaded on the first NSO start.

In case you're not using a fresh working directory, you may need to use the `ncs_load` command to load the file manually. Older versions of the system may also be missing the above `make` target, which you can add to the `Makefile` yourself:

```
showcase-clean-start:
    $(MAKE) clean all
    cp ncs-cdb/ncs_init.xml ${NSO_RUNDIR}/ncs-cdb/
```

Step 2: Create a service

```
cp -a ./packages/router ${NSO_RUNDIR}/packages/
ncs-netsim start
```

Step 2: Create a service

The DNS configuration service that you are implementing will have three parts: the YANG model, the service code, and the XML template. You will put all of these in a package named “dns-config”. First, navigate to the packages subdirectory:

```
$ cd ${NSO_RUNDIR}/packages
```

Then run the following command to set up the service package:

```
$ ncs-make-package --build --service-skeleton python dns-config
bin/ncsc `ls dns-config-ann.yang > /dev/null 2>&1 && echo "-a dns-config-ann.yang" ` \
-c -o ../load-dir/dns-config.fxs yang/dns-config.yang
```

In case you are building on top of the previous showcase, the package folder may already exist and will be updated.

You can leave the YANG model as is for this scenario but you need to add some Python code that will apply an XML template during provisioning. In a text or code editor open the `main.py` file, located inside `dns-config/python/dns_config/`, and find the definition for the `cb_create()` function:

```
@Service.create
def cb_create(self, tctx, root, service, proplist):
    ...
```

You will define one variable for the template, the IP address of the DNS server. To pass its value to the template, you have to create the `Variables` object and add each variable, along with its value. Replace the body of the `cb_create()` function with the following:

```
template_vars = ncs.template.Variables()
template_vars.add('DNS_IP', '192.0.2.1')
```

The `template_vars` object now contains a value for the `DNS_IP` template variable, to be used with the `apply()` method that you are adding next:

```
template = ncs.template.Template(service)
template.apply('dns-config-tpl', template_vars)
```

Here, the first argument to `apply()` defines the template to use. In particular, using “dns-config-tpl”, you are requesting the template from the `dns-config-tpl.xml` file, which you will be creating shortly.

This is all the Python code that is required. The final, complete `cb_create` method is as follows:

```
@Service.create
def cb_create(self, tctx, root, service, proplist):
    template_vars = ncs.template.Variables()
    template_vars.add('DNS_IP', '192.0.2.1')
    template = ncs.template.Template(service)
    template.apply('dns-config-tpl', template_vars)
```

Step 3: Create a template

The most straightforward way to create an XML template is by using the NSO CLI. Return to the running directory and start the NSO:

```
$ cd ${NSO_RUNDIR} && ncs --with-package-reload
```

The `--with-package-reload` option will make sure NSO loads any added packages and save a **packages reload** command on the NSO CLI.

Next, start the NSO CLI:

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

As you are starting with a new NSO instance, first invoke the sync-from action.

```
admin@ncs# devices sync-from
sync-result {
    device ex0
    result true
}
sync-result {
    device ex1
    result true
}
sync-result {
    device ex2
    result true
}
```

Next, make sure that the ex1 router already has an existing entry for a DNS server in its configuration.

```
admin@ncs# show running-config devices device ex1 config sys dns
devices device ex1
config
    sys dns server 10.2.3.4
!
!
```

Pipe the command through the **display xml** and **save** CLI filters to save this configuration in an XML format. According to the Python code, you need to create a template file `dns-config-tpl.xml`. Use `packages/dns-config/templates/dns-config-tpl.xml` for the full file path.

```
admin@ncs# show running-config devices device ex1 config sys dns \
| display xml | save packages/dns-config/templates/dns-config-tpl.xml
```

At this point you have created a complete template that will provision the 10.2.3.4 as the DNS server on the ex1 device. The only problem is, the IP address is not the one you have specified in the Python code. To correct that, open the `dns-config-tpl.xml` file in a text editor and replace the line that reads `<address>10.2.3.4</address>` with the following:

```
<address>{$DNS_IP}</address>
```

The only static part left in the template now is the target device and it's possible to parameterize that, too. The skeleton, created by the **ncs-make-package** command, already contains a node `device` in the service YANG file. It is there to allow the service operator to choose the target device to be configured.

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

One way to use the `device` service parameter is to read its value in the Python code and then set up the template parameters accordingly. However, there is a simpler way with XPath. In the template, replace the line that reads `<name>ex1</name>` with the following:

```
<name>{/device}</name>
```

Step 4: Test the service

The XPath expression inside the curly braces instructs NSO to get the value for device name from the service instance's data, namely the node called `device`. In other words, when configuring a new service instance, you have to add the `device` parameter, which selects the router for provisioning. The final XML template is then:

```
<config xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>{/device}</name>
      <config>
        <sys xmlns="http://example.com/router">
          <dns>
            <server>
              <address>{$DNS_IP}</address>
            </server>
          </dns>
        </sys>
      </config>
    </device>
  </devices>
</config>
```

Step 4: Test the service

Remember to save the template file and return to the NSO CLI. Because you have updated the service code, you have to redeploy it for NSO to pick up the changes:

```
admin@ncs# packages package dns-config redeploy
result true]
```

Alternatively, you could call the **packages reload** command, which does a full reload of all the packages.

Next, enter the configuration mode:

```
admin@ncs# config
```

As you are using the `device` node in the service model for target router selection, configure a service instance for the `ex2` router in the following way:

```
admin@ncs(config)# dns-config dns-for-ex2 device ex2
```

Finally, using the **commit dry-run** command, observe the `ex2` router being configured with an additional DNS server.

```
admin@ncs(config-dns-config-dns-for-ex2)# commit dry-run
```

As a bonus for using an XPath expression to a leaf-list in the service template, you can actually select multiple router devices in a single service instance and they will all be configured.



CHAPTER 4

Applications in NSO

Services are the foundation of network automation in NSO, used to provide network-wide configuration. Configuration is an important aspect of automation, but it is not the only one. A service test action is a useful tool in many situations, from verifying initial provisioning to simplifying troubleshooting if issues arise. The test often relies on executing various checks, such as running a **ping** command to verify connectivity. This is fundamentally different from managing configuration. NSO helps you implement any such automation use-case through a generic application framework.

This chapter explores the concept of services as more general NSO applications and how they are implemented.

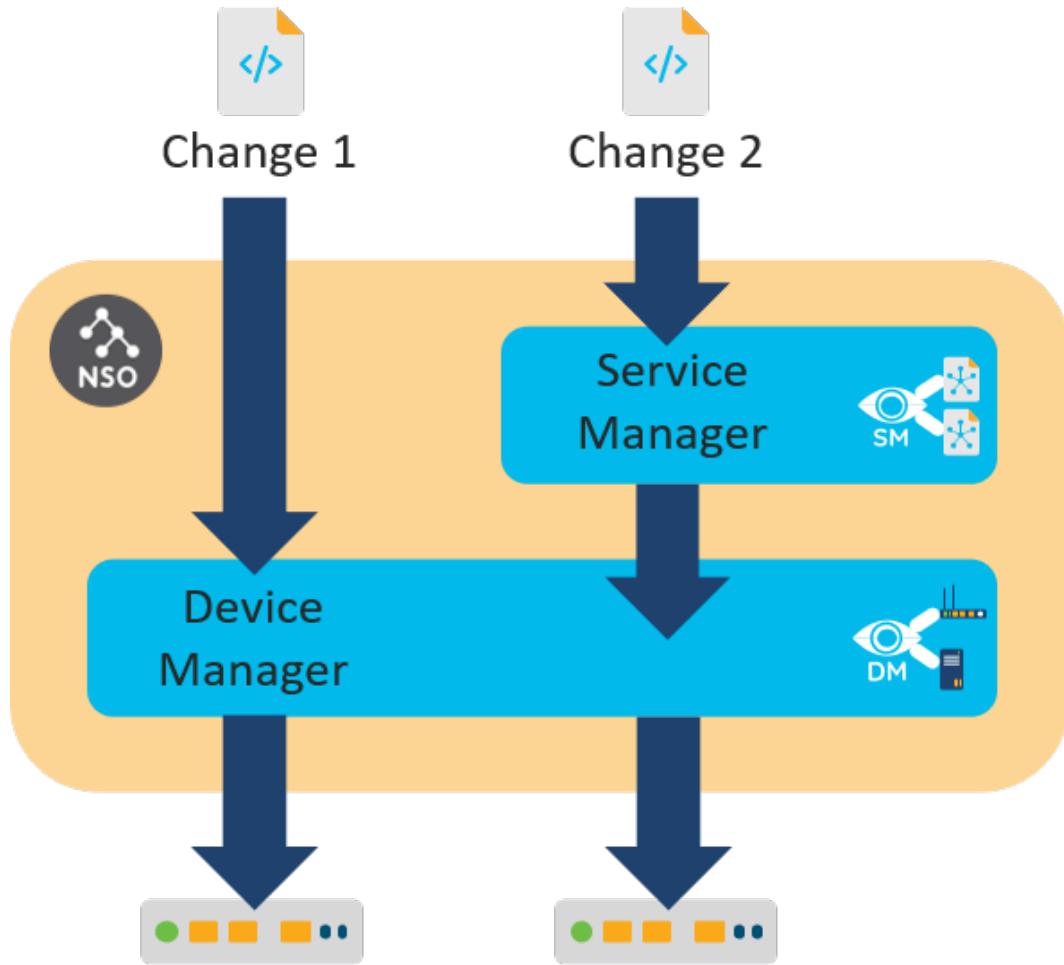
- [NSO Architecture, page 33](#)
- [Callbacks as Extension Mechanism, page 34](#)
- [Actions, page 36](#)
- [Showcase: Implementing Device Count Action, page 37](#)
- [Running Application Code, page 41](#)

NSO Architecture

You have seen two different ways in which you can make a configuration change on a network device. With the first, you make changes directly on the NSO copy of the device configuration. The *Device Manager* picks up the changes and propagates them to the affected devices.

The purpose of the Device Manager is to manage different devices in a uniform way. The Device Manager uses the Network Element Drivers (NEDs) to abstract away the different protocols and APIs towards the devices. The NED contains a YANG data model for a supported device. So, each device type requires an appropriate NED package that allows the Device Manager to handle all devices in the same, YANG-model-based way.

The second way to make configuration changes is through services. Here, the *Service Manager* adds a layer on top of the Device Manager to process the service request and enlists the help of service-aware applications to generate the device changes. The following figure illustrates the difference between the two approaches.

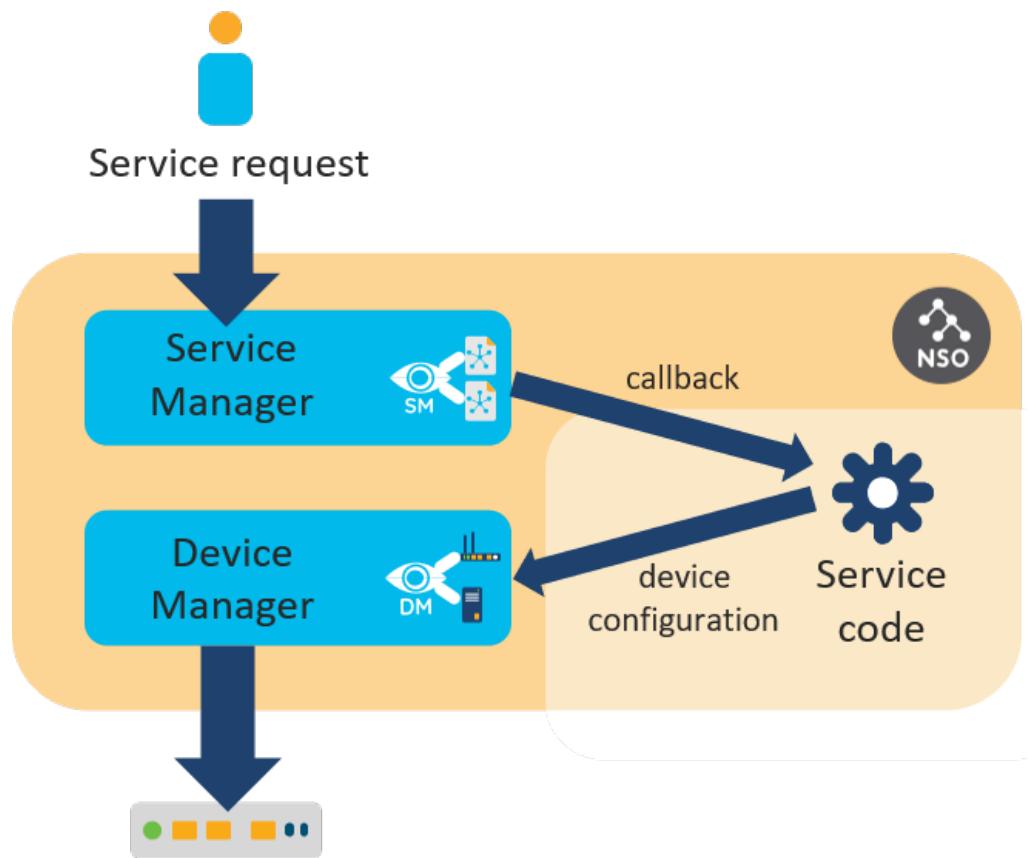
Figure 9. Device and Service Manager

Still, the Device Manager and the Service Manager are tightly integrated into one transactional engine, using the CDB to store data. Another thing the two managers have in common is packages. Like Device Manager uses NED packages to support specific devices, Service Manager relies on service packages to provide an application-specific mapping for each service type.

But a network application can consist of more than just a configuration recipe, or it may only monitor the network and not provision any configuration at all. So, a package in NSO can implement a generic application that executes custom code for specific NSO events.

Callbacks as Extension Mechanism

NSO allows augmenting the base functionality of the system by delegating certain functions to applications. As the communication must happen on demand, NSO implements a system of callbacks. Usually, the application code registers the required callbacks on start-up, then NSO can invoke each callback as needed. A prime example is a Python service, which registers the `cb_create()` function as a service callback that NSO uses to construct the actual configuration.

Figure 10. Service callback

In a Python service skeleton, callback registration happens inside a class `Main`, found in `main.py`:

```
class Main(ncs.application.Application):
    def setup(self):
        # Service callbacks require a registration for a 'service point',
        # as specified in the corresponding data model.
        #
        self.register_service('my-svc-servicepoint', ServiceCallbacks)
```

In this code, the `register_service()` method registers the `ServiceCallbacks` class to receive callbacks for a service. The first argument defines which service that is. In theory, a single class could even handle service callbacks for multiple services but that is not a common practice.

On the other hand, it is also possible that no code registered a callback for a given service. This is quite often a result of misspelling or a bug in the code that causes application code to crash. In these situations, NSO presents an error if you try to use the service:

```
Error: no registration found for callpoint my-svc-servicepoint/service_create of type=external
```

This error refers to a concept of a *service point*. Service points are declared in the service YANG model and allow NSO to distinguish ordinary data from services. They instruct NSO to invoke FASTMAP and the service callbacks when a service instance is being provisioned. That means the service skeleton YANG file also contains a service point definition, such as the following:

```
list my-svc {
    description "This is an RFS skeleton service";
```

```

uses ncs:service-data;
ncs:servicepoint my-svc-servicepoint;
}

```

Service point therefore links the definition in the model with custom code. Some methods in the code will have names starting with “cb_”, for instance the `cb_create()` method, letting you know quickly that they are an implementation of a callback.

NSO implements additional callbacks for each service point, that may be required in some specific circumstances. Most of these callbacks perform work outside of the automatic change tracking, so you need to consider that before using them. The [the section called “ Advanced Mapping Techniques ”](#) contains more details.

As well as services, other extensibility options in NSO also rely on callbacks and *call points*, a generalized version of a service point. Two notable examples are validation callbacks, to implement additional validation logic to that supported by YANG, and custom actions. The [the section called “DP API”](#) provides a comprehensive list and an overview of when to use each.

In summary, you implement custom behavior in NSO by providing the following three parts:

- A YANG model directing NSO to use callbacks, such as service point for services
- Registration of callbacks, telling NSO to call into your code at a given point
- The implementation of each callback with your custom logic

This way, an application in NSO can implement all the required functionality for a given use-case (configuration management and otherwise) by registering the right callbacks.

Actions

The most common way to implement non-configuration automation in NSO is using actions. An action represents a task or an operation that a user of the system can invoke on demand, such as downloading a file, resetting a device, or performing some test.

Like configuration, actions must also be defined in the YANG model. Each action is described by the `action` YANG statement that specifies what are its inputs and outputs, if any. Inputs allow a user of the action to provide additional information to the action invocation, while outputs provide information to the caller. Actions are a form of a Remote Procedure Call (RPC) and have historically evolved from NETCONF RPCs. It's therefore unsurprising that with NSO you implement both in a similar manner.

Let's look at an example action definition:

```

action my-test {
    tailf:actionpoint my-test-action;
    input {
        leaf test-string {
            type string;
        }
    }
    output {
        leaf has-ns0 {
            type boolean;
        }
    }
}

```

The first thing to notice in the code is that, just like services use a service point, actions use an *action point*. It is denoted by the `tailf:actionpoint` statement and tells NSO to execute a callback registered to this name. As discussed, the callback mechanism allows you to provide custom action implementation.

Correspondingly, your code needs to register a callback to this action point, by calling the `register_action()`, as demonstrated here:

```
def setup(self):
    self.register_action('my-test-action', MyTestAction)
```

The `MyTestAction` class, referenced in the call, is responsible for implementing the actual action logic and should inherit from the `ncs.dp.Action` base class. The base class will take care of calling the `cb_action()` class method when users initiate the action. The `cb_action()` is where you put your own code. The following code shows a trivial implementation of an action, that checks whether its input contains the string “NSO”:

```
class MyTestAction(Action):
    @Action.action
    def cb_action(self, uinfo, name, kp, input, output, trans):
        self.log.info('Action invoked: ', name)
        output.has_nso = 'NSO' in input.test_string
```

The `input` and `output` arguments contain input and output data, respectively, which matches the definition in the action YANG model. The example shows the value of a simple Python `in` string check that is assigned to an output value.

The `name` argument has the name of the called action (such as “my-test”), to help you distinguish which action was called in the case where you would register the same class for multiple actions. Similarly, an action may be defined on a list item and the `kp` argument contains the full keypath (a tuple) to an instance where it was called.

Finally, the `uinfo` contains information on the user invoking the action and the `trans` argument represents a transaction, that you can use to access data other than input. This transaction is read-only, as configuration changes should normally be done through services instead. Still, the action may need some data from NSO, such as an IP address of a device, which you can access by using `trans` with the `ncs.maagic.get_root()` function and navigate to the relevant information.

Further details and the format of the arguments can be found in the NSO Python API reference.

The last thing to note in the above action code definition is the use of the decorator `@Action.action`. Its purpose is to set up the function arguments correctly, so variables such as `input` and `output` behave like other Python Maagic objects. This is no different from services, where decorators are required for the same reason.

Showcase: Implementing Device Count Action

Prerequisites:

- No previous NSO or netsim processes are running. Use the `ncs --stop` and `ncs-netsim stop` commands to stop them if necessary.
- NSO local install with a fresh runtime directory has been created by the `ncs-setup --dest ~/nso-lab-rundir` or similar command.
- The environment variable `NSO_RUNDIR` points to this runtime directory, such as set by the `export NSO_RUNDIR=~/nso-lab-rundir` command. It enables the below commands to work as-is, without additional substitution needed.

Step 1: Create a new Python package

One of the most common uses of NSO actions is automating network and service tests but they are also a good choice for any other non-configuration task. Being able to quickly answer questions, such as how

Step 2: Define a new action in YANG

many network ports are available (unused) or how many devices currently reside in a given subnet, can greatly simplify the network planning process. Coding these computations as actions in NSO makes them accessible on-demand to a wider audience.

For this scenario, you will create a new package for the action, however actions can also be placed into existing packages. A common example is adding a self-test action to a service package.

First, navigate to the packages subdirectory:

```
$ cd $NSO_RUNDIR/packages
```

Create a package skeleton with the **ncs-make-package** command and the **--action-example** option. Name the package “count-devices”, like so:

```
$ ncs-make-package --service-skeleton python --action-example count-devices
```

This command created a YANG module file, where you will place a custom action definition. In a text or code editor open the `count-devices.yang` file, located inside `count-devices/src/yang/`. This file already contains an example action which you will remove. Find the following line (after module imports):

```
description
```

Delete this line and all the lines following it, to the very end of the file. The file should now resemble the following:

```
module count-devices {

    namespace "http://example.com/count-devices";
    prefix count-devices;

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

Step 2: Define a new action in YANG

To model an action, you can use the `action` YANG statement. It is part of the YANG standard from version 1.1 onward, requiring you to also define `yang-version 1.1` in the YANG model. So, add the following line at the start of the module, right before `namespace` statement:

```
yang-version 1.1;
```

Note that in YANG version 1.0, actions used the NSO-specific `tailf:action` extension, which you may still find in some YANG models.

Now, go to the end of the file and add a `custom-actions` container with the `count-devices` action, using the `count-devices-action` action point. The input is an IP subnet and output the number of devices managed by NSO in this subnet.

```
container custom-actions {
    action count-devices {
        tailf:actionpoint count-devices-action;
        input {
            leaf in-subnet {
```

```

        type inet:ipv4-prefix;
    }
}
output {
    leaf result {
        type uint16;
    }
}
}
}
```

Also, add the closing bracket for module at the end:

```
}
```

Remember to finally save the file, which should now be similar to the following:

```

module count-devices {

    yang-version 1.1;
    namespace "http://example.com/count-devices";
    prefix count-devices;

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

    container custom-actions {
        action count-devices {
            tailf:actionpoint count-devices-action;
            input {
                leaf in-subnet {
                    type inet:ipv4-prefix;
                }
            }
            output {
                leaf result {
                    type uint16;
                }
            }
        }
    }
}
```

Step 3: Implement the action logic

The action code is implemented in a dedicated class, that you will put in a separate file. Using an editor, create a new, empty file `count_devices_action.py` in the `count-devices/python/count_devices/` subdirectory.

At the start of the file, import the packages that you will need later on and define the action class with the `cb_action()` method:

```
from ipaddress import IPv4Address, IPv4Network
import socket
import ncs
```

Step 4: Register callback

```
from ncs.dp import Action

class CountDevicesAction(Action):
    @Action.action
    def cb_action(self, uinfo, name, kp, input, output, trans):
```

Then initialize the count variable to 0 and construct a reference to the NSO data root, since it is not part of the method arguments:

```
count = 0
root = ncs.maagic.get_root(trans)
```

Using the `root` variable, you can iterate through the devices managed by NSO and find their (IPv4) address:

```
for device in root.devices.device:
    address = socket.gethostbyname(device.address)
```

If the IP address comes from the specified subnet, increment the count:

```
if IPv4Address(address) in IPv4Network(input.in_subnet):
    count = count + 1
```

Lastly, assign the count to the result:

```
output.result = count
```

Step 4: Register callback

Your custom Python code is ready; however, you still need to link it to the `count-devices` action. Open the `main.py` from the same directory in a text or code editor and delete all the content already in there.

Next, create a class called `Main` that inherits from the `ncs.application.Application` base class. Add a single class method `setup()` that takes no additional arguments.

```
import ncs

class Main(ncs.application.Application):
    def setup(self):
```

Inside the `setup()` method call the `register_action()` as follows:

```
self.register_action('count-devices-action', CountDevicesAction)
```

This line instructs NSO to use the `CountDevicesAction` class to handle invocations of the `count-devices-action` action point. Also import the `CountDevicesAction` class from the `count_devices_action` module.

The complete `main.py` file should then be similar to the following:

```
import ncs
from count_devices_action import CountDevicesAction

class Main(ncs.application.Application):
    def setup(self):
        self.register_action('count-devices-action', CountDevicesAction)
```

Step 5: And... action!

With all of the code ready, you are one step away from testing the new action, but to do that, you will need to add some devices to NSO. So, first add a couple of simulated routers to the NSO instance:

```
$ cd $NCS_DIR/examples.ncs/getting-started/developing-with-ncs/0-router-network
$ cp ncs-cdb/ncs_init.xml $NSO_RUNDIR/ncs-cdb/
$ cp -a packages/router $NSO_RUNDIR/packages/
```

Before the packages can be loaded, you must compile them:

```
$ cd $NSO_RUNDIR

$ make -C packages/router/src && make -C packages/count-devices/src
make: Entering directory 'packages/router/src'
< ... output omitted ... >
make: Leaving directory 'packages/router/src'
make: Entering directory 'packages/count-devices/src'
mkdir -p ..../load-dir
mkdir -p java/src/
bin/ncsc `ls count-devices-ann.yang > /dev/null 2>&1 && echo "-a count-devices-ann.yang" \
          -c -o ..../load-dir/count-devices.fxs yang/count-devices.yang
make: Leaving directory 'packages/count-devices/src'
```

You can start the NSO now and connect to the CLI:

```
$ ncs --with-package-reload && ncs_cli -C -u admin
```

Finally, invoke the action:

```
$ admin@ncs# custom-actions count-devices in-subnet 127.0.0.0/16
result 3
```

You can use the **show devices list** command to verify that the result is correct. You can alter the address of any device and see how it affects the result. You can even use a hostname, such as **localhost**.

Running Application Code

Services, actions, and other features all rely on callback registration. In Python code, the class responsible for registration derives from the `ncs.application.Application`. This allows NSO to manage the application code as appropriate, such as starting and stopping in response to NSO events. These events include package load or unload and NSO start or stop events.

While the Python package skeleton names the derived class `Main`, you can choose a different name if you also update the `package-meta-data.xml` file accordingly. This file defines a component with the name of the Python class to use:

```
<ncs-package xmlns="http://tail-f.com/ns/ncs-packages">
  < ... output omitted ... >

  <component>
    <name>main</name>
    <application>
      <python-class-name>dns_config.main.Main</python-class-name>
    </application>
  </component>
</ncs-package>
```

When starting the package, NSO reads the class name from `package-meta-data.xml`, starts the Python interpreter, and instantiates a class instance. The base `Application` class takes care of establishing communication with the NSO process and calling the `setup` and `teardown` methods. The two methods are a good place to do application-specific initialization and cleanup, along with any callback registrations you require.

The communication between the application process and NSO happens through a dedicated control socket, as described in the admin guide the section called “IPC ports” in *NSO 5.7 Administration Guide*. This setup prevents a faulty application to bring down the whole system along with it and enables NSO to support different application environments. In fact, NSO can manage applications written in Java or Erlang in addition to those in Python. If you replace the `python-class-name` element of a component with `java-class-name` in the `package-meta-data.xml` file, NSO will instead try to run the specified Java class in the managed Java VM. If you wanted to, you could implement all of the same services and actions in Java, too.

Regardless of the programming language you use, the high-level approach to automation with NSO does not change, registering and implementing callbacks as part of your network application. Of course, the actual function calls (the API) and other specifics differ for each language. [Chapter 6, The NSO Python VM](#), [Chapter 5, The NSO Java VM](#), and [Chapter 7, Embedded Erlang applications](#), cover the details. Even so, the concepts of actions, services, and YANG modeling remain the same. As you have seen, everything in NSO is ultimately tied to the YANG model, making YANG knowledge such a valuable skill for any NSO developer.



CHAPTER 5

The NSO Java VM

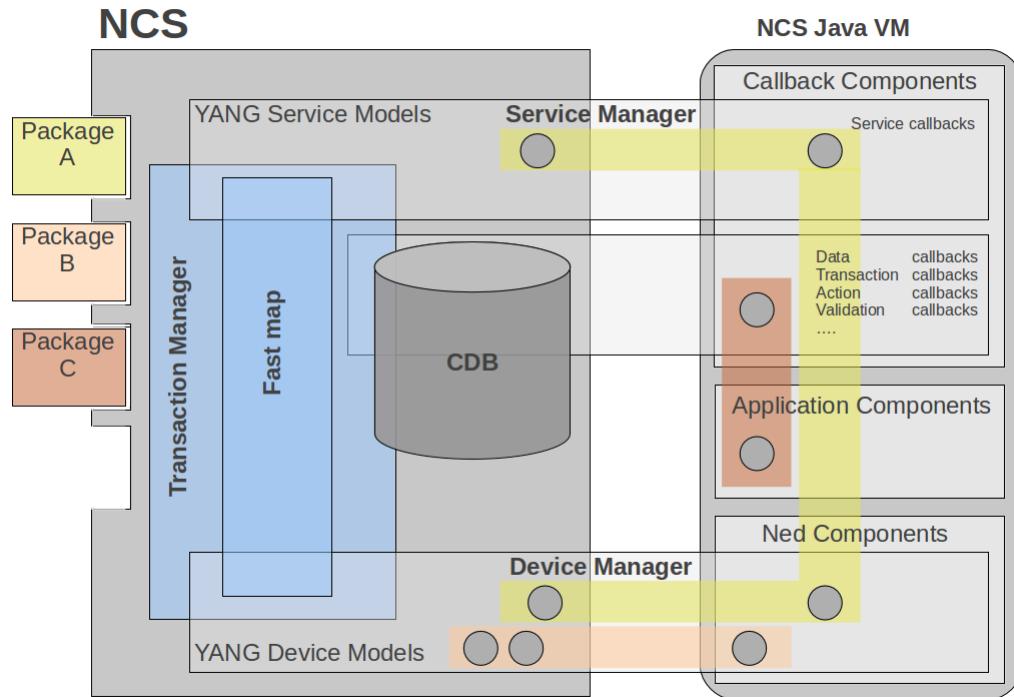
- [Overview, page 43](#)
- [YANG model, page 45](#)
- [Java Packages and the Class Loader, page 45](#)
- [The NED component type, page 47](#)
- [The callback component type, page 47](#)
- [The application component type, page 47](#)
- [The Resource Manager, page 48](#)
- [The Alarm centrals, page 50](#)
- [Embedding the NSO Java VM, page 50](#)
- [JMX interface, page 51](#)
- [Logging, page 54](#)
- [The NSO Java VM timeouts, page 54](#)
- [Debugging Startup, page 54](#)

Overview

The NSO Java VM is the execution container for all java classes supplied by deployed NSO packages. The classes, and other resources, are structured in jar files and the specific use of these classes are described in the *component* tag in respective package-meta-data.xml file. Also as a framework, it starts and control other utilities for the use of these components. To accomplish this a main class `com.tailf.ncs.NcsMain`, implementing the `Runnable` interface, is started as a thread. This thread can be the main thread (running in a java `main()`) or be embedded into another java program.

When the `NcsMain` thread starts it establishes a socket connection towards NSO. This is called the NSO Java VM control socket. It is the responsibility of `NcsMain` to respond to command requests from NSO and pass these commands as events to the underlying finite state machine (FSM). The `NcsMain` FSM will execute all actions as requested by NSO. This include class loading and instantiation as well as registration and start of services, NEDs etc.

Figure 11. NSO Service Manager



When NSO detects the control socket connect from NSO Java VM, it starts an initialization process.

- 1 First NSO sends a `INIT_JVM` request to the NSO Java VM. At this point the NSO Java VM will load schemas i.e. retrieve all known YANG module definitions. The NSO Java VM responds when all modules are loaded.
- 2 Then NSO sends a `LOAD_SHARED_JARS` request for each deployed NSO package. This request contains the URLs for the jars situated in the `shared-jar` directory in respective NSO package. The classes and resources in these jars will be globally accessible for all deployed NSO packages.
- 3 Next step is to send a `LOAD_PACKAGE` request for each deployed NSO package. This request contains the URLs for the jars situated in the `private-jar` directory in respective NSO package. These classes and resources will be private to respective NSO package. In addition, classes that are referenced in a `component` tag in respective NSO package `package-meta-data.xml` file will be instantiated.
- 4 NSO will send a `INSTANTIATE_COMPONENT` request for each component in each deployed NSO package. At this point the NSO Java VM will register a start method for respective component. NSO will send these requests in a proper start phase order. This implies that the `INSTANTIATE_COMPONENT` requests can be sent in an order that mixes components from different NSO packages.
- 5 Last, NSO sends a `DONE_LOADING` request which indicates that the initialization process is finished. After this the NSO Java VM is up and running.

See the section called “[Debugging Startup](#)” for tips on customizing startup behavior and debugging problems when the Java VM fails to start

YANG model

The file `tailf-ncs-java-vm.yang` defines the `java-vm` container which, along with `ncs.conf`, is the entry point for controlling the NSO Java VM functionality. Study the content of the YANG model in [Example 12, “The Java VM YANG model”](#). For a full explanation of all the configuration data, look at the YANG file and `man ncs.conf`.

Many of the nodes beneath `java-vm` are by default invisible due to a `hidden` attribute. In order to make everything beneath `java-vm` visible in the CLI, two steps are required. First the following XML snippet must be added to `ncs.conf`:

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

Now the `unhide` command may be used in the CLI session:

```
admin@ncs(config)# unhide debug
admin@ncs(config)#
```

Example 12. The Java VM YANG model

```
> yanger -f tree tailf-ncs-java-vm.yang
  submodule: tailf-ncs-java-vm (belongs-to tailf-ncs)
  +-rw java-vm
    +-rw stdout-capture
      | +-rw enabled? boolean
      | +-rw file? string
      | +-rw stdout? empty
      +-rw connect-time? uint32
      +-rw initialization-time? uint32
      +-rw synchronization-timeout-action? enumeration
      +-rw exception-error-message
      | +-rw verbosity? error-verbosity-type
      +-rw java-logging
        | +-rw logger* [logger-name]
          |   +-rw logger-name string
          |   +-rw level log-level-type
      +-rw jmx!
        | +-rw jndi-address? inet:ip-address
        | +-rw jndi-port? inet:port-number
        | +-rw jmx-address? inet:ip-address
        | +-rw jmx-port? inet:port-number
      +-ro start-status? enumeration
      +-ro status? enumeration
      +---x stop
        | +-ro output
        |   +-ro result? string
      +---x start
        | +-ro output
        |   +-ro result? string
      +---x restart
        +-ro output
        +-ro result? string
```

Java Packages and the Class Loader

Each NSO package will have a specific java classloader instance that loads its private jar classes. These package classloaders will refer to a single shared classloader instance as its parent. The shared classloader will load all shared jar classes for all deployed NSO packages.

**Note**

The jars in the `shared-jar` and `private-jar` directories should NOT be part of the java classpath

The purpose of this is first to keep integrity between packages which should not have access to each others classes, other than the ones that are contained in the shared jars. Secondly, this way it is possible to hot redeploy the private jars and classes of a specific package while keeping other packages in a run state.

Should this class loading scheme not be desired, it is possible to suppress it by starting the NSO Java VM with the system property `TAILF_CLASSLOADER` set to false.

```
java -DTAILF_CLASSLOADER=false ...
```

This will force NSO Java VM to use the standard java system classloader. For this to work, all jars from all deployed NSO packages needs to be part of the classpath. The drawback of this is that all classes will be globally accessible and hot redeploy will have no effect.

There are 4 types of components that the NSO Java VM can handle:

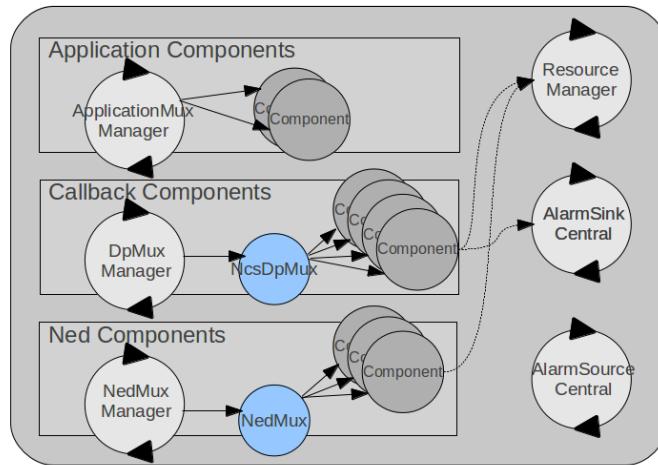
- The *ned* type. The NSO Java VM will handle NEDs of sub type *cli* and *generic* which are the ones that have a java implementation.
- The *callback* type. These are any forms of callbacks that defined by the Dp API.
- The *application* type. These are user defined daemons that implements a specific `ApplicationComponent` java interface.
- The *upgrade* type. This component type is activated when deploying a new version of a NSO package and the NSO automatic CDB data upgrade is not sufficient. See [the section called “Writing an Upgrade Package Component”](#) for more information.

In some situations several NSO packages are expected to use the same code base, e.g. when third party libraries are used or the code is structured with some common parts. Instead of duplicate jars in several NSO packages it is possible to create a new NSO package, add these jars to the `shared-jar` directory and let the `package-meta-data.xml` file contain no component definitions at all. The NSO Java VM will load these shared jars and these will be accessible from all other NSO packages.

Inside the NSO Java VM each component type have a specific Component Manager. The responsibility of these Managers are to manage a set of component classes for each NSO package. The Component Manager act as a FSM that controls when a component should be registered, started, stopped etc.

Figure 13. Component Managers

NCS Java VM



For instance the DpMuxManager controls all callback implementations (services, actions, data providers etc). It can load, register, start and stop such callback implementations.

The NED component type

NEDs can be of type *netconf*, *snmp*, *cli* or *generic*. Only the *cli* and *generic* types are relevant for the NSO Java VM because these are the ones that have a java implementation. Normally these NED components comes in self contained and prefabricated NSO packages for some equipment or class of equipment. It is however possible to tailor make NEDs for any protocol. For more information on this see Chapter 2, *Network Element Drivers (NEDs)* in *NSO 5.7 NED Development* and Chapter 5, *Writing a data model for a CLI NED* in *NSO 5.7 NED Development*

The callback component type

Callbacks are the collective name for a number of different functions that can be implemented in java. One of the most important is the service callbacks, but also actions, transaction control and data provision callbacks are in common use in an NSO implementation. For more on how to program callback using the Dp API. See [the section called “DP API”](#)

The application component type

For programs that are none of the above types but still need to access NSO as a daemon process it is possible to use the `ApplicationComponent` java interface. The `ApplicationComponent` interface expects the implementing classes to implement a `init()`, `finish()` and a `run()` method.

The NSO Java VM will start each class in a separate thread. The `init()` is called prior to the thread is started. The `run()` runs in a thread similar to the `run()` method in the standard java `Runnable` interface. The `finish()` method is called when the NSO Java VM wants the application thread to stop. It is the responsibility of the programmer to stop the application thread i.e stop the execution in the

`run()` method when `finish()` is called. Note, that making the thread stop when `finish()` is called is important so that the NSO Java VM will not be hanging at a `STOP_VM` request.

Example 14. ApplicationComponent Interface

```
package com.tailf.ncs;

/**
 * User defined Applications should implement this interface that
 * extends Runnable, hence also the run() method has to be implemented.
 * These applications are registered as components of type
 * "application" in a Ncs packages.
 *
 * Ncs Java VM will start this application in a separate thread.
 * The init() method is called before the thread is started.
 * The finish() method is expected to stop the thread. Hence stopping
 * the thread is user responsibility
 */
public interface ApplicationComponent extends Runnable {

    /**
     * This method is called by the Ncs Java vm before the
     * thread is started.
     */
    public void init();

    /**
     * This method is called by the Ncs Java vm when the thread
     * should be stopped. Stopping the thread is the responsibility of
     * this method.
     */
    public void finish();

}
```

An example of an application component implementation is found in [Chapter 17, SNMP Notification Receiver](#).

The Resource Manager

User Implementations typically needs resources like Maapi, Maapi Transaction, Cdb, Cdb Session etc. to fulfill their tasks. These resources can be instantiated and used directly in the user code. Which implies that the user code needs to handle connection and close of additional socket used by these resources. There is however another recommended alternative, and that is to use the Resource manager. The Resource manager is capable of injecting these resources into the user code. The principle is that the programmer will annotate the field that should refer to the resource rather than instantiate it.

Example 15. Resource injection

```
@Resource(type=ResourceType.MAAPI, scope=Scope.INSTANCE)
public Maapi m;
```

This way the NSO Java VM and the Resource manager can keep control over used resources and also have the possibility to intervene e.g. close sockets at forced shutdowns.

The Resource manager can handle two types of resources, Maapi and Cdb.

Example 16. Resource types

```
package com.tailf.ncs.annotations;

/**
 * ResourceType set by the Ncs ResourceManager
 */
public enum ResourceType {

    MAAPI(1),
    CDB(2);
}
```

For both the Maapi and Cdb resource types a socket connection is opened towards NSO by the Resource manager. At a stop the Resource manager will disconnects these sockets before ending the program. User programs can also tell the resource manager when its resources are no longer needed with a call to `ResourceManager.unregisterResources()`.

The resource annotation has three attributes:

- `type` defines the resource type.
- `scope` defines if this resource should be unique for each instance of the java class (`Scope.INSTANCE`) or shared between different instances and classes (`Scope.CONTEXT`). For `CONTEXT` scope the sharing is confined to the defining NSO package, i.e. a resource cannot be shared between NSO packages.
- `qualifier` is an optional string to identify the resource the unique resource. All instances that share the same context scoped resource needs to have the same qualifier. If the qualifier is not given it defaults to the value `DEFAULT` i.e shared between all instances that have the `DEFAULT` qualifier.

Example 17. Resource Annotation

```
package com.tailf.ncs.annotations;

/**
 * Annotation class for Action Callbacks Attributes are callPoint and callType
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Resource {

    public ResourceType type();

    public Scope scope();

    public String qualifier() default "DEFAULT";

}
```

Example 18. Scopes

```
package com.tailf.ncs.annotations;

/**
 * Scope for resources managed by the Resource Manager
```

```

    */
public enum Scope {

    /**
     * Context scope implies that the resource is
     * shared for all fields having the same qualifier in any class.
     * The resource is shared also between components in the package.
     * However sharing scope is confined to the package i.e sharing cannot
     * be extended between packages.
     * If the qualifier is not given it becomes "DEFAULT"
     */
    CONTEXT(1),
    /**
     * Instance scope implies that all instances will
     * get new resource instances. If the instance needs
     * several resources of the same type they need to have
     * separate qualifiers.
     */
    INSTANCE(2);
}

```

When the NSO Java VM starts it will receive component classes to load from NSO. Note, that the component classes are the classes that are referred to in the `package-meta-data.xml` file. For each component class the Resource Manager will scan for annotations and inject resources as specified.

However the package jars can contain lots of classes in addition to the component classes. These will be loaded at runtime and will be unknown by the NSO Java VM and therefore not handled automatically by the Resource Manager. These classes can also use resource injection but needs a specific call to the Resource Manager for the mechanism to take effect. Before the resources are used for the first time the resource should be used, a call of `ResourceManager.registerResources(. . .)` will force injection of the resources. If the same class is registered several times the Resource manager will detect this and avoid multiple resource injections.

Example 19. Force resource injection

```

MyClass myclass = new MyClass();
try {
    ResourceManager.registerResources(myclass);
} catch (Exception e) {
    LOGGER.error("Error injecting Resources", e);
}

```

The Alarm centrals

The `AlarmSourceCentral` and `AlarmSinkCentral` that is part of the NSO Alarm API can be used to simplify reading and writing alarms. The NSO Java VM will start these centrals at initialization. User implementations can therefore expect this to be set up without having to handle start and stop of either the `AlarmSinkCentral` or the `AlarmSourceCentral`. For more information on the alarm API see Chapter 6, *The Alarm Manager in NSO 5.7 User Guide*

Embedding the NSO Java VM

As stated above the NSO Java VM is executed in a thread implemented by the `NcsMain`. This implies that somewhere a java `main()` must be implemented that launches this thread. For NSO this is provided by the `NcsJVMLauncher` class. In addition to this there is a script named `ncs-start-java-vm` that

starts java with the `NcsJVMLauncher.main()`. This is the recommended way of launching the NSO Java VM and how it is set up in a default installation.

If there is a need to run the NSO Java VM as an embedded thread inside another program. This can be done as simply as instantiating the class `NcsMain` and start this instance in a new thread.

Example 20. Starting NcsMain

```
NcsMain ncsMain = NcsMain.getInstance(host);
Thread ncsThread = new Thread(ncsMain);

ncsThread.start();
```

However, with the embedding of the NSO Java VM comes the responsibility to manage the life-cycle of the NSO Java VM thread. This thread cannot be started before NSO has started and is running or else the NSO Java VM control socket connection will fail. Also, running NSO without the NSO Java VM being launched will render runtime errors as soon as NSO needs NSO Java VM functionality.

To be able to control an embedded NSO Java VM from another supervising java thread or program an optional JMX interface is provided. The main functionality in this interface is listing, starting and stopping the NSO Java VM and its Component Managers.

JMX interface

Normal control of the NSO Java engine is performed from NSO e.g. using the CLI. However `NcsMain` class and all component managers implements JMX interfaces to make it possible control the NSO Java VM also using standard Java tools like JvisualVM and JConsol.

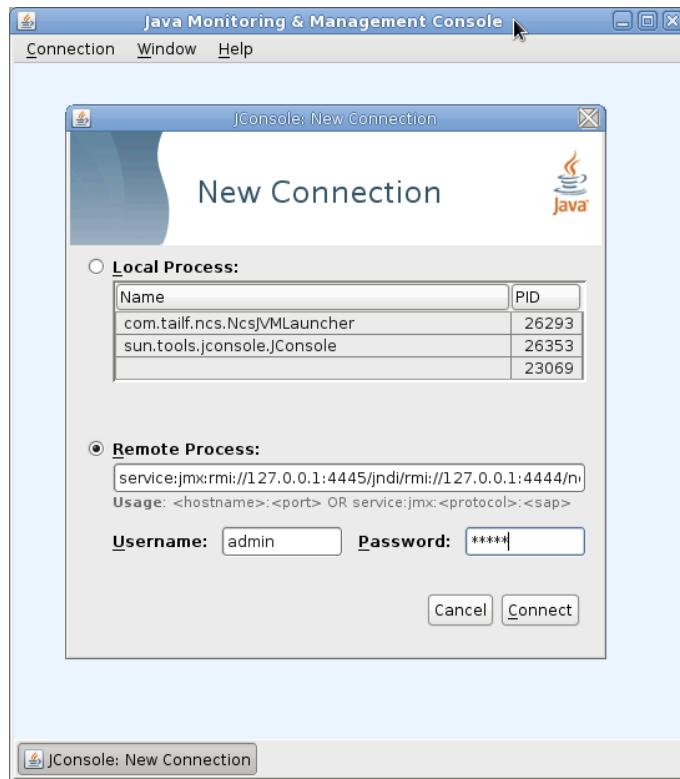
The JMX interface is configured via the `java-vm` YANG model (see `$NCS_DIR/src/ncs/yang/tailf-ncs-java-vm.yang`) in the NSO configuration. For JMX connection purposes there are four attributes to configure:

- `jmx-address` The hostname or IP for the RMI registry.
- `jmx-port` The port for the RMI registry.
- `jndi-address` The hostname or IP for the JMX RMI server.
- `jndi-port` The port for the JMX RMI server.

The JMX connection server uses two sockets for communication with a JMX client. The first socket is the JNDI RMI registry where the JMX Mbean objects are looked up. The second socket is the JMX RMI server from which the JMX connection objects are exported. For all practical purposes the host/ip for both sockets are the same and only the ports differ.

An example on JMX connection URL connecting to localhost is: `service:jmx:rmi://localhost:4445/jndi/rmi://localhost:4444/ncs`

In addition to the JMX URL the JMX user needs to authenticate using a legitimate user/password from the AAA configuration. An example on JMX authentication using the JConsol standard java tool is the following:

Figure 21. jconsole login window

The following JMX MBeans interfaces are defined:

Example 22. NcsMain JMX Bean

```
package com.tailf.ncs;
/**
 * This is the JMX interface for the NcsMain class
 */
public interface NcsMainMBean {

    /**
     * JMX interface - shutdown Ncs java vm main thread
     */
    public void shutdown();

    /**
     * JMX interface - hot redeploy all packages
     */
    public void redeployAll();

    /**
     * JMX interface - list shared jars
     */
    public String[] listSharedJars();
}
```

Example 23. NedMuxManager JMX Bean

```
package com.tailf.ncs.ctrl;
/**
```

```

 * This interface is the JMX interface for the NedMuxManager class
 */
public interface NedMuxManagerMBean {

    /**
     * JMX interface - list all Application components
     */
    public String[] listPackageComponents();
}

```

Example 24. DpMuxManager JMX Bean

```

package com.tailf.ncs.ctrl;
/**
 * This interface is the JMX interface for the DpMuxManager class
 */
public interface DpMuxManagerMBean {

    /**
     * JMX interface - list all callback components
     */
    public String[] listPackageComponents();
}

```

Example 25. ApplicationMuxManager JMX Bean

```

package com.tailf.ncs.ctrl;
/**
 * This interface is the JMX interface for the ApplicationMuxManager class
 */
public interface ApplicationMuxManagerMBean {

    /**
     * JMX interface - list all Application components
     */
    public String[] listPackageComponents();
}

```

Example 26. AlarmSinkCentral JMX Bean

```

package com.tailf.ncs.alarmman.producer;
/**
 * This is the JMX interface for the AlarmSinkCentral class
 */
public interface AlarmSinkCentralMBean {

    public void start();

    public boolean isAlive();

    public void stop();
}

```

Example 27. AlarmSourceCentral JMX Bean

```

package com.tailf.ncs.alarmman.consumer;
/**
 * This is the JMX interface for the AlarmSourceCentral class
 */
public interface AlarmSourceCentralMBean {

    public void start();
}

```

```

    public boolean isAlive();

    public void stop();
}

```

Logging

NSO has extensive logging functionality. Log settings are typically very different for a production system compared to a development system. Furthermore, the logging of the NSO daemon and the NSO Java VM is controlled by different mechanisms. During development, we typically want to turn on the developer-log. The sample ncs.conf that comes with the NSO release has log settings suitable for development, while the ncs.conf created by a "system install" are suitable for production deployment.

The NSO Java VM uses Log4j for logging and will read its default log settings from a provided log4j2.xml file in the ncs.jar. Following that, NSO itself has java-vm log settings that are directly controllable from the NSO CLI. We can do:

```
admin@ncs(config)# java-vm java-logging logger com.tailf.maapi level level-trace
admin@ncs(config-logger-com.tailf.maapi)# commit
Commit complete.
```

This will dynamically reconfigure the log level for package com.tailf.maapi to be at the level *trace*. Where the java logs actually end up is controlled by the log4j2.xml file. By default the NSO Java VM writes to stdout. If the NSO Java VM is started by NSO, as controlled by the ncs.conf parameter /java-vm/auto-start, NSO will pick up the stdout of the service manager and write it to:

```
admin@ncs(config)# show full-configuration java-vm stdout-capture
java-vm stdout-capture file /var/log/ncs/ncs-java-vm.log
```

(The "details" pipe command also displays default values)

The NSO Java VM timeouts

The section /ncs-config/japi in ncs.conf contains a number of very important timeouts. See \$NCS_DIR/src/ncs/ncs_config/tailf-ncs-config.yang and ncs.conf(5) in *NSO 5.7 Manual Pages* for details.

- *new-session-timeout* controls how long NSO will wait for the NSO Java VM to respond to a new session.
- *query-timeout* controls how long NSO will wait for the NSO Java VM to respond to a request to get data.
- *connect-timeout* controls how long NSO will wait for the NSO Java VM to initialize a Dp connection after the initial socket connect.

Whenever any of these timeouts trigger, NSO will close the sockets from NSO to the NSO Java VM. The NSO Java VM will detect the socket close and exit. If NSO is configured to start (and restart) the NSO Java VM, the NSO Java VM will be automatically restarted. If the NSO Java VM is started by some external entity, if it runs within an application server, it is up to that entity to restart NSO Java VM.

Debugging Startup

When using the auto-start feature (the default), NSO will start the NSO Java VM as outlined in the section called “Overview”, there are a number of different settings in the java-vm YANG model (see \$NCS_DIR/src/ncs/yang/tailf-ncs-java-vm.yang) that control what happens when something goes wrong during the startup.

The two timeout configurations `connect-time` and `initialization-time` are most relevant during startup. If the Java VM fails during the initial stages (during `INIT_JVM`, `LOAD_SHARED_JARS`, or `LOAD_PACKAGE`) either because of a timeout or because of a crash, NSO will log "The NCS Java VM synchronization failed" in `ncs.log`.

**Note**

The synchronization error message in the log will also have a hint as to what happened: "closed" usually means that the Java VM crashed (and closed the socket connected to NSO), "timeout" means that it failed to start (or respond) within the time limit. For example if the Java VM runs out of memory and crashes, this will be logged as "closed".

After logging NSO will take action based on the `synchronization-timeout-action` setting:

<code>log</code>	NSO will log the failure, and if <code>auto-restart</code> is set to true NSO will try to restart the Java VM
<code>log-stop (default)</code>	NSO will log the failure, and if the Java VM has not stopped already NSO will also try to stop it. No restart action is taken.
<code>exit</code>	NSO will log the failure, and then stop NSO itself.

If you have problems with the Java VM crashing during startup, a common pitfall is running out of memory (either total memory on the machine, or heap in the JVM). If you have a lot of Java code (or a loaded system) perhaps the Java VM did not start in time. Try to determine the root cause, check `ncs.log` and `ncs-java-vm.log`, and if needed increase the timeout.

For complex problems, for example with the class loader, try logging the internals of the startup:

```
admin@ncs(config)# java-vm java-logging logger com.tailf.ncs level level-all
admin@ncs(config-logger-com.tailf.maapi)# commit
Commit complete.
```

Setting this will result in a lot more detailed information in `ncs-java-vm.log` during startup.

When the `auto-restart` setting is `true` (the default) it means that NSO will try to restart the Java VM when it fails (at any point in time, not just during startup). NSO will at most try three restarts within 30 seconds, i.e. if the Java VM crashes more than three times within 30 seconds NSO gives up. You can check the status of the Java VM using the `java-vm` YANG model. For example in the CLI:

```
admin@ncs# show java-vm
java-vm start-status started
java-vm status running
```

The `start-status` can have the following values:

<code>auto-start-not-enabled</code>	Auto start is not enabled.
<code>stopped</code>	The Java VM has been stopped or is not yet started.
<code>started</code>	The Java VM has been started. See the leaf 'status' to check the status of the Java application code.
<code>failed</code>	The Java VM has terminated. If 'auto-restart' is enabled, the Java VM restart has been disabled due to too many frequent restarts.

The `status` can have the following values:

<code>not-connected</code>	The Java application code is not connected to NSO.
<code>initializing</code>	The Java application code is connected to NSO, but not yet initialized.

running
timeout

The Java application code is connected and initialized.

The Java application connected to NSO, but failed to initialize within the stipulated timeout 'initialization-time'.



CHAPTER 6

The NSO Python VM

- [Introduction, page 57](#)
- [YANG model, page 57](#)
- [Structure of the User provided code, page 59](#)
- [Debugging of Python packages, page 62](#)
- [Using non-standard Python, page 63](#)

Introduction

NSO is capable of starting one or several Python VMs where Python code in user provided packages can run.

An NSO package containing a `python` directory will be considered to be a *Python Package*. By default, a Python VM will be started for each Python package that has got a *python-class-name* defined in its `package-meta-data.xml` file. In this Python VM the `PYTHONPATH` environment variable will be pointing to the `python` directory in the package.

If any *required-package* that is listed in the `package-meta-data.xml` contains a `python` directory, the PATH to that directory will be added to the `PYTHONPATH` of the started Python VM and thus its accompanying Python code will be accessible.

Several Python packages can be started in the same Python VM if their corresponding `package-meta-data.xml` files contains the same `python-package/vm-name`.

A Python package skeleton can be created by making use of the `ncs-make-package` command:

```
ncs-make-package --service-skeleton python <package-name>
```

YANG model

The `tailf-ncs-python-vm.yang` defines the `python-vm` container which, along with `ncs.conf`, is the entry point for controlling the NSO Python VM functionality. Study the content of the YANG model in [Example 28, “The Python VM YANG model”](#). For a full explanation of all the configuration data, look at the YANG file and man `ncs.conf`. Here will follow a description of the most important configuration parameters.

Note that some of the nodes beneath `python-vm` are by default invisible due to a `hidden` attribute. In order to make everything beneath `python-vm` visible in the CLI, two steps are required. First the following XML snippet must be added to `ncs.conf`:

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

Now the `unhide` command may be used in the CLI session:

```
admin@ncs(config)# unhide debug
admin@ncs(config)#
```

With the *logging/level* the amount of logged information can be controlled. This is a global setting applied to all started Python VMs unless explicitly set for a particular VM, see [the section called “Debugging of Python packages”](#). The levels corresponds to the pre-defined Python levels in the Python *logging* module, ranging from *level-critical* to *level-debug*.



Note

Refer to the official Python documentation for the *logging* module for more information about the log levels.

The *logging/log-file-prefix* define the prefix part of the log file path used for the Python VMs. This prefix will be appended with a Python VM specific suffix which is based on the Python package name or the `python-package/vm-name` from the `package-meta-data.xml` file. The default prefix is `logs/ncs-python-vm` so e.g. if a Python package named `l3vpn` is started, a logfile with the name `logs/ncs-python-vm-l3vpn.log` will be created.

The *status/start* and *status/current* contains operational data. The *status/start* command will show information about what Python classes, as declared in the `package-meta-data.xml` file, that where started and whether the outcome was successful or not. The *status/current* command will show which Python classes that are currently running in a separate thread. The latter assume that the user provided code cooperate by informing NSO about any thread(s) started by the user code, see [the section called “Structure of the User provided code”](#).

The *start* and *stop* actions makes it possible to start and stop a particular Python VM.

Example 28. The Python VM YANG model

```
> yanger -f tree tailf-ncs-python-vm.yang
submodule: tailf-ncs-python-vm (belongs-to tailf-ncs)
  +-rw python-vm
    +-rw logging
      +-rw log-file-prefix? string
      +-rw level?          py-log-level-type
      +-rw vm-levels* [node-id]
        +-rw node-id      string
        +-rw level       py-log-level-type
    +-rw status
      +-ro start* [node-id]
      | +-ro node-id      string
      | +-ro packages* [package-name]
      |   +-ro package-name string
      |   +-ro components* [component-name]
      |     +-ro component-name string
      |     +-ro class-name?   string
      |     +-ro status?       enumeration
```

```

|           +-+ro error-info?      string
+-+ro current* [node-id]
|   +-+ro node-id      string
|   +-+ro packages* [package-name]
|     +-+ro package-name    string
|     +-+ro components* [component-name]
|       +-+ro component-name  string
|       +-+ro class-names* [class-name]
|         +-+ro class-name    string
|         +-+ro status?      enumeration
+---x stop
|   +---w input
|   |   +---w name?      string
|   +-+ro output
|     +-+ro result?      string
+---x start
|   +---w input
|   |   +---w name?      string
|   +-+ro output
|     +-+ro result?      string

```

Structure of the User provided code

The `package-meta-data.xml` file must contain a *component* of type *application* with a *python-class-name* specified as shown in the example below.

Example 29. `package-meta-data.xml` excerpt

```

<component>
  <name>L3VPN Service</name>
  <application>
    <python-class-name>l3vpn.service.Service</python-class-name>
  </application>
</component>
<component>
  <name>L3VPN Service model upgrade</name>
  <upgrade>
    <python-class-name>l3vpn.upgrade.Upgrade</python-class-name>
  </upgrade>
</component>

```

The component name (*L3VPN Service* in the example) is a human readable name of this application component. It will be shown when doing `show python-vm` in the CLI. The *python-class-name* should specify the Python class that implements the application entry point. Note that it needs to be specified using Python's dot-notation and should be fully qualified (given the fact that `PYTHONPATH` is pointing to the package `python` directory).

Study the excerpt of the directory listing from a package named *l3vpn* below.

Example 30. Python package directory structure

```

packages/
+-- l3vpn/
    --- package-meta-data.xml
    --- python/
        |   +-- l3vpn/
        |   |   +-- __init__.py
        |   |   +-- service.py
        |   |   +-- upgrade.py
        |   |   +-- _namespaces/

```

```

|           +-+ __init__.py
|           +-+ l3vpn_ns.py
+-+ src
    +-+ Makefile
    +-+ yang/
        +-+ l3vpn.yang

```

Look closely at the `python` directory above. Note that directly under this directory is another directory named as the package (`l3vpn`) that contains the user code. This is an important structural choice which eliminates the chance of code clashes between dependent packages (only if all dependent packages uses this pattern of course).

As you can see, the `service.py` is located according to the description above. There is also a `__init__.py` (which is empty) there to make the `l3vpn` directory considered a *module* from Python's perspective.

Note the `_namespaces/l3vpn_ns.py` file. It is generated from the `l3vpn.yang` model using the `ncsc --emit-python` command and contains constants representing the namespace and the various components of the YANG model, which the User code can import and make use of.

The `service.py` file should include a class definition named *Service* which acts as the component's entry point. See [the section called “The application component”](#) for details.

Notice that there is also a file named `upgrade.py` present which hold the implementation of the *upgrade* component specified in the `package-meta-data.xml` excerpt above. See [the section called “The upgrade component”](#) for details regarding *upgrade* components.

The application component

The Python class specified in the `package-meta-data.xml` file will be started in a Python thread which we call a *component thread*. This Python class should inherit `ncs.application.Application` and should implement the methods `setup()` and `teardown()`.

NSO supports two different modes for executing the implementations of the registered callpoints, *threading* and *multiprocessing*.

threading mode will use a single thread pool for executing the callbacks for all callpoints.

multiprocessing mode will start a subprocess for each callpoint. Depending on the user code this can greatly improve the performance on systems with a lot of parallel requests.

The behavior is controlled by three factors:

- *callpoint-model* setting in the `package-meta-data.xml` file.
- Number of registered callpoints in the *Application*.
- Operating System support for killing child processes when the parent exits.

If the *callpoint-model* is set to *multiprocessing*, more than one callpoint is registered in the *Application* and the Operating System supports killing child processes when the parent exits, NSO will enable multiprocessing mode.

Example 31. Component class skeleton

```

import ncs

class Service(ncs.application.Application):
    def setup(self):

```

```

# The application class sets up logging for us. It is accessible
# through 'self.log' and is a ncs.log.Log instance.
self.log.info('Service RUNNING')

# Service callbacks require a registration for a 'service point',
# as specified in the corresponding data model.
#
self.register_service('13vpn-servicepoint', ServiceCallbacks)

# If we registered any callback(s) above, the Application class
# took care of creating a daemon (related to the service/action point).

# When this setup method is finished, all registrations are
# considered done and the application is 'started'.

def teardown(self):
    # When the application is finished (which would happen if NCS went
    # down, packages were reloaded or some error occurred) this teardown
    # method will be called.

    self.log.info('Service FINISHED')

```

The *Service* class will be instantiated by NSO when started or whenever packages are reloaded. Custom initialization such as registering service- and action callbacks should be done in the *setup()* method. If any cleanup is needed when NSO finishes or when packages are reloaded it should be placed in the *teardown()* method.

The existing log functions are named after the standard Python log levels, thus in the example above the *self.log* object contains the functions *debug,info,warning,error,critical*. Where to log and with what level can be controlled from NSO.

The upgrade component

The Python class specified in the *upgrade* section of *package-meta-data.xml* will be run by NSO in a separately started Python VM. The class must be instantiable using the empty constructor and it must have a method called *upgrade* as in the example below. It should inherit *ncs.upgrade.Upgrade*.

Example 32. Upgrade class example

```

import ncs
import _ncs

class Upgrade(ncs.upgrade.Upgrade):
    """An upgrade 'class' that will be instantiated by NSO.

    This class can be named anything as long as NSO can find it using the
    information specified in <python-class-name> for the <upgrade>
    component in package-meta-data.xml.

    Is should inherit ncs.upgrade.Upgrade.

    NSO will instantiate this class using the empty constructor.
    The class MUST have a method named 'upgrade' (as in the example below)
    which will be called by NSO.
    """

    def upgrade(self, cdbsock, trans):
        """The upgrade 'method' that will be called by NSO.

```

```

Arguments:
cdbsock -- a connected CDB data socket for reading current (old) data.
trans -- a ncs.maapi.Transaction instance connected to the init
transaction for writing (new) data.

There is no need to connect a CDB data socket to NSO - that part is
already taken care of and the socket is passed in the first argument
'cdbsock'. A session against the DB needs to be started though. The
session doesn't need to be ended and the socket doesn't need to be
closed - NSO will do that automatically.

The second argument 'trans' is already attached to the init transaction
and ready to be used for writing the changes. It can be used to create a
magic object if that is preferred. There's no need to detach or finish
the transaction, and, remember to NOT apply() the transaction when work
is finished.

The method should return True (or None, which means that a return
statement is not needed) if everything was OK.
If something went wrong the method should return False or throw an
error. The northbound client initiating the upgrade will be alerted
with an error message.

Anything written to stdout/stderr will end up in the general log file
for spurious output from Python VMs. If not configured the file will
be named ncs-python-vm.log.

"""

# start a session against running
_ncs.cdb.start_session2(cdbsock, ncs.cdb.RUNNING,
                        ncs.cdb.LOCK_SESSION | ncs.cdb.LOCK_WAIT)

# loop over a list and do some work
num = _ncs.cdb.num_instances(cdbsock, '/path/to/list')
for i in range(0, num):
    # read the key (which in this example is 'name') as a ncs.Value
    value = _ncs.cdb.get(cdbsock, '/path/to/list[{0}]/name'.format(i))
    # create a mandatory leaf 'level' (enum - low, normal, high)
    key = str(value)
    trans.set_elem('normal', '/path/to/list{{0}}/level'.format(key))

# not really needed
return True

# Error return example:
#
# This indicates a failure and the string written to stdout below will
# written to the general log file for spurious output from Python VMs.
#
# print('Error: not implemented yet')
# return False

```

Debugging of Python packages

Python code packages are not running with an attached console and the standard out from the Python VMs are collected and put into the common log file ncs-python-vm.log. Possible python compilation errors will also end up in this file.

Normally the logging objects provided by the Python APIs is used. They are based on the standard Python logging module. This gives the possibility to control the logging if needed, e.g. getting a module local logger to increase logging granularity.

The default logging level is set to *info*. For debugging purposes it is very useful to increase the logging level:

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

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

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

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

Log file output example for package `l3vpn`:

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

Using non-standard Python

There are occasions where the standard Python installation is incompatible or maybe not preferred to be used together with NSO. In such cases there are several options to tell NSO to use another Python installation for starting a Python VM.

By default NSO will use the file `$NCS_DIR/bin/ncs-start-python-vm` when starting a new Python VM. The last few lines in that file reads:

```
if [ -x "$(which python3)" ]; then
    echo "Starting python3 -u $main $*"
    exec python3 -u "$main" "$@"
fi
echo "Starting python -u $main $*"
exec python -u "$main" "$@"
```

As seen above NSO first looks for `python3` and if found it will be used to start the VM. If `python3` is not found NSO will try to use the command `python` instead. Here we describe a couple of options for deciding which Python NSO should start.

Configure NSO to use a custom start command (recommended)

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

Example:

```
$ cd $NCS_DIR/bin
$ pwd
```

Changing the path to *python3* or *python*

```
/usr/local/ns0/bin  
$ cp ncs-start-python-vm my-start-python-vm  
$ # Use your favourite editor to update the last lines of the new  
$ # file to start the desired Python executable.
```

And add the following snippet to ncs.conf:

```
<python-vm>  
  <start-command>/usr/local/ns0/bin/my-start-python-vm</start-command>  
</python-vm>
```

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

Changing the path to *python3* or *python*

Another way of telling NSO to start a specific Python executable is to configure the environment so that executing *python3* or *python* starts the desired Python. This may be done system wide or can be made specific for the user running NSO.

Updating the default start command (not recommended)

Changing the last line of \$NCS_DIR/bin/ncs-start-python-vm is of course an option but altering any of the installation files of NSO is discouraged.



CHAPTER 7

Embedded Erlang applications

- [Introduction, page 65](#)
- [Erlang API, page 65](#)
- [Application configuration, page 66](#)
- [Example, page 67](#)

Introduction

NSO is capable of starting user provided Erlang applications embedded in the same Erlang VM as NSO.

The Erlang code is packaged into applications which are automatically started and stopped by NSO if they are located at the proper place. NSO will search all packages for top level directories called `erlang-lib`. The structure of such a directory is the same as a standard `lib` directory in Erlang. The directory may contain multiple Erlang applications. Each one must have a valid `.app` file. See the Erlang documentation of `application` and `app` for more info.

An Erlang package skeleton can be created by making use of the `ncs-make-package` command:

```
ncs-make-package --erlang-skeleton --erlang-application-name <appname> <package-name>
```

Multiple applications can be generated by using the option `--erlang-application-name NAME` multiple times with different names.

All application code **SHOULD** use the prefix `"ec_"` for module names, application names, registered processes (if any), and named ets tables (if any), to avoid conflict with existing or future names used by NSO itself.

Erlang API

The Erlang API to NSO is implemented as an Erlang/OTP application called `econfd`. This application comes in two flavours. One is builtin into NSO in order to support applications running in the same Erlang VM as NSO. The other is a separate library which is included in source form in the NSO release, in the `$NCS_DIR/erlang` directory. Building `econfd` as described in the `$NCS_DIR/erlang/econfd/README` file will compile the Erlang code and generate the documentation.

This API can be used by applications written in Erlang in much the same way as the C and Java APIs are used, i.e. code running in an Erlang VM can use the `econfd` API functions to make socket connections to NSO for data provider, MAAPI, CDB, etc access. However the API is also available internally in NSO,

which makes it possible to run Erlang application code inside the NSO daemon, without the overhead imposed by the socket communication.

When the application is started, one of its processes should make initial connections to the NSO subsystems, register callbacks etc. This is typically done in the `init/1` function of a `gen_server` or similar. While the internal connections are made using the exact same API functions (e.g. `econfd_maapi:connect/2`) as for an application running in an external Erlang VM, any `Address` and `Port` arguments are ignored, and instead standard Erlang inter-process communication is used.

There is little or no support for testing and debugging Erlang code executing internally in NSO, since NSO provides a very limited runtime environment for Erlang in order to minimize disk and memory footprints. Thus the recommended method is to develop Erlang code targeted for this by using `econfd` in a separate Erlang VM, where an interactive Erlang shell and all the other development support included in the standard Erlang/OTP releases are available. When development and testing is completed, the code can be deployed to run internally in NSO without changes.

For information about the Erlang programming language and development tools, please refer to www.erlang.org and the available books about Erlang (some are referenced on the web site).

The `--printlog` option to `ncs`, which prints the contents of the NSO error log, is normally only useful for Cisco support and developers, but it may also be relevant for debugging problems with application code running inside NSO. The error log collects the events sent to the OTP error_logger, e.g. crash reports as well as info generated by calls to functions in the `error_logger(3)` module. Another possibility for primitive debugging is to run `ncs` with the `--foreground` option, where calls to `io:format/2` etc will print to standard output. Printouts may also be directed to the developer log by using `econfd:log/3`.

While Erlang application code running in an external Erlang VM can use basically any version of Erlang/OTP, this is not the case for code running inside NSO, since the Erlang VM is evolving and provides limited backward/forward compatibility. To avoid incompatibility issues when loading the beam files, the Erlang compiler `erlc` should be of the same version as was used to build the NSO distribution.

NSO provides the VM, `erlc` and the kernel, `stdlib`, and `crypto` OTP applications.



Note

Obviously application code running internally in the NSO daemon can have an impact on the execution of the standard NSO code. Thus it is critically important that the application code is thoroughly tested and verified before being deployed for production in a system using NSO.

Application configuration

Applications may have dependencies to other applications. These dependencies affects the start order. If the dependent application resides in another package this should be expressed by using `required-package` in the `package-meta-data.xml` file. Application dependencies within the same package should be expressed in the `.app`. See below.

The following config settings in the `.app` file are explicitly treated by NSO:

<code>applications</code>	A list of applications which needs to be started before this application can be started. This info is used to compute a valid start order.
<code>included_applications</code>	A list of applications which are started on behalf of this application. This info is used to compute a valid start order.
<code>env</code>	A property list, containing <code>[{Key, Val}]</code> tuples. Besides other keys, used by the application itself, a few predefined keys are used by

NSO. The key `ncs_start_phase` is used by NSO to determine which start phase the application is to be started in. Valid values are `early_phase0`, `phase0`, `phase1`, `phase1_delayed` and `phase2`. Default is `phase1`. If the application is not required in the early phases of startup, set `ncs_start_phase` to `phase2` to avoid issues with NSO services being unavailable to the application. The key `ncs_restart_type` is used by NSO to determine which impact a restart of the application will have. This is the same as the `restart_type()` type in `application`. Valid values are `permanent`, `transient` and `temporary`. Default is `temporary`.

Example

The `examples.ncs/getting-started/developing-with-ncs/18-simple-service-erlang` example in the bundled collection shows how to create a service written in Erlang and executing it internally in NSO. This Erlang example is a subset of the Java example `examples.ncs/getting-started/developing-with-ncs/4-rfs-service`.

Example



CHAPTER 8

The YANG Data Modeling Language

- [The YANG Data Modeling Language, page 69](#)
- [YANG in NSO, page 69](#)
- [YANG Introduction, page 70](#)
- [Working With YANG Modules, page 78](#)
- [Integrity Constraints, page 79](#)
- [The when statement, page 81](#)
- [Using the Tail-f Extensions with YANG, page 81](#)
- [Custom Help Texts and Error Messages, page 83](#)
- [An Example: Modeling a List of Interfaces, page 84](#)
- [More on leafrefs, page 92](#)
- [Using Multiple Namespaces, page 94](#)
- [Module Names, Namespaces and Revisions, page 95](#)
- [Hash Values and the id-value Statement, page 96](#)
- [NSO caveats, page 96](#)

The YANG Data Modeling Language

YANG is a data modeling language used to model configuration and state data manipulated by a NETCONF agent. The YANG modeling language is defined in RFC 6020 (version 1) and RFC 7950 (version 1.1). YANG as a language will not be described in its entirety here - rather we refer to the IETF RFC text at <https://www.ietf.org/rfc/rfc6020.txt> and <https://www.ietf.org/rfc/rfc7950.txt>.

Another source of information regarding the YANG language is the wiki based web site <http://www.yang-central.org/>. For a tutorial on the data modeling capabilities of YANG, see <http://www.yang-central.org/twiki/bin/view/Main/DhcpTutorial>.

YANG in NSO

In NSO, YANG is not only used for NETCONF data. On the contrary, YANG is used to describe the data model as a whole and used by all northbound interfaces.

NSO uses YANG for Service Models as well as for specifying device interfaces. Where do these models come from? When it comes to services, the YANG service model is specified as part of the service design activity. NSO ships several examples of service models that can be used as a starting point. For devices, it depends on the underlying device interface how the YANG model is derived. For native NETCONF/YANG devices the YANG model is of course given by the device. For SNMP devices, the NSO tool-chain generates the corresponding YANG modules, (SNMP NED). For CLI devices, the package for the device contains the YANG data-model. This is shipped in text and can be modified to cater for upgrades. Customers can also write their own YANG data-models to render the CLI integration (CLI NED). The situation for other interfaces are similar to CLI, a YANG model that corresponds to the device interface data model is written and bundled in the NED package.

NSO also relies on the revision statement in YANG modules for revision management of different versions of the same type of managed device, but running different software versions.

A YANG module can be directly transformed into a final schema (.fxs) file that can be loaded into NSO. Currently all features of the YANG language except the `anyxml` statement are supported.

The data-models including the .fxs file along with any code are bundled into packages that can be loaded to NSO. This is true for service applications as well as for NEDs and other packages. The corresponding YANG can be found in the `src/yang` directory in the package.

YANG Introduction

This section is a brief introduction to YANG. The exact details of all language constructs is fully described in RFC 6020 and RFC 7950.

The NSO programmer must know YANG well, since all APIs use various paths that are derived from the YANG datamodel.

Modules and Submodules

A module contains three types of statements: module-header statements, revision statements, and definition statements. The module header statements describe the module and give information about the module itself, the revision statements give information about the history of the module, and the definition statements are the body of the module where the data model is defined.

A module may be divided into submodules, based on the needs of the module owner. The external view remains that of a single module, regardless of the presence or size of its submodules.

The `include` statement allows a module or submodule to reference material in submodules, and the `import` statement allows references to material defined in other modules.

Data Modeling Basics

YANG defines four types of nodes for data modeling. In each of the following subsections, the example shows the YANG syntax as well as a corresponding NETCONF XML representation.

Leaf Nodes

A leaf node contains simple data like an integer or a string. It has exactly one value of a particular type, and no child nodes.

```
leaf host-name {
    type string;
    description "Hostname for this system";
}
```

With XML value representation for example as:

```
<host-name>my.example.com</host-name>
```

An interesting variant of leaf nodes are typeless leafs.

```
leaf enabled {
    type empty;
    description "Enable the interface";
}
```

With XML value representation for example as:

```
<enabled/>
```

Leaf-list Nodes

A **leaf-list** is a sequence of leaf nodes with exactly one value of a particular type per leaf.

```
leaf-list domain-search {
    type string;
    description "List of domain names to search";
}
```

With XML value representation for example as:

```
<domain-search>high.example.com</domain-search>
<domain-search>low.example.com</domain-search>
<domain-search>everywhere.example.com</domain-search>
```

Container Nodes

A container node is used to group related nodes in a subtree. It has only child nodes and no value and may contain any number of child nodes of any type (including leafs, lists, containers, and leaf-lists).

```
container system {
    container login {
        leaf message {
            type string;
            description
                "Message given at start of login session";
        }
    }
}
```

With XML value representation for example as:

```
<system>
    <login>
        <message>Good morning, Dave</message>
    </login>
</system>
```

List Nodes

A **list** defines a sequence of list entries. Each entry is like a structure or a record instance, and is uniquely identified by the values of its key leafs. A list can define multiple keys and may contain any number of child nodes of any type (including leafs, lists, containers etc.).

```
list user {
    key "name";
    leaf name {
        type string;
```

```

    }
    leaf full-name {
        type string;
    }
    leaf class {
        type string;
    }
}

```

With XML value representation for example as:

```

<user>
  <name>glocks</name>
  <full-name>Goldie Locks</full-name>
  <class>intruder</class>
</user>
<user>
  <name>snowey</name>
  <full-name>Snow White</full-name>
  <class>free-loader</class>
</user>
<user>
  <name>rzull</name>
  <full-name>Repun Zell</full-name>
  <class>tower</class>
</user>

```

Example Module

These statements are combined to define the module:

```

// Contents of "acme-system.yang"
module acme-system {
    namespace "http://acme.example.com/system";
    prefix "acme";

    organization "ACME Inc.";
    contact "joe@acme.example.com";
    description
        "The module for entities implementing the ACME system.";

    revision 2007-06-09 {
        description "Initial revision.";
    }

    container system {
        leaf host-name {
            type string;
            description "Hostname for this system";
        }

        leaf-list domain-search {
            type string;
            description "List of domain names to search";
        }

        container login {
            leaf message {
                type string;
                description
                    "Message given at start of login session";
            }
        }
    }
}
```

```
list user {
    key "name";
    leaf name {
        type string;
    }
    leaf full-name {
        type string;
    }
    leaf class {
        type string;
    }
}
```

State Data

YANG can model state data, as well as configuration data, based on the `config` statement. When a node is tagged with `config false`, its sub hierarchy is flagged as state data, to be reported using NETCONF's `get` operation, not the `get-config` operation. Parent containers, lists, and key leafs are reported also, giving the context for the state data.

In this example, two leafs are defined for each interface, a configured speed and an observed speed. The observed speed is not configuration, so it can be returned with NETCONF **get** operations, but not with **get-config** operations. The observed speed is not configuration data, and cannot be manipulated using **edit-config**.

```
list interface {
    key "name";
    config true;

    leaf name {
        type string;
    }
    leaf speed {
        type enumeration {
            enum 10m;
            enum 100m;
            enum auto;
        }
    }
    leaf observed-speed {
        type uint32;
        config false;
    }
}
```

Built-in Types

YANG has a set of built-in types, similar to those of many programming languages, but with some differences due to special requirements from the management domain. The following table summarizes the built-in types.

Table 33. YANG built-in types

Name	Type	Description
binary	Text	Any binary data
bits	Text/Number	A set of bits or flags

Name	Type	Description
boolean	Text	"true" or "false"
decimal64	Number	64-bit fixed point real number
empty	Empty	A leaf that does not have any value
enumeration	Text/Number	Enumerated strings with associated numeric values
identityref	Text	A reference to an abstract identity
instance-identifier	Text	References a data tree node
int8	Number	8-bit signed integer
int16	Number	16-bit signed integer
int32	Number	32-bit signed integer
int64	Number	64-bit signed integer
leafref	Text/Number	A reference to a leaf instance
string	Text	Human readable string
uint8	Number	8-bit unsigned integer
uint16	Number	16-bit unsigned integer
uint32	Number	32-bit unsigned integer
uint64	Number	64-bit unsigned integer
union	Text/Number	Choice of member types

Derived Types (typedef)

YANG can define derived types from base types using the `typedef` statement. A base type can be either a built-in type or a derived type, allowing a hierarchy of derived types. A derived type can be used as the argument for the `type` statement.

```
typedef percent {
    type uint16 {
        range "0 .. 100";
    }
    description "Percentage";
}

leaf completed {
    type percent;
}
```

With XML value representation for example as:

```
<completed>20</completed>
```

User defined typedefs are useful when we want to name and reuse a type several times. It is also possible to restrict leafs inline in the data model as in:

```
leaf completed {
    type uint16 {
        range "0 .. 100";
    }
    description "Percentage";
}
```

Reusable Node Groups (grouping)

Groups of nodes can be assembled into the equivalent of complex types using the grouping statement. grouping defines a set of nodes that are instantiated with the uses statement:

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

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

With XML value representation for example as:

```
<peer>
  <destination>
    <address>192.0.2.1</address>
    <port>830</port>
  </destination>
</peer>
```

The grouping can be refined as it is used, allowing certain statements to be overridden. In this example, the description is refined:

```
container connection {
    container source {
        uses target {
            refine "address" {
                description "Source IP address";
            }
            refine "port" {
                description "Source port number";
            }
        }
    }
    container destination {
        uses target {
            refine "address" {
                description "Destination IP address";
            }
            refine "port" {
                description "Destination port number";
            }
        }
    }
}
```

Choices

YANG allows the data model to segregate incompatible nodes into distinct choices using the choice and case statements. The choice statement contains a set of case statements which define sets of schema

nodes that cannot appear together. Each `case` may contain multiple nodes, but each node may appear in only one `case` under a `choice`.

When the nodes from one case are created, all nodes from all other cases are implicitly deleted. The device handles the enforcement of the constraint, preventing incompatibilities from existing in the configuration.

The choice and case nodes appear only in the schema tree, not in the data tree or XML encoding. The additional levels of hierarchy are not needed beyond the conceptual schema.

```
container food {
    choice snack {
        mandatory true;
        case sports-arena {
            leaf pretzel {
                type empty;
            }
            leaf beer {
                type empty;
            }
        }
        case late-night {
            leaf chocolate {
                type enumeration {
                    enum dark;
                    enum milk;
                    enum first-available;
                }
            }
        }
    }
}
```

With XML value representation for example as:

```
<food>
  <chocolate>first-available</chocolate>
</food>
```

Extending Data Models (augment)

YANG allows a module to insert additional nodes into data models, including both the current module (and its submodules) or an external module. This is useful e.g. for vendors to add vendor-specific parameters to standard data models in an interoperable way.

The `augment` statement defines the location in the data model hierarchy where new nodes are inserted, and the `when` statement defines the conditions when the new nodes are valid.

```
augment /system/login/user {
    when "class != 'wheel'";
    leaf uid {
        type uint16 {
            range "1000 .. 30000";
        }
    }
}
```

This example defines a `uid` node that only is valid when the user's `class` is not `wheel`.

If a module augments another model, the XML representation of the data will reflect the prefix of the augmenting model. For example, if the above augmentation were in a module with prefix `other`, the XML would look like:

```
<user>
  <name>alicew</name>
  <full-name>Alice N. Wonderland</full-name>
  <class>drop-out</class>
  <other:uid>1024</other:uid>
</user>
```

RPC Definitions

YANG allows the definition of NETCONF RPCs. The method names, input parameters and output parameters are modeled using YANG data definition statements.

```
rpc activate-software-image {
    input {
        leaf image-name {
            type string;
        }
    }
    output {
        leaf status {
            type string;
        }
    }
}

<rpc message-id="101"
      xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <activate-software-image xmlns="http://acme.example.com/system">
      <name>acmefw-2.3</name>
    </activate-software-image>
</rpc>

<rpc-reply message-id="101"
           xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <status xmlns="http://acme.example.com/system">
      The image acmefw-2.3 is being installed.
    </status>
</rpc-reply>
```

Notification Definitions

YANG allows the definition of notifications suitable for NETCONF. YANG data definition statements are used to model the content of the notification.

```
notification link-failure {
    description "A link failure has been detected";
    leaf if-name {
        type leafref {
            path "/interfaces/interface/name";
        }
    }
    leaf if-admin-status {
        type ifAdminStatus;
    }
}

<notification xmlns="urn:ietf:params:netconf:capability:notification:1.0">
  <eventTime>2007-09-01T10:00:00Z</eventTime>
  <link-failure xmlns="http://acme.example.com/system">
    <if-name>so-1/2/3.0</if-name>
    <if-admin-status>up</if-admin-status>
  </link-failure>
```

```
</notification>
```

Working With YANG Modules

Assume we have a small trivial YANG file `test.yang`:

```
module test {
    namespace "http://tail-f.com/test";
    prefix "t";

    container top {
        leaf a {
            type int32;
        }
        leaf b {
            type string;
        }
    }
}
```



Tip There is an Emacs mode suitable for YANG file editing in the system distribution. It is called `yang-mode.el`

We can use **ncsc** compiler to compile the YANG module.

```
$ ncsc -c test.yang
```

The above command creates an output file `test.fxs` that is a compiled schema that can be loaded into the system. The **ncsc** compiler with all its flags is fully described in `ncsc(1)` in *NSO 5.7 Manual Pages*.

There exists a number of standards based auxiliary YANG modules defining various useful data types. These modules, as well as their accompanying `.fxs` files can be found in the `$(NCS_DIR)/src/confd/yang` directory in the distribution.

The modules are:

- *ietf-yang-types* - defining some basic data types such as counters, dates and times.
- *ietf-inet-types* - defining several useful types related to IP addresses.

Whenever we wish to use any of those predefined modules we need to not only import the module into our YANG module, but we must also load the corresponding `.fxs` file for the imported module into the system.

So if we extend our `test` module so that it looks like:

```
module test {
    namespace "http://tail-f.com/test";
    prefix "t";

    import ietf-inet-types {
        prefix inet;
    }

    container top {
        leaf a {
            type int32;
        }
        leaf b {
            type string;
        }
    }
}
```

```

        }
        leaf ip {
            type inet:ipv4-address;
        }
    }
}

```

Normally when importing other YANG modules we must indicate through the --yangpath flag to **ncsc** where to search for the imported module. In the special case of the standard modules, this is not required.

We compile the above as:

```

$ ncsc -c test.yang
$ ncsc --get-info test.fxs
fxs file
Ncsc version:          "3.0_2"
uri:                  http://tail-f.com/test
id:                   http://tail-f.com/test
prefix:                "t"
flags:                 6
type:                  cs
mountpoint:           undefined
exported agents:      all
dependencies:         ['http://www.w3.org/2001/XMLSchema',
                      'urn:ietf:params:xml:ns:yang:inet-types']
source:                ["test.yang"]

```

We see that the generated .fxs file has a dependency to the standard `urn:ietf:params:xml:ns:yang:inet-types` namespace. Thus if we try to start NSO we must also ensure that the fxs file for that namespace is loaded.

Failing to do so gives:

```

$ ncs -c ncs.conf --foreground --verbose
The namespace urn:ietf:params:xml:ns:yang:inet-types (referenced by http://tail-f.com/test) c
Daemon died status=21

```

The remedy is to modify `ncs.conf` so that it contains the proper load path or to provide the directory containing the fxs file, alternatively we can provide the path on the command line. The directory `$(NCS_DIR)/etc/ncs` contains pre-compiled versions of the standard YANG modules.

```
$ ncs -c ncs.conf --addloadpath $(NCS_DIR)/etc/ncs --foreground --verbose
```

`ncs.conf` is the configuration file for NSO itself. It is described in the `ncs.conf(5)` in *NSO 5.7 Manual Pages*.

Integrity Constraints

The YANG language has built-in declarative constructs for common integrity constraints. These constructs are conveniently specified as `must` statements.

A `must` statement is an XPath expression that must evaluate to true or a non-empty node-set.

An example is:

```

container interface {
    leaf ifType {
        type enumeration {
            enum ethernet;
            enum atm;
        }
    }
}

```

```

leaf ifMTU {
    type uint32;
}
must "ifType != 'ethernet' or "
+ "(ifType = 'ethernet' and ifMTU = 1500)" {
    error-message "An ethernet MTU must be 1500";
}
must "ifType != 'atm' or "
+ "(ifType = 'atm' and ifMTU <= 17966 and ifMTU >= 64)" {
    error-message "An atm MTU must be 64 .. 17966";
}
}
}

```

XPath is a very powerful tool here. It is often possible to express most realistic validation constraints using XPath expressions. Note that for performance reasons, it is recommended to use the `tailf:dependency` statement in the `must` statement. The compiler gives a warning if a `must` statement lacks a `tailf:dependency` statement, and it cannot derive the dependency from the expression. The options `--fail-on-warnings` or `-E TAILF_MUST_NEED_DEPENDENCY` can be given to force this warning to be treated as an error. See `tailf:dependency` in `tailf_yang_extensions(5)` in *NSO 5.7 Manual Pages* for details.

Another useful built-in constraint checker is the `unique` statement.

With the YANG code:

```

list server {
    key "name";
    unique "ip port";
    leaf name {
        type string;
    }
    leaf ip {
        type inet:ip-address;
    }
    leaf port {
        type inet:port-number;
    }
}

```

We specify that the combination of IP and port must be unique. Thus the configuration:

```

<server>
  <name>smtp</name>
  <ip>192.0.2.1</ip>
  <port>25</port>
</server>

<server>
  <name>http</name>
  <ip>192.0.2.1</ip>
  <port>25</port>
</server>

```

is not valid.

The usage of leafrefs (See the YANG specification) ensures that we do not end up with configurations with dangling pointers. Leafrefs are also especially good, since the CLI and Web UI can render a better interface.

If other constraints are necessary, validation callback functions can be programmed in Java, Python, or Erlang. See `tailf:validate` in `tailf_yang_extensions(5)` in *NSO 5.7 Manual Pages* for details.

The when statement

The when statement is used to make its parent statement conditional. If the XPath expression specified as the argument to this statement evaluates to false, the parent node cannot be given configured. Furthermore, if the parent node exists, and some other node is changed so that the XPath expression becomes false, the parent node is automatically deleted. For example:

```
leaf a {
    type boolean;
}
leaf b {
    type string;
    when ".../a = 'true'";
}
```

This data model snippet says that 'b' can only exist if 'a' is true. If 'a' is true, and 'b' has a value, and 'a' is set to false, 'b' will automatically be deleted.

Since the XPath expression in theory can refer to any node in the data tree, it has to be re-evaluated when any node in the tree is modified. But this would have a disastrous performance impact, so in order to avoid this, NSO keeps track of dependencies for each when expression. In some simple cases, the **confdc** can figure out these dependencies by itself. In the example above, NSO will detect that 'b' is dependent on 'a', and evaluate b's XPath expression only if 'a' is modified. If **confdc** cannot detect the dependencies by itself, it requires a **tailf:dependency** statement in the when statement. See **tailf:dependency** in [tailf_yang_extensions\(5\)](#) in *NSO 5.7 Manual Pages* for details.

Using the Tail-f Extensions with YANG

Tail-f has an extensive set of extensions to the YANG language that integrates YANG models in NSO. For example when we have `config false;` data, we may wish to invoke user C code to deliver the statistics data in runtime. To do this we annotate the YANG model with a Tail-f extension called `tailf:callpoint`.

Alternatively we may wish to invoke user code to validate the configuration, this is also controlled through an extension called `tailf:validate`.

All these extensions are handled as normal YANG extensions. (YANG is designed to be extended) We have defined the Tail-f proprietary extensions in a file `$(NCS_DIR)/src/ncs/yang/tailf-common.yang`

Continuing with our previous example, adding a callpoint and a validation point we get:

```
module test {
    namespace "http://tail-f.com/test";
    prefix "t";

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

    container top {
        leaf a {
            type int32;
            config false;
            tailf:callpoint mycp;
        }
    }
}
```

```

leaf b {
    tailf:validate myvalcp {
        tailf:dependency "../a";
    }
    type string;
}
leaf ip {
    type inet:ipv4-address;
}
}
}

```

The above module contains a callpoint and a validation point. The exact syntax for all Tail-f extensions are defined in the `tailf-common.yang` file.

Note the import statement where we import `tailf-common`.

When we are using YANG specifications in order to generate Java classes for ConfM, these extensions are ignored. They only make sense on the device side. It is worth mentioning them though, since EMS developers will certainly get the YANG specifications from the device developers, thus the YANG specifications may contain extensions

The man page `tailf_yang_extensions(5)` in *NSO 5.7 Manual Pages* describes all the Tail-f YANG extensions.

Using a YANG annotation file

Sometimes it is convenient to specify all Tail-f extension statements in-line in the original YANG module. But in some cases, e.g. when implementing a standard YANG module, it is better to keep the Tail-f extension statements in a separate annotation file. When the YANG module is compiled to an fxs file, the compiler is given the original YANG module, and any number of annotation files.

A YANG annotation file is a normal YANG module which imports the module to annotate. Then the `tailf:annotate` statement is used to annotate nodes in the original module. For example, the module test above can be annotated like this:

```

module test {
    namespace "http://tail-f.com/test";
    prefix "t";

    import ietf-inet-types {
        prefix inet;
    }

    container top {
        leaf a {
            type int32;
            config false;
        }
        leaf b {
            type string;
        }
        leaf ip {
            type inet:ipv4-address;
        }
    }
}

module test-ann {
    namespace "http://tail-f.com/test-ann";
    prefix "ta";
}

```

```

import test {
    prefix t;
}
import tailf-common {
    prefix tailf;
}

tailf:annotate "/t:top/t:a" {
    tailf:callpoint mycp;
}

tailf:annotate "/t:top" {
    tailf:annotate "t:b" { // recursive annotation
        tailf:validate myvalcp {
            tailf:dependency ".../t:a";
        }
    }
}

```

In order to compile the module with annotations, use the -a parameter to **confdc**:

```
confdc -c -a test-ann.yang test.yang
```

Custom Help Texts and Error Messages

Certain parts of a YANG model are used by northbound agents, e.g. CLI and Web UI, to provide the end-user with custom help texts and error messages.

Custom Help Texts

A YANG statement can be annotated with a `description` statement which is used to describe the definition for a reader of the module. This text is often too long and too detailed to be useful as help text in a CLI. For this reason, NSO by default does not use the text in the `description` for this purpose. Instead, a tail-f specific statement, `tailf:info` is used. It is recommended that the standard `description` statement contains a detailed description suitable for a module reader (e.g. NETCONF client or server implementor), and `tailf:info` contains a CLI help text.

As an alternative, NSO can be instructed to use the text in the `description` statement also for CLI help text. See the option `--use-description` in `ncsc(1)` in *NSO 5.7 Manual Pages*.

For example, CLI uses the help text to prompt for a value of this particular type. The CLI shows this information during tab/command completion or if the end-user explicitly asks for help using the `?-` character. The behavior depends on the mode the CLI is running in.

The Web UI uses this information likewise to help the end-user.

The `mtu` definition below has been annotated to enrich the end-user experience:

```

leaf mtu {
    type uint16 {
        range "1 .. 1500";
    }
    description
        "MTU is the largest frame size that can be transmitted
         over the network. For example, an Ethernet MTU is 1,500
         bytes. Messages longer than the MTU must be divided
         into smaller frames.";
    tailf:info

```

```

        "largest frame size";
    }
}
```

Custom Help Text in a Typedef

Alternatively, we could have provided the help text in a `typedef` statement as in:

```

typedef mtuType {
    type uint16 {
        range "1 .. 1500";
    }
    description
        "MTU is the largest frame size that can be transmitted over the
         network. For example, an Ethernet MTU is 1,500
         bytes. Messages longer than the MTU must be
         divided into smaller frames.";
    tailf:info
        "largest frame size";
}

leaf mtu {
    type mtuType;
}
```

If there is an explicit help text attached to a leaf, it overrides the help text attached to the type.

Custom Error Messages

A statement can have an optional error-message statement. The north-bound agents, for example, the CLI uses this to inform the end-user about a provided value which is not of the correct type. If no custom error-message statement is available NSO generates a built-in error message, e.g. "1505 is too large.".

All northbound agents use the extra information provided by an `error-message` statement.

The `typedef` statement below has been annotated to enrich the end-user experience when it comes to error information:

```

typedef mtuType {
    type uint32 {
        range "1..1500" {
            error-message
                "The MTU must be a positive number not "
                + "larger than 1500";
        }
    }
}
```

An Example: Modeling a List of Interfaces

Say for example that we want to model the interface list on a Linux based device. Running the `ip link list` command reveals the type of information we have to model

```
$ /sbin/ip link list
1: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 00:12:3f:7d:b0:32 brd ff:ff:ff:ff:ff:ff
2: lo: <LOOPBACK,UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
3: dummy0: <BROADCAST,NOARP> mtu 1500 qdisc noop
    link/ether a6:17:b9:86:2c:04 brd ff:ff:ff:ff:ff:ff
```

and this is how we want to represent the above in XML:

```
<?xml version="1.0"?>
<config xmlns="http://example.com/ns/link">
  <links>
    <link>
      <name>eth0</name>
      <flags>
        <UP/>
        <BROADCAST/>
        <MULTICAST/>
      </flags>
      <addr>00:12:3f:7d:b0:32</addr>
      <brd>ff:ff:ff:ff:ff:ff</brd>
      <mtu>1500</mtu>
    </link>

    <link>
      <name>lo</name>
      <flags>
        <UP/>
        <LOOPBACK/>
      </flags>
      <addr>00:00:00:00:00:00</addr>
      <brd>00:00:00:00:00:00</brd>
      <mtu>16436</mtu>
    </link>
  </links>
</config>
```

An interface or a link has data associated with it. It also has a name, an obvious choice to use as the key - the data item which uniquely identifies an individual interface.

The structure of a YANG model is always a header, followed by type definitions, followed by the actual structure of the data. A YANG model for the interface list starts with a header:

```
module links {
  namespace "http://example.com/ns/links";
  prefix link;

  revision 2007-06-09 {
    description "Initial revision.";
  }
  ...
}
```

A number of datatype definitions may follow the YANG module header. Looking at the output from **/sbin/ip** we see that each interface has a number of boolean flags associated with it, e.g. UP, and NOARP.

One way to model a sequence of boolean flags is as a sequence of statements:

```
leaf UP {
  type boolean;
  default false;
}
leaf NOARP {
  type boolean;
  default false;
}
```

A better way is to model this as:

```
leaf UP {
  type empty;
}
```

```
leaf NOARP {
    type empty;
}
```

We could choose to group these leafs together into a grouping. This makes sense if we wish to use the same set of boolean flags in more than one place. We could thus create a named grouping such as:

```
grouping LinkFlags {
    leaf UP {
        type empty;
    }
    leaf NOARP {
        type empty;
    }
    leaf BROADCAST {
        type empty;
    }
    leaf MULTICAST {
        type empty;
    }
    leaf LOOPBACK {
        type empty;
    }
    leaf NOTRAILERS {
        type empty;
    }
}
```

The output from `/sbin/ip` also contains Ethernet MAC addresses. These are best represented by the `mac-address` type defined in the `ietf-yang-types.yang` file. The `mac-address` type is defined as:

```
typedef mac-address {
    type string {
        pattern '[0-9a-fA-F]{2}(:[0-9a-fA-F]{2}){5}';
    }
    description
        "The mac-address type represents an IEEE 802 MAC address.

        This type is in the value set and its semantics equivalent to
        the MacAddress textual convention of the SMIv2.";
    reference
        "IEEE 802: IEEE Standard for Local and Metropolitan Area
        Networks: Overview and Architecture
        RFC 2579: Textual Conventions for SMIv2";
}
```

This defines a restriction on the string type, restricting values of the defined type "mac-address" to be strings adhering to the regular expression `[0-9a-fA-F]{2}(:[0-9a-fA-F]{2}){5}`. Thus strings such as `a6:17:b9:86:2c:04` will be accepted.

Queue disciplines are associated with each device. They are typically used for bandwidth management. Another string restriction we could do is to define an enumeration of the different queue disciplines that can be attached to an interface.

We could write this as:

```
typedef QueueDisciplineType {
    type enumeration {
        enum pfifo_fast;
        enum noqueue;
        enum noop;
        enum htp;
```

```

    }
}
```

There are a large number of queue disciplines and we only list a few here. The example serves to show that using enumerations we can restrict the values of the data set in a way that ensures that data entered always is valid from a syntactical point of view.

Now that we have a number of usable datatypes, we continue with the actual data structure describing a list of interface entries:

```

container links {
    list link {
        key name;
        unique addr;
        max-elements 1024;
        leaf name {
            type string;
        }
        container flags {
            uses LinkFlags;
        }
        leaf addr {
            type yang:mac-address;
            mandatory true;
        }
        leaf brd {
            type yang:mac-address;
            mandatory true;
        }
        leaf qdisc {
            type QueueDisciplineType;
            mandatory true;
        }
        leaf qlen {
            type uint32;
            mandatory true;
        }
        leaf mtu {
            type uint32;
            mandatory true;
        }
    }
}
```

The `key` attribute on the leaf named "name" is important. It indicates that the leaf is the instance key for the list entry named "link". All the link leafs are guaranteed to have unique values for their name leafs due to the key declaration.

If one leaf alone does not uniquely identify an object, we can define multiple keys. At least one leaf *must* be an instance key - we cannot have lists without a key.

List entries are ordered and indexed according to the value of the key(s).

Modeling Relationships

A very common situation when modeling a device configuration is that we wish to model a relationship between two objects. This is achieved by means of the `leafref` statements. A `leafref` points to a child of a list entry which either is defined using a `key` or `unique` attribute.

The `leafref` statement can be used to express three flavors of relationships: *extensions*, *specializations* and *associations*. Below we exemplify this by extending the "link" example from above.

Firstly, assume we want to put/store the queue disciplines from the previous section in a separate container - not embedded inside the links container.

We then specify a separate container , containing all the queue disciplines which each refers to a specific link entry. This is written as:

```
container queueDisciplines {
    list queueDiscipline {
        key linkName;
        max-elements 1024;
        leaf linkName {
            type leafref {
                path "/config/links/link/name";
            }
        }

        leaf type {
            type QueueDisciplineType;
            mandatory true;
        }
        leaf length {
            type uint32;
        }
    }
}
```

The linkName statement is both an instance key of the queueDiscipline list, and at the same time refers to a specific link entry. This way we can extend the amount of configuration data associated with a specific link entry.

Secondly, assume we want to express a restriction or specialization on Ethernet link entries, e.g. it should be possible to restrict interface characteristics such as 10Mbps and half duplex.

We then specify a separate container, containing all the specializations which each refers to a specific link:

```
container linkLimitations {
    list LinkLimitation {
        key linkName;
        max-elements 1024;
        leaf linkName {
            type leafref {
                path "/config/links/link/name";
            }
        }

        container limitations {
            leaf only10Mbps { type boolean; }
            leaf onlyHalfDuplex { type boolean; }
        }
    }
}
```

The linkName leaf is both an instance key to the linkLimitation list, and at the same time refers to a specific link leaf. This way we can restrict or specialize a specific link.

Thirdly, assume we want to express that one of the link entries should be the default link. In that case we enforce an association between a non-dynamic defaultLink and a certain link entry:

```
leaf defaultLink {
    type leafref {
        path "/config/links/link/name";
    }
}
```

```
}
```

Ensuring Uniqueness

Key leafs are always unique. Sometimes we may wish to impose further restrictions on objects. For example, we can ensure that all link entries have a unique MAC address. This is achieved through the use of the `unique` statement:

```
container servers {
    list server {
        key name;
        unique "ip port";
        unique "index";
        max-elements 64;
        leaf name {
            type string;
        }
        leaf index {
            type uint32;
            mandatory true;
        }
        leaf ip {
            type inet:ip-address;
            mandatory true;
        }
        leaf port {
            type inet:port-number;
            mandatory true;
        }
    }
}
```

In this example we have two `unique` statements. These two groups ensure that each server has a unique index number as well as a unique ip and port pair.

Default Values

A leaf can have a static or dynamic default value. Static default values are defined with the `default` statement in the data model. For example:

```
leaf mtu {
    type int32;
    default 1500;
}
```

and:

```
leaf UP {
    type boolean;
    default true;
}
```

A dynamic default value means that the default value for the leaf is the value of some other leaf in the data model. This can be used to make the default values configurable by the user. Dynamic default values are defined using the `tailf:default-ref` statement. For example, suppose we want to make the MTU default value configurable:

```
container links {
    leaf mtu {
        type uint32;
    }
    list link {
```

```

        key name;
        leaf name {
            type string;
        }
        leaf mtu {
            type uint32;
            tailf:default-ref '../../.mtu';
        }
    }
}

```

Now suppose we have the following data:

```

<links>
    <mtu>1000</mtu>
    <link>
        <name>eth0</name>
        <mtu>1500</mtu>
    </link>
    <link>
        <name>eth1</name>
    </link>
</links>

```

In the example above, link `eth0` has the mtu 1500, and link `eth1` has mtu 1000. Since `eth1` does not have a mtu value set, it defaults to the value of `../../.mtu`, which is 1000 in this case.



Note Whenever a leaf has a default value it implies that the leaf can be left out from the XML document, i.e. mandatory = false.

With the default value mechanism an old configuration can be used even after having added new settings.

Another example where default values are used is when a new instance is created. If all leafs within the instance have default values, these need not be specified in, for example, a NETCONF create operation.

The Final Interface YANG model

Here is the final interface YANG model with all constructs described above:

```

module links {
    namespace "http://example.com/ns/link";
    prefix link;

    import ietf-yang-types {
        prefix yang;
    }

    grouping LinkFlagsType {
        leaf UP {
            type empty;
        }
        leaf NOARP {
            type empty;
        }
        leaf BROADCAST {
            type empty;
        }
        leaf MULTICAST {
            type empty;
        }
    }
}

```

```

        }
    leaf LOOPBACK {
        type empty;
    }
    leaf NOTRAILERS {
        type empty;
    }
}

typedef QueueDisciplineType {
    type enumeration {
        enum pfifo_fast;
        enum noqueue;
        enum noop;
        enum htb;
    }
}
container config {
    container links {
        list link {
            key name;
            unique addr;
            max-elements 1024;
            leaf name {
                type string;
            }
            container flags {
                uses LinkFlagsType;
            }
            leaf addr {
                type yang:mac-address;
                mandatory true;
            }
            leaf brd {
                type yang:mac-address;
                mandatory true;
            }
            leaf mtu {
                type uint32;
                default 1500;
            }
        }
    }
    container queueDisciplines {
        list queueDiscipline {
            key linkName;
            max-elements 1024;
            leaf linkName {
                type leafref {
                    path "/config/links/link/name";
                }
            }
            leaf type {
                type QueueDisciplineType;
                mandatory true;
            }
            leaf length {
                type uint32;
            }
        }
    }
    container linkLimitations {

```

```

list linkLimitation {
    key linkName;
    leaf linkName {
        type leafref {
            path "/config/links/link/name";
        }
    }
    container limitations {
        leaf only10Mbps {
            type boolean;
            default false;
        }
        leaf onlyHalfDuplex {
            type boolean;
            default false;
        }
    }
}
container defaultLink {
    leaf linkName {
        type leafref {
            path "/config/links/link/name";
        }
    }
}
}
}

```

If the above YANG file is saved on disk, as `links.yang`, we can compile and link it using the `confdc` compiler:

```
$ confdc -c links.yang
```

We now have a ready to use schema file named `links.fxs` on disk. To actually run this example, we need to copy the compiled `links.fxs` to a directory where NSO can find it.

More on leafrefs

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

```

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

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

```

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

```

list host {
    key "name";

```

```

leaf name {
    type string;
}

list server {
    key "ip port";
    leaf ip {
        type inet:ip-address;
    }
    leaf port {
        type inet:port-number;
    }
    ...
}
}

```

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

```

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

```

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

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

This syntax quickly gets awkward and error prone. NSO supports a shorthand syntax, by introducing an XPath function `deref()` (see the section called “*XPATH FUNCTIONS*” in *NSO 5.7 Manual Pages*). Technically, this function follows a leafref value, and returns all nodes that the leafref refer to (typically just one). The example above can be written like this:

```

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

```

```
}
```

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

Using Multiple Namespaces

There are several reasons for supporting multiple configuration namespaces. Multiple namespaces can be used to group common datatypes and hierarchies to be used by other YANG models. Separate namespaces can be used to describe the configuration of unrelated sub-systems, i.e. to achieve strict configuration data model boundaries between these sub-systems.

As an example, `datatypes.yang` is a YANG module which defines a reusable data type.

```
module datatypes {
    namespace "http://example.com/ns/dt";
    prefix dt;

    grouping countersType {
        leaf recvBytes {
            type uint64;
            mandatory true;
        }
        leaf sentBytes {
            type uint64;
            mandatory true;
        }
    }
}
```

We compile and link `datatypes.yang` into a final schema file representing the `http://example.com/ns/dt` namespace:

```
$ confdc -c datatypes.yang
```

To reuse our user defined `countersType`, we must import the `datatypes` module.

```
module test {
    namespace "http://tail-f.com/test";
    prefix "t";

    import datatypes {
        prefix dt;
    }

    container stats {
        uses dt:countersType;
    }
}
```

When compiling this new module that refers to another module, we must indicate to `confdc` where to search for the imported module:

```
$ confdc -c test.yang --yangpath /path/to/dt
```

`confdc` also searches for referred modules in the colon (:) separated path defined by the environment variable `YANG_MODPATH` and `.` (dot) is implicitly included.

Module Names, Namespaces and Revisions

We have three different entities that define our configuration data.

- The module name. A system typically consists of several modules. In the future we also expect to see standard modules in a manner similar to how we have standard SNMP modules.
It is highly recommended to have the vendor name embedded in the module name, similar to how vendors have their names in proprietary MIB s today.
- The XML namespace. A module defines a namespace. This is an important part of the module header. For example we have:

```
module acme-system {
    namespace "http://acme.example.com/system";
    ....
```

The namespace string must uniquely define the namespace. It is very important that once we have settled on a namespace we never change it. The namespace string should remain the same between revisions of a product. Do not embed revision information in the namespace string since that breaks manager side NETCONF scripts.

- The revision statement as in:

```
module acme-system {
    namespace "http://acme.example.com/system";
    prefix "acme";

    revision 2007-06-09;
    ....
```

The revision is exposed to a NETCONF manager in the capabilities sent from the agent to the NETCONF manager in the initial hello message. The fine details of revision management is being worked on in the IETF NETMOD working group and is not finalized at the time of this writing.

What is clear though, is that a manager should base its version decisions on the information in the revision string.

A capabilities reply from a NETCONF agent to the manager may look as:

```
<?xml version="1.0" encoding="UTF-8"?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:writable-running:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:confirmed-commit:1.0</capability>
    <capability>urn:ietf:params:netconf:capability>xpath:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:validate:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:rollback-on-error:1.0</capability>
    <capability>http://example.com/ns/link?revision=2007-06-09</capability>
    ....
```

where the revision information for the `http://example.com/ns/link` namespace is encoded as `?revision=2007-06-09` using standard URI notation.

When we change the data model for a namespace, it is recommended to change the revision statement, and to never make any changes to the data model that are backwards incompatible. This means that all leafs that are added must be either optional or have a default value. That way it is ensured that old NETCONF client code will continue to function on the new data model. Section 10

of RFC 6020 and section 11 of RFC 7950 defines exactly what changes can be made to a data model in order to not break old NETCONF clients.

Hash Values and the id-value Statement

Internally and in the programming APIs, NSO uses integer values to represent YANG node names and the namespace URI. This conserves space and allows for more efficient comparisons (including `switch` statements) in the user application code. By default, `confdc` automatically computes a hash value for the namespace URI and for each string that is used as a node name.

Conflicts can occur in the mapping between strings and integer values - i.e. the initial assignment of integers to strings is unable to provide a unique, bi-directional mapping. Such conflicts are extremely rare (but possible) when the default hashing mechanism is used.

The conflicts are detected either by `confdc` or by the NSO daemon when it loads the `.fxs` files.

If there are any conflicts reported they will pertain to XML tags (or the namespace URI),

There are two different cases:

- Two different strings mapped to the same integer. This is the classical hash conflict - extremely rare due to the high quality of the hash function used. The resolution is to manually assign a unique value to one of the conflicting strings. The value should be greater than $2^{31}+2$ but less than $2^{32}-1$. This way it will be out of the range of the automatic hash values, which are between 0 and $2^{31}-1$. The best way to choose a value is by using a random number generator, as in `2147483649 + rand:uniform(2147483645)`. The `tailf:id-value` should be placed as a substatement to the statement where the conflict occurs, or in the `module` statement in case of namespace URI conflict.
- One string mapped to two different integers. This is even more rare than the previous case - it can only happen if a hash conflict was detected and avoided through the use of `tailf:id-value` on one of the strings, and that string also occurs somewhere else. The resolution is to add the same `tailf:id-value` to the second occurrence of the string.

NSO caveats

The union type and value conversion

When converting a string to an enumeration value, the order of types in the union is important when the types overlap. The first matching type will be used, so we recommend to have the narrower (or more specific) types first.

Consider the example below:

```
leaf example {
    type union {
        type string; // NOTE: widest type first
        type int32;
        type enumeration {
            enum "unbounded";
        }
    }
}
```

Converting the string `42` to a typed value using the YANG model above, will always result in a string value even though it is the string representation of an `int32`. Trying to convert the string `unbounded`

will also result in a string value instead of the enumeration, because the enumeration is placed after the string.

Instead consider the example below where the string (being a wider type) is placed last:

```
leaf example {
  type union {
    type enumeration {
      enum "unbounded";
    }
    type int32;
    type string; // NOTE: widest type last
  }
}
```

Converting the string 42 to the corresponding union value will result in a `int32`. Trying to convert the string `unbounded` will also result in the enumeration value as expected. The relative order of the `int32` and enumeration do not matter as they do not overlap.

Using the C and Python APIs to convert a string to a given value is further limited by the lack of restriction matching on the types. Consider the following example:

```
leaf example {
  type union {
    type string {
      pattern "[a-z]+[0-9]+";
    }
    type int32;
  }
}
```

Converting the string `42` will result in a string value, even though the pattern requires the string to begin with a character in the "a" to "z" range. This value will be considered invalid by NSO if used in any calls handled by NSO.

To avoid issues when working with unions place wider types at the end. As an example put `string` last, `int8` before `int16` etc.

User defined types

When using user defined types together with NSO the compiled schema does not contain the original type as specified in the YANG file. This imposes some limitations on the running system.

High-level APIs are unable to infer the correct type of a value as this information is left out when the schema is compiled. It is possible to work around this issue by specifying the type explicitly whenever setting values of a user-defined type.



CHAPTER 9

Using CDB

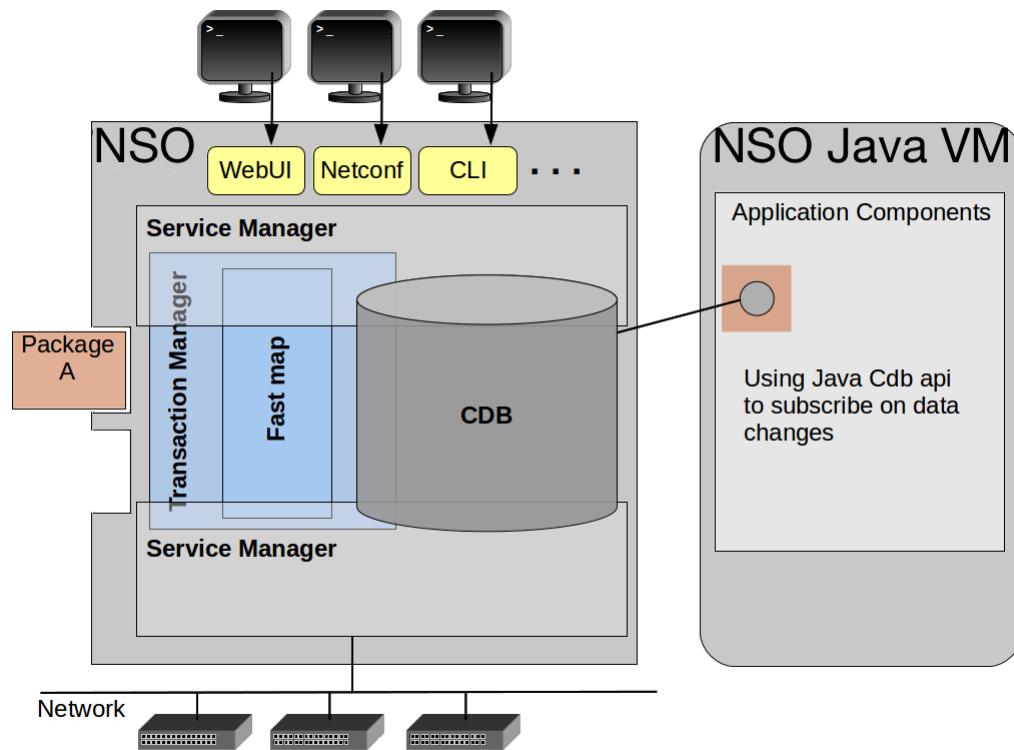
- [Introduction, page 99](#)
- [The NSO Data Model, page 100](#)
- [Addressing Data Using Keypaths, page 102](#)
- [Subscriptions, page 103](#)
- [Sessions, page 104](#)
- [Loading Initial Data Into CDB, page 105](#)
- [Operational Data in CDB, page 106](#)
- [Example, page 107](#)
- [Automatic Schema Upgrades and Downgrades, page 118](#)
- [Using Initialization Files for Upgrade, page 120](#)
- [Writing an Upgrade Package Component, page 123](#)

Introduction

When using CDB to store the configuration data, the applications need to be able to:

- 1 Read configuration data from the database.
- 2 React to changes to the database. There are several possible writers to the database, such as the CLI, NETCONF sessions, the Web UI, either of the NSO sync commands, alarms that get written into the alarm table, NETCONF notifications that arrive at NSO or the NETCONF agent.

[Figure 34, “NSO CDB Architecture Scenario”](#) illustrates the architecture when CDB is used. The Application components read configuration data and subscribe to changes to the database using a simple RPC-based API. The API is part of the Java library and is fully documented in the Javadoc for CDB.

Figure 34. NSO CDB Architecture Scenario

While CDB is the default data store for configuration data in NSO, it is possible to use an external database, if needed. See the example `examples.ncs/getting-started/developing-with-ncs/6-extern-db` for details.

In the following, we will use the files in `examples.ncs/service-provider/mpls-vpn` as a source for our examples. Refer to README in that directory for additional details.

The NSO Data Model

NSO is designed to manage devices and services. NSO uses YANG as the overall modeling language. YANG models describe the NSO configuration, the device configurations and the configuration of services. Therefore it is vital to understand the data model for NSO including these aspects. The YANG models are available in `$NCS_DIR/src/ncs/yang` and are structured as follows.

`tailf-ncs.yang` is the top module that includes the following sub-modules:

<code>tailf-ncs-common.yang</code>	common definitions
<code>tailf-ncs-packages.yang</code>	this sub-module defines the management of packages that are run by NSO. A package contains custom code, models, documentation for any function added to the NSO platform. It can for example be a service application, or a southbound integration to a device.
<code>tailf-ncs-devices.yang</code>	this is a core model of NSO. The device model defines everything a user can do with a device that NSO speaks to via a Network Element Driver, NED.

`tailf-ncs-services.yang`

services represent anything that spans across devices. This can for example be MPLS VPN, MEF e-line, BGP peer, web site. NSO provides several mechanisms to handle services in general which are specified by this model. Also it defines placeholder containers under which developers, as an option, can augment their specific services.

`tailf-ncs-snmp-notification-receiver.yang`

NSO can subscribe to SNMP notifications from the devices. The subscription is specified by this model.

`tailf-ncs-java-vm.yang`

Custom code that is part of a package is loaded and executed by the NSO Java VM. This is managed by this model.

Further, when browsing `$NCS_DIR/src/ncs/yang` you will find models for all aspects of NSO functionality, for example

`tailf-ncs-alarms.yang`

This model defines how NSO manages alarms. The source of an alarm can be anything like an NSO state change, SNMP or NETCONF notification.

`tailf-ncs-snmp.yang`

This model defines how to configure the NSO northbound SNMP agent.

`tailf-ncs-config.yang`

This model describes the layout of the NSO config file, usually called `ncs.conf`

`tailf-ncs-packages.yang`

This model describes the layout of the file `package-meta-data.xml`. All user code, data models MIBS, Java code is always contained in an NSO package. The `package-meta-data.xml` file must always exist in a package and describes the package.

These models will be illustrated and briefly explained below. Note that the figures only contain some relevant aspects of the model and are far from complete. The details of the model are explained in the respective sections.

A good way to learn the model is to start the NSO CLI and use tab completion to navigate the model. Note that depending if you are in operation mode or configuration mode different parts of the model will show up. Also try using TAB to get a list of actions at the level you want, for example **devices TAB**.

Another way to learn and explore the NSO model is to use the **yanger** tool to render a tree output from the NSO model: **yanger -f tree --tree-depth=3 tailf-ncs.yang** This will show a tree for the complete model. Below is a truncated example:

Example 35. Using yanger

```
$ yanger -f tree --tree-depth=3 tailf-ncs.yang
module: tailf-ncs
  +-rw ssh
    |   +-rw host-key-verification?    ssh-host-key-verification-level
    |   +-rw private-key* [name]
    |     +-rw name                  string
    |     +-rw key-data              ssh-private-key
    |     +-rw passphrase?          tailf:aes-256-cfb-128-encrypted-string
  +-rw cluster
    |   +-rw remote-node* [name]
    |     +-rw name                node-name
    |     +-rw address?            inet:host
    |     +-rw port?               inet:port-number
    |     +-rw ssh
    |     +-rw authgroup          -> /cluster/authgroup/name
    |     +-rw trace?              trace-flag
```

```

|   |   +---rw username?          string
|   |   +---rw notifications
|   |   +---ro device* [name]
+---rw authgroup* [name]
|   |   +---rw name             string
|   |   +---rw default-map!
|   |   +---rw umap* [local-user]
+---rw commit-queue
|   +---rw enabled?           boolean
+---ro enabled?             boolean
+---ro connection*
    +---ro remote-node?      -> /cluster/remote-node/name
    +---ro address?          inet:ip-address
    +---ro port?              inet:port-number
    +---ro channels?          uint32
    +---ro local-user?        string
    +---ro remote-user?       string
    +---ro status?            enumeration
    +---ro trace?             enumeration
...

```

Addressing Data Using Keypaths

As CDB stores hierarchical data as specified by a YANG model, data is addressed by a path to the key. We call this a *keypath*. A keypath provides a path through the configuration data tree. A keypath can be either absolute or relative. An absolute keypath starts from the root of the tree, while a relative path starts from the "current position" in the tree. They are differentiated by presence or absence of a leading "/". Navigating the configuration data tree is thus done in the same way as a directory structure. It is possible to change the *current position* with for example the `CdbSession.cd()` method. Several of the API methods take a keypath as a parameter.

YANG elements that are lists of other YANG elements can be traversed using two different path notations. Consider the following YANG model fragment:

Example 36. L3 VPN YANG Extract

```

module l3vpn {

namespace "http://com/example/l3vpn";
prefix l3vpn;

...

container topology {
    list role {
        key "role";
        tailf:cli-compact-syntax;
        leaf role {
            type enumeration {
                enum ce;
                enum pe;
                enum p;
            }
        }
    }

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

```

```

        }
    }

list connection {
    key "name";
    leaf name {
        type string;
    }
    container endpoint-1 {
        tailf:cli-compact-syntax;
        uses connection-grouping;
    }
    container endpoint-2 {
        tailf:cli-compact-syntax;
        uses connection-grouping;
    }
    leaf link-vlan {
        type uint32;
    }
}
}

```

We can use the method `CdbSession.getNumberOfInstances()` to find the number of elements in a list has, and then traverse them using a standard index notation, i.e., `<path to list>[integer]`. The children of a list are numbered starting from 0. Looking at [Example 36, “L3 VPN YANG Extract”](#) the path `/l3vpn:topology/connection[2]/endpoint-1` refers to the `endpoint-1` leaf of the third connection. This numbering is only valid during the current CDB session. CDB is always locked for writing during a read session.

We can also refer to list instances using the values of the keys of the list. In a YANG model you specify which leafs (there can be several) are to be used for keys by using the `key <name>` statement at the beginning of the list. In our case a `connection` has the `name` leaf as key. So the path `/l3vpn:topology/connection{c1}/endpoint-2` refers to the `endpoint-2` leaf of the connection whose name is “`c1`”.

A YANG list may have more than one key. The syntax for the keys is a space separated list of key values enclosed within curly brackets: `{Key1 Key2 ...}`

Which version of list element referencing to use depends on the situation. Indexing with an integer is convenient when looping through all elements. As a convenience all methods expecting keypaths accept formatting characters and accompanying data items. For example you can use `CdbSession.getElem("server[%d]/ifc{%s}/mtu", 2, "eth0")` to fetch the MTU of the third server instance's interface named "eth0". Using relative paths and `CdbSession.pushd()` it is possible to write code that can be re-used for common sub-trees.

The current position also includes the namespace. To read elements from a different namespace use the prefix qualified tag for that element like in `l3vpn:topology`.

Subscriptions

The CDB subscription mechanism allows an external program to be notified when some part of the configuration changes. When receiving a notification it is also possible to iterate through the changes written to CDB. Subscriptions are always towards the running data-store (it is not possible to subscribe to changes to the startup data-store). Subscriptions towards operational data (see [the section called “Operational Data in CDB”](#)) kept in CDB are also possible, but the mechanism is slightly different.

The first thing to do is to inform CDB which paths we want to subscribe to. Registering a path returns a subscription point identifier. This is done by acquiring an subscriber instance by

calling `CdbSubscription Cdb.newSubscription()` method. For the subscriber (or `CdbSubscription` instance) the paths are registered with the `CdbSubscription.subscribe()` that returns the actual subscription point identifier. A subscriber can have multiple subscription points, and there can be many different subscribers. Every point is defined through a path - similar to the paths we use for read operations, with the exception that instead of fully instantiated paths to list instances we can selectively use tagpaths.

When a client is done defining subscriptions it should inform NSO that it is ready to receive notifications by calling `CdbSubscription.subscribeDone()`, after which the subscription socket is ready to be polled.

We can subscribe either to specific leaves, or entire subtrees. Explaining this by example we get:

```
/ncs:devices/global-settings/trace
    Subscription to a leaf. Only changes to this leaf will generate a notification.
/ncs:devices
    Subscription to the subtree rooted at /ncs:devices. Any changes to this subtree will generate a notification. This includes additions or removals of device instances, as well as changes to already existing device instances.
/ncs:devices/device{"ex0"}/address
    Subscription to a specific element in a list. A notification will be generated when the device "ex0" changes its ip address.
/ncs:devices/device/address
    Subscription to a leaf in a list. A notification will be generated leaf address is changed in any device instance.
```

When adding a subscription point the client must also provide a priority, which is an integer (a smaller number means higher priority). When data in CDB is changed, this change is part of a transaction. A transaction can be initiated by a **commit** operation from the CLI or a **edit-config** operation in NETCONF resulting in the running database being modified. As the last part of the transaction CDB will generate notifications in lock-step priority order. First all subscribers at the lowest numbered priority are handled, once they all have replied and synchronized by calling `CdbSubscription.sync()` the next set - at the next priority level - is handled by CDB. Not until all subscription points have been acknowledged is the transaction complete. This implies that if the initiator of the transaction was for example a **commit** command in the CLI, the command will hang until notifications have been acknowledged.

Note that even though the notifications are delivered within the transaction it is not possible for a subscriber to reject the changes (since this would break the two-phase commit protocol used by the NSO backplane towards all data-providers).

As a subscriber has read its subscription notifications using `CdbSubscription.read()` it can iterate through the changes that caused the particular subscription notification using the `CdbSubscription.diffIterate()` method. It is also possible to start a new read-session to the `CdbDbType.CDB_PRE_COMMIT_RUNNING` database to read the running database as it was before the pending transaction.

To view registered subscribers use the **ncs --status** command.

Sessions

It is important to note that CDB is locked for writing during a read session using the Java API. A session starts with `CdbSession Cdb.startSession()` and the lock is not released until the

`CdbSession.endSession()` (or the `Cdb.close()`) call. CDB will also automatically release the lock if the socket is closed for some other reason, such as program termination.

Loading Initial Data Into CDB

When NSO starts for the first time, the CDB database is empty. The location of the database files used by CDB are given in `ncs.conf`. At first startup, when CDB is empty, i.e., no database files are found in the directory specified by `<db-dir> (. /ncs-cdb` as given by [Example 37, “CDB Init”](#)), CDB will try to initialize the database from all XML documents found in the same directory.

Example 37. CDB Init

```
<!-- Where the database (and init XML) files are kept -->
<cdb>
  <db-dir>./ncs-cdb</db-dir>
</cdb>
```

This feature can be used to reset the configuration to factory settings.

Given the YANG model in [Example 36, “L3 VPN YANG Extract”](#) the initial data for topology can be found in `topology.xml` as seen in [Example 38, “Initial Data for Topology”](#)

Example 38. Initial Data for Topology

```
<config xmlns="http://tail-f.com/ns/config/1.0">
  <topology xmlns="http://com/example/l3vpn">
    <role>
      <role>ce</role>
      <device>ce0</device>
      <device>ce1</device>
      <device>ce2</device>
      ...
    </role>
    <role>
      <role>pe</role>
      <device>pe0</device>
      <device>pe1</device>
      <device>pe2</device>
      <device>pe3</device>
    </role>
    ...
    <connection>
      <name>c0</name>
      <endpoint-1>
        <device>ce0</device>
        <interface>GigabitEthernet0/8</interface>
        <ip-address>192.168.1.1/30</ip-address>
      </endpoint-1>
      <endpoint-2>
        <device>pe0</device>
        <interface>GigabitEthernet0/0/0/3</interface>
        <ip-address>192.168.1.2/30</ip-address>
      </endpoint-2>
      <link-vlan>88</link-vlan>
    </connection>
    <connection>
      <name>c1</name>
      ...
    </connection>
  </topology>
</config>
```

Another example of using this features is when initializing the AAA database. This is described in Chapter 9, *The AAA infrastructure* in *NSO 5.7 Administration Guide*.

All files ending in .xml will be loaded (in an undefined order) and committed in a single transaction when CDB enters start phase 1 (see the section called “Starting NSO” in *NSO 5.7 Administration Guide* for more details on start phases). The format of the init files is rather lax in that it is not required that a complete instance document following the data-model is present, much like the NETCONF edit-config operation. It is also possible to wrap multiple top-level tags in the file with a surrounding config tag, as shown in [Example 39, “Wrapper for Multiple Top Level Tags”](#) like this:

Example 39. Wrapper for Multiple Top Level Tags

```
<config xmlns="http://tail-f.com/ns/config/1.0">
  ...
</config>
```


Note

The actual names of the XML files does not matter, i.e., they do not need to correspond to the part of the YANG model being initialized.

Operational Data in CDB

In addition to handling configuration data, CDB can also take care of operational data such as alarms and traffic statistics. By default, operational data is not persistent and thus not kept between restarts. In the YANG model annotating a node with `config false` will mark the subtree rooted at that node as operational data. Reading and writing operational data is done similar to ordinary configuration data, with the main difference being that you have to specify that you are working against operational data. Also, the subscription model is different.

Subscriptions

Subscriptions towards the operational data in CDB are similar to the above, but due to the fact that the operational data store is designed for light-weight access, and does not have transactions and normally avoids the use of any locks, there are several differences - in particular:

- Subscription notifications are only generated if the writer obtains the “subscription lock”, by using the `Cdb.startSession()` method with the `CdbLockType.LOCK_REQUEST` flag.
- Subscriptions are registered with the `CdbSubscription.subscribe()` method with the flag `CdbSubscriptionType.SUB_OPERATIONAL` rather than `CdbSubscriptionType.SUB_RUNNING`.
- No priorities are used.
- Neither the writer that generated the subscription notifications nor other writes to the same data are blocked while notifications are being delivered. However the subscription lock remains in effect until notification delivery is complete.
- The previous value for modified leaf is not available when using the `CdbSubscriber.diffIterate()` method.

Essentially a write operation towards the operational data store, combined with the subscription lock, takes on the role of a transaction for configuration data as far as subscription notifications are concerned. This means that if operational data updates are done with many single-element write operations, this can potentially result in a lot of subscription notifications. Thus it is a good idea to use the multi-element `CdbSession.setObject()` etc methods for updating operational data that applications subscribe to.

Since write operations that do not attempt to obtain the subscription lock are allowed to proceed even during notification delivery, it is the responsibility of the applications using the operational data store to obtain the lock as needed when writing. If subscribers should be able to reliably read the exact data that

resulted from the write that triggered their subscription, the subscription lock must always be obtained when writing that particular set of data elements. One possibility is of course to obtain the lock for *all* writes to operational data, but this may have an unacceptable performance impact.

Example

We will take a first look at the `examples.ncs/getting-started/developing-with-ncs/1-cdb` example. This example is a NSO project with two packages: `cdb` and `router`.

Example packages

- | | |
|--------|--|
| router | A NED package with a simple but still realistic model of a network device. The only component in this package is the NED component that uses NETCONF to communicate with the device. This package is used in many NSO examples including <code>examples.ncs/getting-started/developing-with-ncs/0-router-network</code> which is an introduction to NSO device manager, NSO netsim and this router package. |
| cdb | This package has an even simpler YANG model to illustrate some aspects of CDB data retrieval. The package consists of 5 application components: <ul style="list-style-type: none"> • <i>Plain CDB Subscriber</i> - This cdb subscriber subscribes to changes under the path <code>/devices/device{ex0}/config</code>. Whenever a change occurs there, the code iterates through the change and prints the values. • <i>CdbCfgSubscriber</i> - An more advanced CDB subscriber that subscribes to changes under the path <code>/devices/device/config/sys/interfaces/interface</code>. • <i>OperSubscriber</i> - An operational data subscribe that subscribes to changes under the path <code>t:test/stats-item</code>. |

The `cdb` package include the YANG shown in [Example 40, “1-cdb Simple Config Data”](#).

Example 40. 1-cdb Simple Config Data

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

    import tailf-common {
        prefix tailf;
    }

    description "This model is used as a simple example model
        illustrating some aspects of CDB subscriptions
        and CDB operational data";

    revision 2012-06-26 {
        description "Initial revision.";
    }

    container test {
        list config-item {
            key ckey;
            leaf ckey {
                type string;
            }
            leaf i {
                type int32;
            }
        }
        list stats-item {
```

Example

```
config false;
tailf:cdb-oper;
key skey;
leaf skey {
    type string;
}
leaf i {
    type int32;
}
container inner {
    leaf l {
        type string;
    }
}
}
```

Let us now populate the database and look at the "Plain CDB Subscriber" and how it can use the Java API to react to changes to the data. This component subscribes on changes under the path /devices/device{ex0}/config which is configuration changes for the device named "ex0" which is a device connected to NSO via the router NED.

Being an application component in the cdb package implies that this component is realized by a Java class that implements the `com.tailf.ncs.ApplicationComponent` Java interface. This interface inherits the Java standard `Runnable` interface which requires the `run()` method to be implemented. In addition to this method there is a `init()` and a `finish()` method that has to be implemented. When the NSO Java-VM starts this class will be started in a separate thread with an initial call to `init()` before thread start. When the package is requested to stop execution a call to `finish()` is performed and this method is expected to end thread execution.

Example 41. Plain CDB Subscriber Java Code

```
public class PlainCdbSub implements ApplicationComponent {
    private static Logger LOGGER = LogManager.getLogger(PlainCdbSub.class);

    @Resource(type=ResourceType.CDB, scope=Scope.INSTANCE, qualifier="plain")
    private Cdb cdb;

    private CdbSubscription sub;
    private int subId;
    private boolean requestStop = false;

    public PlainCdbSub() {
    }

    public void init() {
        try {
            LOGGER.info(" init cdb subscriber ");
            sub = new CdbSubscription(cdb);
            String str = "/devices/device{ex0}/config";
            subId = sub.subscribe(1, new Ncs(), str);
            sub.subscribeDone();
            LOGGER.info("subscribeDone");
            requestStop = false;
        } catch (Exception e) {
            throw new RuntimeException("FAIL in init", e);
        }
    }

    public void run() {
```

```

try {
    while (!requestStop) {
        try {
            int[] points = sub.read();
            sub.diffIterate(subId, new Iter());
        } finally {
            sub.sync(CdbSubscriptionSyncType.DONE_SOCKET);
        }
    }
} catch (ConfException e) {
    if (e.getErrorCode() == ErrorCode.ERR_EOF) {
        // Triggered by finish method
        // if we throw further NCS JVM will try to restart
        // the package
        LOGGER.warn(" Socket Closed!");
    } else {
        throw new RuntimeException("FAIL in run", e);
    }
} catch (Exception e) {
    LOGGER.warn("Exception:" + e.getMessage() );
    throw new RuntimeException("FAIL in run", e);
} finally {
    requestStop = false;
    LOGGER.warn(" run end ");
}
}

public void finish() {
    requestStop = true;
    LOGGER.warn(" PlainSub in finish () =>" );
    try {
        // ResourceManager will close the resource (cdb) used by this
        // instance that triggers ConfException with ErrorCode.ERR_EOF
        // in run method
        ResourceManager.unregisterResources(this);
    } catch (Exception e) {
        throw new RuntimeException("FAIL in finish", e);
    }
    LOGGER.warn(" PlainSub in finish () => ok");
}

private class Iter implements CdbDiffIterate {
    public DiffIterateResultFlag iterate(ConfObject[] kp,
                                         DiffIterateOperFlag op,
                                         ConfObject old_value,
                                         ConfObject new_value,
                                         Object state) {
        try {
            String kpString = Conf.kpToString(kp);
            LOGGER.info("diffIterate: kp= "+kpString+", OP="+op+
                       ", old_value="+old_value+", new_value="+
                       new_value);
            return DiffIterateResultFlag.ITER_RECURSE;
        } catch (Exception e) {
            return DiffIterateResultFlag.ITER_CONTINUE;
        }
    }
}
}

```

We will walk through the code and highlight different aspects. We start with how the Cdb instance is retrieved in this example. It is always possible to open a socket to NSO and create the Cdb instance

Example

with this socket. But with this comes the responsibility to manage that socket. In NSO there is a *ResourceManager* that can take over this responsibility. In the code, the field that should contain the Cdb instance is simply annotated with a @Resource annotation. The ResourceManager will find this annotation and create the Cdb instance as specified. In this example ([Example 42, “Resource Annotation”](#)) Scope.INSTANCE implies that new instances of this example class should have unique Cdb instances (see more on this in the section called “[The Resource Manager](#)”).

Example 42. Resource Annotation

```
@Resource(type=ResourceType.CDB, scope=Scope.INSTANCE, qualifier="plain")
private Cdb cdb;
```

The init() method (shown in [Example 43, “Plain Subscriber Init”](#)) is called before this application component thread is started. For this subscriber this is the place to setup the subscription. First an CdbSubscription instance is created and in this instance the subscription points are registered (one in this case). When all subscription points are registered a call to CdbSubscriber.subscribeDone() will indicate that the registration is finished and the subscriber is ready to start.

Example 43. Plain Subscriber Init

```
public void init() {
    try {
        LOGGER.info(" init cdb subscriber ");
        sub = new CdbSubscription(cdb);
        String str = "/devices/device{ex0}/config";
        subId = sub.subscribe(1, new Ncs(), str);
        sub.subscribeDone();
        LOGGER.info("subscribeDone");
        requestStop = false;
    } catch (Exception e) {
        throw new RuntimeException("FAIL in init", e);
    }
}
```

The run() method comes from the standard Java API Runnable interface and is executed when the application component thread is started. For this subscriber ([Example 44, “Plain CDB Subscriber”](#)) a loop over the CdbSubscription.read() method drives the subscription. This call will block until data has changed for some of the subscription points that was registered, and the ids for these subscription points will then be returned. In our example since we only have one subscription point we know that this id the one stored as subId. This subscriber choose to find the changes by calling the CdbSubscription.diffIterate() method. Important is to acknowledge the subscription by calling CdbSubscription.sync() or else this subscription will block the ongoing transaction.

Example 44. Plain CDB Subscriber

```
public void run() {
    try {
        while (!requestStop) {
            try {
                int[] points = sub.read();
                sub.diffIterate(subId, new Iter());
            } finally {
                sub.sync(CdbSubscriptionSyncType.DONE_SOCKET);
            }
        }
    }
```

```

        }
    } catch (ConfException e) {
        if (e.getErrorCode() == ErrorCode.ERR_EOF) {
            // Triggered by finish method
            // if we throw further NCS JVM will try to restart
            // the package
            LOGGER.warn(" Socket Closed!");
        } else {
            throw new RuntimeException("FAIL in run", e);
        }
    } catch (Exception e) {
        LOGGER.warn("Exception:" + e.getMessage());
        throw new RuntimeException("FAIL in run", e);
    } finally {
        requestStop = false;
        LOGGER.warn(" run end ");
    }
}
}

```

The call to the `CdbSubscription.diffIterate()` requires an object instance implementing an `iterate()` method. To do this the `CdbDiffIterate` interface is implemented by a suitable class. In our example this done by a private inner class called `Iter` ([Example 45, “Plain Subscriber Iterator Implementation”](#)). The `iterate()` method is called for every all changes and the path, type of change and data is provided as arguments. In the end the `iterate()` should return a flag that controls how further iteration should prolong, or if it should stop. Our example `iterate()` method just logs the changes.

Example 45. Plain Subscriber Iterator Implementation

```

private class Iter implements CdbDiffIterate {
    public DiffIterateResultFlag iterate(ConfObject[] kp,
                                         DiffIterateOpFlag op,
                                         ConfObject old_value,
                                         ConfObject new_value,
                                         Object state) {
        try {
            String kpString = Conf.kpToString(kp);
            LOGGER.info("diffIterate: kp= "+kpString+", OP="+op+
                       ", old_value="+old_value+", new_value="+
                       new_value);
            return DiffIterateResultFlag.ITER_RECURSE;
        } catch (Exception e) {
            return DiffIterateResultFlag.ITER_CONTINUE;
        }
    }
}

```

The `finish()` method ([Example 46, “Plain Subscriber finish”](#)) is called when the NSO Java-VM wants the application component thread to stop execution. An orderly stop of the thread is expected. Here the subscription will stop if the subscription socket and underlying Cdb instance is closed. This will be done by the `ResourceManager` when we tell it that the resources retrieved for this Java object instance could be unregistered and closed. This is done by a call to the `ResourceManager.unregisterResources()` method.

Example 46. Plain Subscriber finish

```
public void finish() {
```

Example

```

    requestStop = true;
    LOGGER.warn(" PlainSub in finish () =>");
    try {
        // ResourceManager will close the resource (cdb) used by this
        // instance that triggers ConfException with ErrorCode.ERR_EOF
        // in run method
        ResourceManager.unregisterResources(this);
    } catch (Exception e) {
        throw new RuntimeException("FAIL in finish", e);
    }
    LOGGER.warn(" PlainSub in finish () => ok");
}

```

We will now compile and start the 1-cdb example, populate some config data and look at the result. [Example 47, “Plain Subscriber Startup”](#) shows how to do this.

Example 47. Plain Subscriber Startup

```

$ make clean all
$ ncs-netsim start
DEVICE ex0 OK STARTED
DEVICE ex1 OK STARTED
DEVICE ex2 OK STARTED

$ ncs

```

By far the easiest way to populate the database with some actual data is to run the CLI ([Example 48, “Populate Data using CLI”](#)).

Example 48. Populate Data using CLI

```

$ ncs_cli -u admin
admin connected from 127.0.0.1 using console on ncs
admin@ncs# config exclusive
Entering configuration mode exclusive
Warning: uncommitted changes will be discarded on exit
admin@ncs(config)# devices sync-from
sync-result {
    device ex0
    result true
}
sync-result {
    device ex1
    result true
}
sync-result {
    device ex2
    result true
}

admin@ncs(config)# devices device ex0 config r:sys syslog server 4.5.6.7 enabled
admin@ncs(config-server-4.5.6.7)# commit
Commit complete.
admin@ncs(config-server-4.5.6.7)# top
admin@ncs(config)# exit
admin@ncs# show devices device ex0 config r:sys syslog
NAME
-----
4.5.6.7

```

10.3.4.5

We have now added a server to the syslog. What remains is to check what our "Plain CDB Subscriber" ApplicationComponent got as a result of this update. In the logs directory of the 1-cdb example there is a file named PlainCdbSub.out which contains the log data from this application component. In the beginning of this file a lot of logging is performed which emanates from the sync-from of the device. In the end of this file we can find the three logrows which comes from our update. See extract in [Example 49, "Plain Subscriber Output"](#) (with each row split over several to fit on the page).

Example 49. Plain Subscriber Output

```
<INFO> 05-Feb-2015::13:24:55,760 PlainCdbSub$Iter
  (cdb-examples:Plain CDB Subscriber) -Run-4: - diffIterate:
  kp= /ncs:devices/device{ex0}/config/r:sys/syslog/server{4.5.6.7},
  OP=MOP_CREATED, old_value=null, new_value=null
<INFO> 05-Feb-2015::13:24:55,761 PlainCdbSub$Iter
  (cdb-examples:Plain CDB Subscriber) -Run-4: - diffIterate:
  kp= /ncs:devices/device{ex0}/config/r:sys/syslog/server{4.5.6.7}/name,
  OP=MOP_VALUE_SET, old_value=null, new_value=4.5.6.7
<INFO> 05-Feb-2015::13:24:55,762 PlainCdbSub$Iter
  (cdb-examples:Plain CDB Subscriber) -Run-4: - diffIterate:
  kp= /ncs:devices/device{ex0}/config/r:sys/syslog/server{4.5.6.7}/enabled,
  OP=MOP_VALUE_SET, old_value=null, new_value=true
```

We will turn to look at another subscriber which has a more elaborate diff iteration method. In our example *cdb* package we have an application component named "CdbCfgSubscriber". This component consists of a subscriber for the subscription point */ncs:devices/device/config/r:sys/interfaces/interface*. The iterate() method is here implemented as an inner class called DiffIterateImpl.

The code for this subscriber is left out but can be found in the file ConfigCdbSub.java.

[Example 50, "Run CdbCfgSubscriber Example"](#) shows how to build and run the example.

Example 50. Run CdbCfgSubscriber Example

```
$ make clean all
$ ncs-netsim start
DEVICE ex0 OK STARTED
DEVICE ex1 OK STARTED
DEVICE ex2 OK STARTED

$ ncs

$ ncs_cli -u admin
admin@ncs# devices sync-from suppress-positive-result
admin@ncs# config
admin@ncs(config)# no devices device ex* config r:sys interfaces
admin@ncs(config)# devices device ex0 config r:sys interfaces \
> interface en0 mac 3c:07:54:71:13:09 mtu 1500 duplex half unit 0 family inet \
> address 192.168.1.115 broadcast 192.168.1.255 prefix-length 32
admin@ncs(config-address-192.168.1.115)# commit
Commit complete.
admin@ncs(config-address-192.168.1.115)# top
admin@ncs(config)# exit
```

If we look in the file logs/ConfigCdbSub.out we will find logrecords from the subscriber ([Example 51, "Subscriber Output"](#)). In the end of this file the last DUMP DB will show only one remaining interface.

Example 51. Subscriber Output

```

...
<INFO> 05-Feb-2015::16:10:23,346 ConfigCdbSub
  (cdb-examples:CdbCfgSubscriber)-Run-1: - Device {ex0}
<INFO> 05-Feb-2015::16:10:23,346 ConfigCdbSub
  (cdb-examples:CdbCfgSubscriber)-Run-1: - INTERFACE
<INFO> 05-Feb-2015::16:10:23,346 ConfigCdbSub
  (cdb-examples:CdbCfgSubscriber)-Run-1: - name: {en0}
<INFO> 05-Feb-2015::16:10:23,346 ConfigCdbSub
  (cdb-examples:CdbCfgSubscriber)-Run-1: - description:null
<INFO> 05-Feb-2015::16:10:23,350 ConfigCdbSub
  (cdb-examples:CdbCfgSubscriber)-Run-1: - speed:null
<INFO> 05-Feb-2015::16:10:23,354 ConfigCdbSub
  (cdb-examples:CdbCfgSubscriber)-Run-1: - duplex:half
<INFO> 05-Feb-2015::16:10:23,354 ConfigCdbSub
  (cdb-examples:CdbCfgSubscriber)-Run-1: - mtu:1500
<INFO> 05-Feb-2015::16:10:23,354 ConfigCdbSub
  (cdb-examples:CdbCfgSubscriber)-Run-1: - mac:<<60,7,84,113,19,9>>
<INFO> 05-Feb-2015::16:10:23,354 ConfigCdbSub
  (cdb-examples:CdbCfgSubscriber)-Run-1: - UNIT
    name: {0}
    description: null
    vlan-id:null
<INFO> 05-Feb-2015::16:10:23,355 ConfigCdbSub
  (cdb-examples:CdbCfgSubscriber)-Run-1: - ADDRESS-FAMILY
    key: {192.168.1.115}
    prefixLength: 32
    broadCast:192.168.1.255
<INFO> 05-Feb-2015::16:10:23,356 ConfigCdbSub
  (cdb-examples:CdbCfgSubscriber)-Run-1: - Device {ex1}
<INFO> 05-Feb-2015::16:10:23,356 ConfigCdbSub
  (cdb-examples:CdbCfgSubscriber)-Run-1: - Device {ex2}

```

Operational Data

We will look once again at the YANG model for the cdb package in the examples.ncs/getting-started/developing-with-ncs/1-cdb example. Inside the test.yang YANG model there is a test container. As a child to this container there is a list stats-item (see [Example 52, “1-cdb Simple Operational Data”](#))

Example 52. 1-cdb Simple Operational Data

```

list stats-item {
  config false;
  tailf:cdb-oper;
  key skey;
  leaf skey {
    type string;
  }
  leaf i {
    type int32;
  }
  container inner {
    leaf l {

```

```
        type string;  
    }  
}
```

Note the list `stats-item` has the substatement `config false`; and below it we find a `tailf:cdb-oper`; statement. A standard way to implement operational data is to define a callpoint in the YANG model and write instrumentation callback methods for retrieval of the operational data (see more on data callbacks in [the section called “DP API”](#)). Here on the other hand we use the `tailf:cdb-oper`; statement which implies that these instrumentation callbacks are automatically provided internally by NSO. The downside is that we must populate this operational data in CDB from the outside.

An example of Java code that create operational data using the Navu API is shown in [Example 53, “Creating Operational Data using Navu API”](#).

Example 53. Creating Operational Data using Navu API

```

public static void createEntry(String key)
throws IOException, NavuException, ConfException {

    Socket socket = new Socket("127.0.0.1", Conf.NCS_PORT);
    Maapi maapi = new Maapi(socket);
    maapi.startUserSession("system", InetAddress.getByName(null),
                          "system", new String[] {}, 
                          MaapiUserSessionFlag.PROTO_TCP);
    NavuContext operContext = new NavuContext(maapi);
    int th = operContext.startOperationalTrans(Conf.MODE_READ_WRITE);
    NavuContainer mroot = new NavuContainer(operContext);
    LOGGER.debug("ROOT --> " + mroot);

    ConfNamespace ns = new test();
    NavuContainer testModule = mroot.container(ns.hash());
    NavuList list = testModule.container("test").list("stats-item");
    LOGGER.debug("LIST: --> " + list);

    List<ConfXMLParam> param = new ArrayList<ConfXMLParam>();
    param.add(new ConfXMLParamValue(ns, "skey", new ConfBuf(key)));
    param.add(new ConfXMLParamValue(ns, "i", new ConfInt32(key.hashCode())));
    param.add(new ConfXMLParamStart(ns, "inner"));
    param.add(new ConfXMLParamValue(ns, "l", new ConfBuf("test-"+key)));
    param.add(new ConfXMLParamStop (ns, "inner"));
    list.setValues(param.toArray(new ConfXMLParam[0]));
    maapi.applyTrans(th, false);
    maapi.finishTrans(th);
    maapi.endUserSession();
    socket.close();
}

```

An example of Java code that delete operational data using the CDB API is shown in [Example 54, “Deleting Operational Data using CDB API”](#).

Example 54. Deleting Operational Data using CDB API

```
public static void deleteEntry(String key)
    throws IOException, NavuException, ConfException {
    Socket s = new Socket("127.0.0.1", Conf.NCS_PORT);
    Cdb c = new Cdb("writer", s);
```

```

        CdbSession sess = c.startSession(CdbDBType.CDB_OPERATIONAL,
                                         EnumSet.of(CdbLockType.LOCK_REQUEST,
                                                    CdbLockType.LOCK_WAIT));
        ConfPath path = new ConfPath("/t:test/stats-item%{xx}",
                                     new ConfKey(new ConfBuf(key)));
        sess.delete(path);
        sess.endSession();
        s.close();
    }
}

```

In the 1-cdb example in the cdb package there is also a application component with a operational data subscriber that subscribes on data from the path "/t:test/stats-item" (see [Example 55, "CDB Operational Subscriber Java code"](#)).

Example 55. CDB Operational Subscriber Java code

```

public class OperCdbSub implements ApplicationComponent, CdbDiffIterate {
    private static Logger LOGGER = LogManager.getLogger(OperCdbSub.class);

    // let our ResourceManager inject Cdb sockets to us
    // no explicit creation of creating and opening sockets needed
    @Resource(type=ResourceType.CDB, scope=Scope.INSTANCE, qualifier="sub-sock")
    private Cdb cdbSub;
    @Resource(type=ResourceType.CDB, scope=Scope.INSTANCE, qualifier="data-sock")
    private Cdb cdbData;

    private boolean requestStop = false;
    private int point;
    private CdbSubscription cdbSubscription;

    public OperCdbSub() {
    }

    public void init() {
        LOGGER.info(" init oper subscriber ");
        try {
            cdbSubscription = cdbSub.newSubscription();
            String path = "/t:test/stats-item";
            point =
                cdbSubscription.subscribe(CdbSubscriptionType.SUB_OPERATIONAL,
                                         1, test.hash, path);
            cdbSubscription.subscribeDone();
            LOGGER.info("subscribeDone");
            requestStop = false;
        } catch (Exception e) {
            LOGGER.error("Fail in init",e);
        }
    }

    public void run() {
        try {
            while (!requestStop) {
                try {
                    int[] point = cdbSubscription.read();
                    CdbSession cdbSession =
                        cdbData.startSession(CdbDBType.CDB_OPERATIONAL);
                    EnumSet<DiffIterateFlags> diffFlags =
                        EnumSet.of(DiffIterateFlags.ITER_WANT_PREV);
                    cdbSubscription.diffIterate(point[0], this, diffFlags,

```

```

                cdbSession);
        cdbSession.endSession();
    } finally {
        cdbSubscription.sync(
            CdbSubscriptionSyncType.DONE_OPERATIONAL);
    }
}
} catch (Exception e) {
    LOGGER.error("Fail in run shouldrun", e);
}
requestStop = false;
}

public void finish() {
    requestStop = true;
    try {
        ResourceManager.unregisterResources(this);
    } catch (Exception e) {
        LOGGER.error("Fail in finish", e);
    }
}

@Override
public DiffIterateResultFlag iterate(ConfObject[] kp,
                                     DiffIterateOperFlag op,
                                     ConfObject oldValue,
                                     ConfObject newValue,
                                     Object initstate) {
    LOGGER.info(op + " " + Arrays.toString(kp) + " value: " + newValue);
    switch (op) {
        case MOP_DELETED:
            break;
        case MOP_CREATED:
        case MOP_MODIFIED: {
            break;
        }
    }
    return DiffIterateResultFlag.ITER_RECURSE;
}
}
}

```

Notice that the `CdbOperSubscriber` is very similar to the `CdbConfigSubscriber` described earlier.

In the `1-cdb` examples there are two shellscripts `setoper` and `deoper` that will execute the above `CreateEntry()` and `DeleteEntry()` respectively. We can use these to populate the operational data in CDB for the `test.yang` YANG model (see [Example 56, “Populating Operational Data”](#)).

Example 56. Populating Operational Data

```

$ make clean all
$ ncs
$ ./setoper eth0
$ ./setoper ethX
$ ./deoper ethX
$ ncs_cli -u admin

admin@ncs# show test
SKEY  I      L
-----
```

```
eth0 3123639 test-eth0
```

And if we look at the output from the "CDB Operational Subscriber" that is found in the logs/OperCdbSub.out we will see output similar to [Example 57, "Operational subscription Output"](#).

Example 57. Operational subscription Output

```
<INFO> 05-Feb-2015::16:27:46,583 OperCdbSub
(cdb-examples:OperSubscriber)-Run-0:
- MOP_CREATED [{eth0}, t:stats-item, t:test] value: null
<INFO> 05-Feb-2015::16:27:46,584 OperCdbSub
(cdb-examples:OperSubscriber)-Run-0:
- MOP_VALUE_SET [t:skey, {eth0}, t:stats-item, t:test] value: eth0
<INFO> 05-Feb-2015::16:27:46,584 OperCdbSub
(cdb-examples:OperSubscriber)-Run-0:
- MOP_VALUE_SET [t:l, t:inner, {eth0}, t:stats-item, t:test] value: test-eth0
<INFO> 05-Feb-2015::16:27:46,585 OperCdbSub
(cdb-examples:OperSubscriber)-Run-0:
- MOP_VALUE_SET [t:i, {eth0}, t:stats-item, t:test] value: 3123639
<INFO> 05-Feb-2015::16:27:52,429 OperCdbSub
(cdb-examples:OperSubscriber)-Run-0:
- MOP_CREATED [{ethX}, t:stats-item, t:test] value: null
<INFO> 05-Feb-2015::16:27:52,430 OperCdbSub
(cdb-examples:OperSubscriber)-Run-0:
- MOP_VALUE_SET [t:skey, {ethX}, t:stats-item, t:test] value: ethX
<INFO> 05-Feb-2015::16:27:52,430 OperCdbSub
(cdb-examples:OperSubscriber)-Run-0:
- MOP_VALUE_SET [t:l, t:inner, {ethX}, t:stats-item, t:test] value: test-ethX
<INFO> 05-Feb-2015::16:27:52,431 OperCdbSub
(cdb-examples:OperSubscriber)-Run-0:
- MOP_VALUE_SET [t:i, {ethX}, t:stats-item, t:test] value: 3123679
<INFO> 05-Feb-2015::16:28:00,669 OperCdbSub
(cdb-examples:OperSubscriber)-Run-0:
- MOP_DELETED [{ethX}, t:stats-item, t:test] value: null
```

Automatic Schema Upgrades and Downgrades

Software upgrades and downgrades represent one of the main problems of managing configuration data of network devices. Each software release for a network device is typically associated with a certain version of configuration data layout, i.e., a schema. In NSO the schema is the data model stored in the .fxs files. Once CDB has initialized it also stores a copy of the schema associated with the data it holds.

Every time NSO starts, CDB will check the current contents of the .fxs files with its own copy of the schema files. If CDB detects any changes in the schema it initiates an upgrade transaction. In the simplest case CDB automatically resolves the changes and commits the new data before NSO reaches start-phase one.

The CDB upgrade can be followed by checking the devel.log. The development log is meant to be used as support while the application is developed. It is enabled in ncs.conf as shown in [Example 58, "Enabling Developer Logging"](#)

Example 58. Enabling Developer Logging

```
<developer-log>
<enabled>true</enabled>
<file>
<name>./logs-devel.log</name>
<enabled>true</enabled>
```

```

</file>
<syslog>
  <enabled>true</enabled>
</syslog>
</developer-log>
<developer-log-level>trace</developer-log-level>

```

CDB can automatically handle the following changes to the schema:

Deleted elements

When an element is deleted from the schema, CDB simply deletes it (and any children) from the database.

Added elements

If a new element is added to the schema it needs to either be optional, dynamic, or have a default value. New elements with a default are added set to their default value. New dynamic or optional elements are simply noted as a schema change.

Re-ordering elements

An element with the same name, but in a different position on the same level, is considered to be the same element. If its type hasn't changed it will retain its value, but if the type has changed it will be upgraded as described below.

Type changes

If a leaf is still present but its type has changed, automatic coercions are performed, so for example integers may be transformed to their string representation if the type changed from e.g. int32 to string. Automatic type conversion succeeds as long as the string representation of the current value can be parsed into its new type. (Which of course also implies that a change from a smaller integer type, e.g. int8, to a larger type, e.g. int32, succeeds for any value - while the opposite will not hold, but might!) If the coercion fails, any supplied default value will be used. If no default value is present in the new schema the *automatic* upgrade will fail.

Type changes when user-defined types are used are also handled automatically, provided that some straightforward rules are followed for the type definitions. Read more about user-defined types in the `confd_types(3)` manual page, which also describes these rules.

Hash changes

When a hash value of particular element has changed (due to an addition of, or a change to, a `tailf:id-value` statement) CDB will update that element.

Key changes

When a key of a list is modified, CDB tries to upgrade the key using the same rules as explained above for adding, deleting, re-ordering, change of type, and change of hash value. If automatic upgrade of a key fails the entire list entry will be deleted.

When individual entries upgrade successfully, but results in an invalid list, all list entries will be deleted. This can happen, e.g., when an upgrade removes a leaf from the key, resulting in several entries having the same key.

Default values

If a leaf has a default value, which has not been changed from its default, then the automatic upgrade will use the new default value (if any). If the leaf value has been changed from the old default, then that value will be kept.

Adding / Removing namespaces

If a namespace no longer is present after an upgrade, CDB removes all data in that namespace. When CDB detects a new namespace, it is initialized with default values.

Changing to/from operational

Elements that previously had `config false` set that are changed into database elements will be treated as added elements. In the opposite case, where data elements in the new data model are tagged with `config false`, the elements will be deleted from the database.

Callpoint changes

CDB only considers the part of the data model in YANG modules that do not have external data callpoints. But while upgrading, CDB does handle moving subtrees into CDB from a callpoint and vice versa. CDB simply considers these as added and deleted schema elements.

Thus an application can be developed using CDB in the first development cycle. When the external database component is ready it can easily replace CDB without changing the schema.

Should the *automatic* upgrade fail, exit codes and log-entries will indicate the reason (see the section called “Disaster management” in *NSO 5.7 Administration Guide*).

Using Initialization Files for Upgrade

As described earlier, when NSO starts with an empty CDB database, CDB will load all instantiated XML documents found in the CDB directory and use these to initialize the database. We can also use this mechanism for CDB upgrade, since CDB will again look for files in the CDB directory ending in `.xml` when doing an upgrade.

This allows for handling many of the cases that the automatic upgrade can not do by itself, e.g., addition of mandatory leaves (without default statements), or multiple instances of new dynamic containers. Most of the time we can probably simply use the XML init file that is appropriate for a fresh install of the new version also for the upgrade from a previous version.

When using XML files for initialization of CDB, the complete contents of the files is used. On upgrade however, doing this could lead to modification of the user's existing configuration - e.g., we could end up resetting data that the user has modified since CDB was first initialized. For this reason two restrictions are applied when loading the XML files on upgrade:

- Only data for elements that are new as of the upgrade, i.e., elements that did not exist in the previous schema, will be considered.
- The data will only be loaded if all old, i.e., previously existing, optional/dynamic parent elements and instances exist in the current configuration.

To clarify this, let's make up the following example. Some `ServerManager` package was developed and delivered. It was realized that the data model had a serious shortcoming in that there was no way to specify the protocol to use, TCP or UDP. To fix this, in a new version of the package, another leaf was added to the `/servers/server` list, and the new YANG module can be seen in [Example 59, “New YANG module for the ServerManager Package”](#).

Example 59. New YANG module for the ServerManager Package

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

    import ietf-inet-types {
        prefix inet;
    }

    revision "2007-06-01" {
        description "added protocol.";
    }
}
```

The differences from the earlier version of the YANG module can be seen in [Example 60, “Difference between YANG Modules”](#).

Example 60. Difference between YANG Modules

```
diff ./servers1.5.yang ./servers1.4.yang

9,12d8
<     revision "2007-06-01" {
<         description "added protocol.";
<     }
<
31,37d26
<             mandatory true;
<         }
<         leaf protocol {
<             type enumeration {
<                 enum tcp;
<                 enum udp;
<             }

```

Since it was considered important that the user explicitly specified the protocol, the new leaf was made mandatory. The XML init file must include this leaf, and the result can be seen in [Example 61, “Protocol Upgrade Init File”](#) like this:

Example 61. Protocol Upgrade Init File

```
<servers:servers xmlns:servers="http://example.com/ns/servers">
```

```

<servers:server>
  <servers:name>www</servers:name>
  <servers:ip>192.168.3.4</servers:ip>
  <servers:port>88</servers:port>
  <servers:protocol>tcp</servers:protocol>
</servers:server>
<servers:server>
  <servers:name>www2</servers:name>
  <servers:ip>192.168.3.5</servers:ip>
  <servers:port>80</servers:port>
  <servers:protocol>tcp</servers:protocol>
</servers:server>
<servers:server>
  <servers:name>smtp</servers:name>
  <servers:ip>192.168.3.4</servers:ip>
  <servers:port>25</servers:port>
  <servers:protocol>tcp</servers:protocol>
</servers:server>
<servers:server>
  <servers:name>dns</servers:name>
  <servers:ip>192.168.3.5</servers:ip>
  <servers:port>53</servers:port>
  <servers:protocol>udp</servers:protocol>
</servers:server>
</servers:servers>

```

We can then just use this new init file for the upgrade, and the existing server instances in the user's configuration will get the new /servers/server/protocol leaf filled in as expected. However some users may have deleted some of the original servers from their configuration, and in those cases we obviously do not want those servers to get re-created during the upgrade just because they are present in the XML file - the above restrictions make sure that this does not happen. The configuration after the upgrade can be seen in [Example 62, "Configuration after Upgrade"](#). Here is what the configuration looks like after upgrade if the "smtp" server has been deleted before upgrade:

Example 62. Configuration after Upgrade

```

<servers xmlns="http://example.com/ns/servers">
  <server>
    <name>dns</name>
    <ip>192.168.3.5</ip>
    <port>53</port>
    <protocol>udp</protocol>
  </server>
  <server>
    <name>www</name>
    <ip>192.168.3.4</ip>
    <port>88</port>
    <protocol>tcp</protocol>
  </server>
  <server>
    <name>www2</name>
    <ip>192.168.3.5</ip>
    <port>80</port>
    <protocol>tcp</protocol>
  </server>
</servers>

```

This example also implicitly shows a limitation with this method. If the user has created additional servers, the new XML file will not specify what protocol to use for those servers, and the upgrade cannot succeed.

unless the package upgrade component method is used, see below. However the example is a bit contrived. In practice this limitation is rarely a problem. It does not occur for new lists or optional elements, nor for new mandatory elements that are not children of old lists. And in fact correctly adding this "protocol" leaf for user-created servers would require user input - it can not be done by *any* fully automated procedure.

**Note**

Since CDB will attempt to load all *.xml files in the CDB directory at the time of upgrade, it is important to not leave XML init files from a previous version that are no longer valid there.

It is always possible to write an package specific upgrade component to change the data belonging to a package before the upgrade transaction is committed. This will be explained in the following section.

Writing an Upgrade Package Component

In previous sections we showed how automatic upgrades and XML initialization files can help in upgrading CDB when YANG models have changed. In some situations this is not sufficient. For instance if a YANG model is changed and new mandatory leaves are introduced that need calculations to set the values then a programmatic upgrade is needed. This is when the *upgrade* component of a package comes in play.

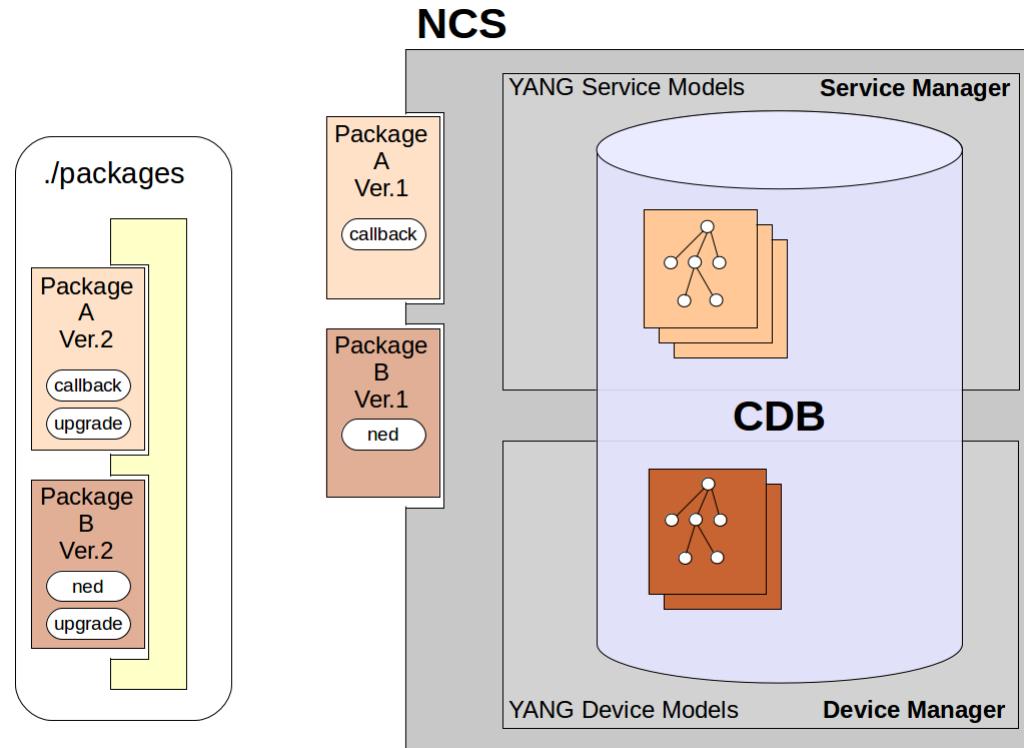
A *upgrade* component is a Java class with a standard `main()` method that becomes a standalone program that is run as part of the package *reload* action.

As with any package component types, the *upgrade* components has to be defined in the *package-metadata.xml* file for the package ([Example 63, “Upgrade Package Components”](#)).

Example 63. Upgrade Package Components

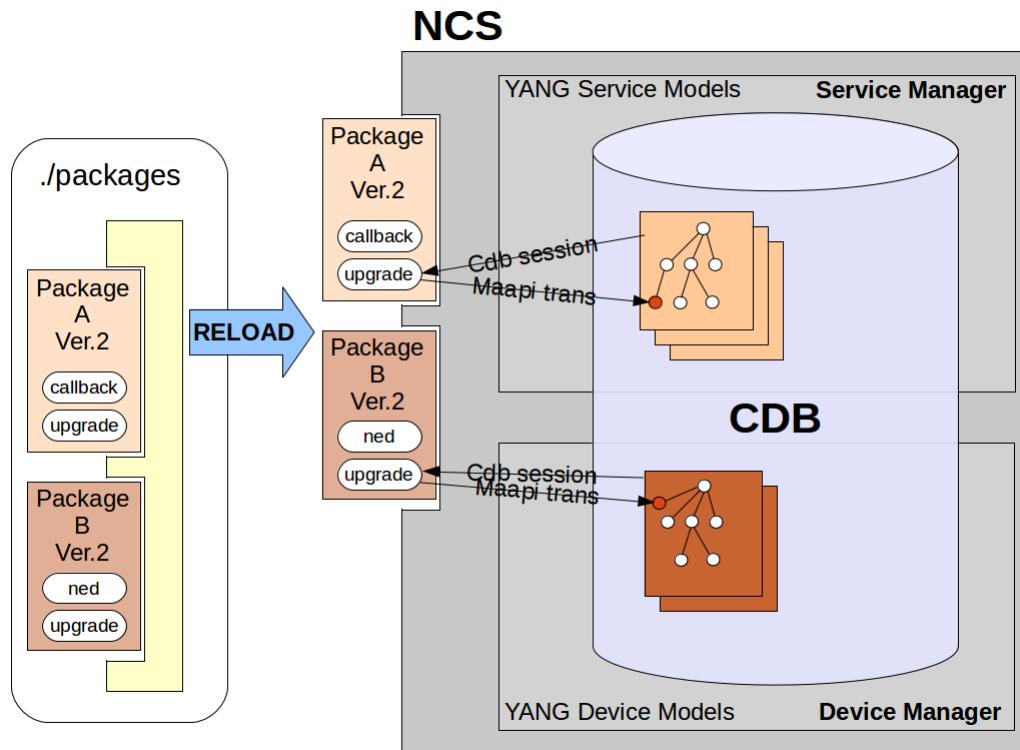
```
<ncs-package xmlns="http://tail-f.com/ns/ncs-packages">
    ...
    <component>
        <name>do-upgrade</name>
        <upgrade>
            <java-class-name>com.example.DoUpgrade</java-class-name>
        </upgrade>
    </component>
</ncs-package>
```

Lets recapitulate how packages are loaded and reloaded. NSO can search the `/ncs-config/load-path` for packages to run and will copy these to a private directory tree under `/ncs-config/state-dir` with root directory `packages-in-use.cur`. However NSO will only do this search when `packages-in-use.cur` is empty or when a *reload* is requested. This scheme makes package upgrades controlled and predictable, for more on this see the section called “[Loading Packages](#)”.



NSO Package before reload

So in preparation for a package upgrade the new packages replaces the old ones in the load path. In our scenario the YANG model changes are such that the automatic schema upgrade that CDB performs are not sufficient, therefore the new packages also contain *upgrade* components. At this point NSO is still running with the old package definitions.



NSO Package at reload

When the package reload is requested the packages in the load-path is copied to the state directory. The old state directory is scratched so that packages than no longer exist in the load path are removed and new packages are added. Obviously, unchanged packages will be unchanged. Automatic schema CDB upgrades will be performed, and afterwards, for all packages which have an upgrade component and for which at least one YANG model was changes, this upgrade component will be executed. Also for added packages that have an upgrade component this component will be executed. Hence the upgrade component needs to be programmed in such a way that care is taken for both the *new* and *upgrade* package scenarios.

So how should an upgrade component be implemented? In the previous section we described how CDB can perform an automatic upgrade. But this means that CDB has deleted all values that are no longer part of the schema? Well, not quite yet. At an initial phase of the NSO startup procedure (called start-phase0) it is possible to use all the CDB Java API calls to access the data using the schema from the database as it looked *before* the automatic upgrade. That is, the *complete* database as it stood before the upgrade is still available to the application. It is under this condition that the upgrade components are executed and is the reason why they are standalone programs and not executed by the NSO Java-VM as all other java code for components are.

So the CDB Java API can be used to read data defined by the old YANG models. To write new config data Maapi has a specific method `Maapi.attachInit()`. This method attaches a Maapi instance to the upgrade transaction (or init transaction) during phase0. This special upgrade transaction is only available during phase0. NSO will commit this transaction when the phase0 is ended, so the user should only write config data (not attempt commit etc).

We take a look at the example `$NCS_DIR/examples.ncs/getting-started/developing-with-ncs/14-upgrade-service` to see how a upgrade component can be implemented. Here the

vlan package has an original version which is replaced with a version *vlan_v2*. See the README and play with example to get aquainted.

**Note**

The *14-upgrade-service* is a *service* package. But the upgrade components here described work equally well and in the same way for any package type. The only requirement is that the package contain at least one YANG model for the upgrade component to have meaning. If not the upgrade component will never be executed.

The complete YANG model for the version 2 of the VLAN service looks as follows:

Example 64. VLAN Service v2 YANG Model

```
module vlan-service {
    namespace "http://example.com/vlan-service";
    prefix vl;

    import tailf-common {
        prefix tailf;
    }
    import tailf-ncs {
        prefix ncs;
    }

    description
        "This service creates a vlan iface/unit on all routers in our network. ";

    revision 2013-08-30 {
        description
            "Added mandatory leaf global-id.";
    }
    revision 2013-01-08 {
        description
            "Initial revision.";
    }

    augment /ncs:services {
        list vlan {
            key name;
            leaf name {
                tailf:info "Unique service id";
                tailf:cli-allow-range;
                type string;
            }

            uses ncs:service-data;
            ncs:servicepoint vlanspnt_v2;

            tailf:action self-test {
                tailf:info "Perform self-test of the service";
                tailf:actionpoint vlanselftest;
                output {
                    leaf success {
                        type boolean;
                    }
                    leaf message {
                        type string;
                        description
                            "Free format message.";
                    }
                }
            }
        }
    }
}
```

```

        }
    }

leaf global-id {
    type string;
    mandatory true;
}
leaf iface {
    type string;
    mandatory true;
}
leaf unit {
    type int32;
    mandatory true;
}
leaf vid {
    type uint16;
    mandatory true;
}
leaf description {
    type string;
    mandatory true;
}
}
}
}

```

If we diff the changes between the two YANG models for the service we see that in version 2 a new mandatory leaf has been added (see [Example 65, “YANG Service diff”](#)).

Example 65. YANG Service diff

```
$ diff vlan/src/yang/vlan-service.yang \
                      vlan_v2/src/yang/vlan-service.yang
16a18,22
>     revision 2013-08-30 {
>         description
>             "Added mandatory leaf global-id.";
>     }
>
48a55,58
>     leaf global-id {
>         type string;
>         mandatory true;
>     }
68c78
```

We need to create a Java class with a `main()` method that connects to CDB and MAAPI. This main will be executed as a separate program and all private and shared jars defined by the package will be in the classpath. To upgrade the vlan service the following Java code is needed:

Example 66. VLAN Service Upgrade Component Java Class

```
public class UpgradeService {

    public UpgradeService() {
    }
```

```

public static void main(String[] args) throws Exception {
    Socket s1 = new Socket("localhost", Conf.NCS_PORT);
    Cdb cdb = new Cdb("cdb-upgrade-sock", s1);
    cdb.setUseForCdbUpgrade();
    CdbUpgradeSession cdbsess =
        cdb.startUpgradeSession(
            CdbDBType.CDB_RUNNING,
            EnumSet.of(CdbLockType.LOCK_SESSION,
                       CdbLockType.LOCK_WAIT));
}

Socket s2 = new Socket("localhost", Conf.NCS_PORT);
Maapi maapi = new Maapi(s2);
int th = maapi.attachInit();

int no = cdbsess.getNumberOfInstances("/services/vlan");
for(int i = 0; i < no; i++) {
    Integer offset = Integer.valueOf(i);
    ConfBuf name = (ConfBuf)cdbsess.getElem("/services/vlan[%d]/name",
                                              offset);
    ConfBuf iface = (ConfBuf)cdbsess.getElem("/services/vlan[%d]/iface",
                                              offset);
    ConfInt32 unit =
        (ConfInt32)cdbsess.getElem("/services/vlan[%d]/unit",
                                   offset);
    ConfUInt16 vid =
        (ConfUInt16)cdbsess.getElem("/services/vlan[%d]/vid",
                                   offset);

    String nameStr = name.toString();
    System.out.println("SERVICENAME = " + nameStr);

    String globId = String.format("%1$s-%2$s-%3$s", iface.toString(),
                                  unit.toString(), vid.toString());
    ConfPath gidpath = new ConfPath("/services/vlan[%s]/global-id",
                                    name.toString());
    maapi.setElem(th, new ConfBuf(globId), gidpath);
}

s1.close();
s2.close();
}

```

Lets go through the code and point out the different aspects of writing a upgrade component. First (see [Example 67, “Upgrade Init”](#)) we open a socket and connect to NSO. We pass this socket to a Java API Cdb instance and call `Cdb.setUseForCdbUpgrade()`. This method will prepare cdb sessions for reading old data from the CDB database, and it should only be called in this context. In the end of this first code fragment we start the CDB upgrade session:

Example 67. Upgrade Init

```
Socket s1 = new Socket("localhost", Conf.NCS_PORT);
Cdb cdb = new Cdb("cdb-upgrade-sock", s1);
cdb.setUseForCdbUpgrade();
CdbUpgradeSession cdbsess =
    cdb.startUpgradeSession(
        CdbDBType.CDB_RUNNING,
        EnumSet.of(CdbLockType.LOCK_SESSION,
```

```
CdbLockType.LOCK_WAIT));
```

We then open and connect a second socket to NSO and pass this to a Java API Maapi instance. We call the `Maapi.attachInit()` method to get the init transaction ([Example 68, “Upgrade Get Transaction”](#)).

Example 68. Upgrade Get Transaction

```
Socket s2 = new Socket("localhost", Conf.NCS_PORT);
Maapi maapi = new Maapi(s2);
int th = maapi.attachInit();
```

Using the `CdbSession` instance we read the number of service instance that exists in the CDB database. We will work on all these instances. Also if the number of instances is zero the loop will not be entered. This is a simple way to prevent the upgrade component from doing any harm in the case of this being a new package that is added to NSO for the first time:

```
int no = cdbsess.getNumberOfInstances("/services/vlan");
for(int i = 0; i < no; i++) {
```

Via the `CdbUpgradeSession`, the old service data is retrieved:

```
ConfBuf name = (ConfBuf)cdbsess.getElement("/services/vlan[%d]/name",
                                             offset);
ConfBuf iface = (ConfBuf)cdbsess.getElement("/services/vlan[%d]/iface",
                                             offset);
ConfInt32 unit =
    (ConfInt32)cdbsess.getElement("/services/vlan[%d]/unit",
                                 offset);
ConfUInt16 vid =
    (ConfUInt16)cdbsess.getElement("/services/vlan[%d]/vid",
                                 offset);
```

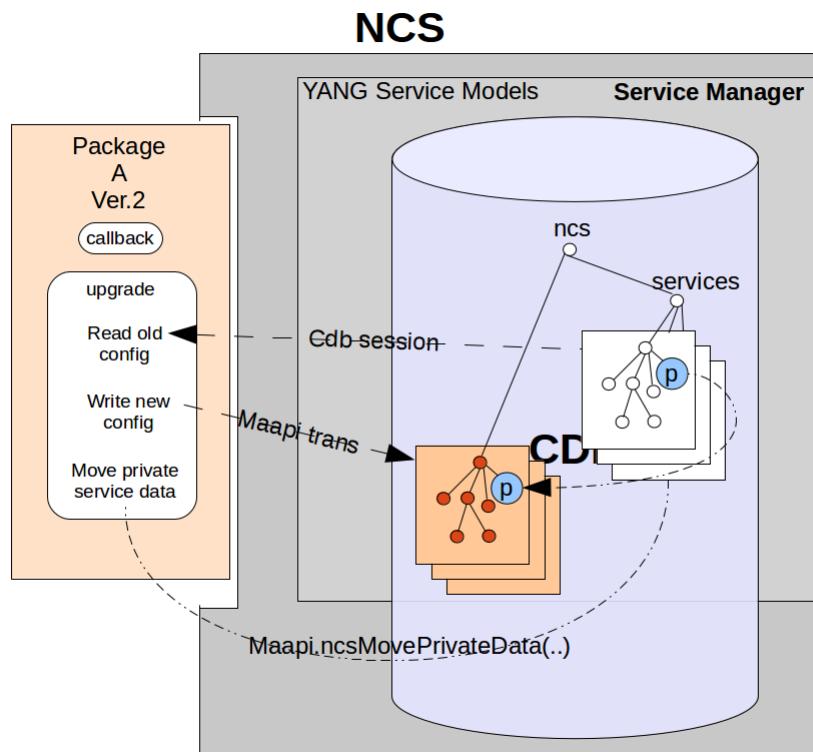
The value for new leaf introduced in the new version of the YANG model is calculated, and the value is set using Maapi and the init transaction:

```
String globId = String.format("%1$s-%2$s-%3$s", iface.toString(),
                               unit.toString(), vid.toString());
ConfPath gidpath = new ConfPath("/services/vlan[%s]/global-id",
                                name.toString());
maapi.setElement(th, new ConfBuf(globId), gidpath);
```

In the end of the program the sockets are closed. Important to note is that no commits or other handling of the init transaction is done. This is NSO responsibility:

```
s1.close();
s2.close();
```

More complicated service package upgrades scenarios occur when a YANG model containing a service point is renamed, or moved and augmented to a new place in the NSO model. This is because, not only, does the complete config data set need to be recreated on the new position but a service also have hidden private data that is part of the FASTMAP algorithm and necessary for the service to be valid. For this reason a specific MAAPi method `Maapi.ncsMovePrivateData()` exists that takes the both the old and the new position for the service point and moves the service data between these positions.



NSO Advanced service upgrade

In the 14-upgrade-service example this more complicated scenario is illustrated with the tunnel package. The tunnel package YANG model maps the vlan_v2 package one-to-one but is a complete rename of the model containers and all leafs:

Example 69. Tunnel Service YANG model

```
module tunnel-service {
    namespace "http://example.com/tunnel-service";
    prefix tl;

    import tailf-common {
        prefix tailf;
    }
    import tailf-ncs {
        prefix ncs;
    }

    description
        "This service creates a tunnel assembly on all routers in our network. ";

    revision 2013-01-08 {
        description
            "Initial revision.";
    }

    augment /ncs:services {
        list tunnel {
            key tunnel-name;
            leaf tunnel-name {
```

To upgrade from the `vlan_v2` to the `tunnel` package an new upgrade component for the `tunnel` package has to be implemented:

Example 70. Tunnel Service Upgrade Java class

```
public class UpgradeService {  
  
    public UpgradeService() {  
    }  
  
    public static void main(String[] args) throws Exception {  
        ArrayList<ConfNamespace> nsList = new ArrayList<ConfNamespace>();  
        nsList.add(new vlanService());  
        Socket s1 = new Socket("localhost", Conf.NCS_PORT);  
        Cdb cdb = new Cdb("cdb-upgrade-sock", s1);
```

```

        cdb.setUseForCdbUpgrade(nsList);
        CdbUpgradeSession cdbsess =
            cdb.startUpgradeSession(
                CdbDbType.CDB_RUNNING,
                EnumSet.of(CdbLockType.LOCK_SESSION,
                           CdbLockType.LOCK_WAIT));

        Socket s2 = new Socket("localhost", Conf.NCS_PORT);
        Maapi maapi = new Maapi(s2);
        int th = maapi.attachInit();

        int no = cdbsess.getNumberInstances("/services/vlan");
        for(int i = 0; i < no; i++) {
            ConfBuf name =(ConfBuf)cdbsess.getElem("/services/vlan[%d]/name",
                                                     Integer.valueOf(i));
            String nameStr = name.toString();
            System.out.println("SERVICENAME = " + nameStr);

            ConfCdbUpgradePath oldPath =
                new ConfCdbUpgradePath("/ncs:services/vl:vlan%{s}",
                                      name.toString());
            ConfPath newPath = new ConfPath("/services/tunnel%{x}", name);
            maapi.create(th, newPath);

            ConfXMLParam[] oldparams = new ConfXMLParam[] {
                new ConfXMLParamLeaf("vl", "global-id"),
                new ConfXMLParamLeaf("vl", "iface"),
                new ConfXMLParamLeaf("vl", "unit"),
                new ConfXMLParamLeaf("vl", "vid"),
                new ConfXMLParamLeaf("vl", "description"),
            };
            ConfXMLParam[] data =
                cdbsess.getValues(oldparams, oldPath);

            ConfXMLParam[] newparams = new ConfXMLParam[] {
                new ConfXMLParamValue("tl", "gid", data[0].getValue()),
                new ConfXMLParamValue("tl", "interface", data[1].getValue()),
                new ConfXMLParamValue("tl", "assembly", data[2].getValue()),
                new ConfXMLParamValue("tl", "tunnel-id", data[3].getValue()),
                new ConfXMLParamValue("tl", "descr", data[4].getValue()),
            };
            maapi.setValues(th, newparams, newPath);

            maapi.ncsMovePrivateData(th, oldPath, newPath);
        }

        s1.close();
        s2.close();
    }
}

```

We will walk through this code also and point out the aspects that differ from the earlier more simple scenario. First we want to create the Cdb instance and get the CdbSession. However in this scenario the old namespace is removed and the Java API cannot retrieve it from NSO. To be able to use CDB to read and interpret the old YANG Model the old generated and removed Java namespace classes has to be temporarily reinstalled. This is solved by adding a jar (Java archive) containing these removed namespaces into the private-jar directory of the tunnel package. The removed namespace can then be instantiated and passed to Cdb via an overridden version of the Cdb.setUseForCdbUpgrade() method:

```
ArrayList<ConfNamespace> nsList = new ArrayList<ConfNamespace>();
```

```

nsList.add(new vlanService());
Socket s1 = new Socket("localhost", Conf.NCS_PORT);
Cdb cdb = new Cdb("cdb-upgrade-sock", s1);
cdb.setUseForCdbUpgrade(nsList);
CdbUpgradeSession cdbsess =
    cdb.startUpgradeSession(
        CdbDbType.CDB_RUNNING,
        EnumSet.of(CdbLockType.LOCK_SESSION,
        CdbLockType.LOCK_WAIT));

```

As an alternative to including the old namespace file in the package, a ConfNamespaceStub can be constructed for each old model that is to be accessed:

```

nslist.add(new ConfNamespaceStub(500805321,
    "http://example.com/vlan-service",
    "http://example.com/vlan-service",
    "vl"));

```

Since the old YANG model with the service point is removed the new service container with the new service has to be created before any config data can be written to this position:

```

ConfPath newPath = new ConfPath("/services/tunnel{?x}", name);
maapi.create(th, newPath);

```

The complete config for the old service is read via the CdbUpgradeSession. Note in particular that the path oldPath is constructed as a ConfCdbUpgradePath. These are paths that allow access to nodes that are not available in the current schema (i.e., nodes in deleted models).

```

ConfXMLParam[] oldparams = new ConfXMLParam[] {
    new ConfXMLParamLeaf("vl", "global-id"),
    new ConfXMLParamLeaf("vl", "iface"),
    new ConfXMLParamLeaf("vl", "unit"),
    new ConfXMLParamLeaf("vl", "vid"),
    new ConfXMLParamLeaf("vl", "description"),
};
ConfXMLParam[] data =
    cdbsess.getValues(oldparams, oldPath);

```

The new data structure with the service data is created and written to NSO via Maapi and the init transaction:

```

ConfXMLParam[] newparams = new ConfXMLParam[] {
    new ConfXMLParamValue("tl", "gid", data[0].getValue()),
    new ConfXMLParamValue("tl", "interface", data[1].getValue()),
    new ConfXMLParamValue("tl", "assembly", data[2].getValue()),
    new ConfXMLParamValue("tl", "tunnel-id", data[3].getValue()),
    new ConfXMLParamValue("tl", "descr", data[4].getValue()),
};
maapi.setValues(th, newparams, newPath);

```

Last the service private data is moved from the old position to the new position via the method Maapi.ncsMovePrivateData():

```

maapi.ncsMovePrivateData(th, oldPath, newPath);

```




CHAPTER 10

Java API Overview

- [Introduction, page 135](#)
- [MAAPI, page 138](#)
- [CDB API, page 140](#)
- [DP API, page 144](#)
- [NED API, page 156](#)
- [NAVU API, page 156](#)
- [ALARM API, page 164](#)
- [NOTIF API, page 166](#)
- [HA API, page 168](#)
- [Java API Conf Package, page 168](#)
- [Namespace classes and the loaded schema, page 171](#)

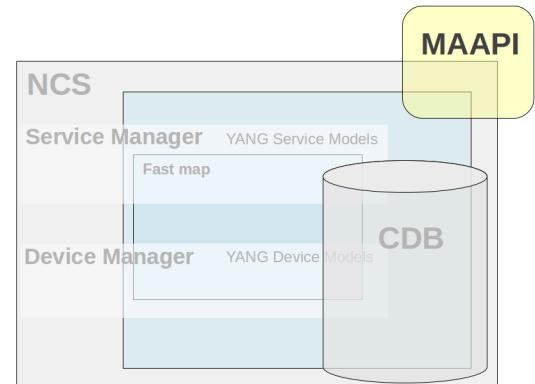
Introduction

The NSO Java library contains a variety of APIs for different purposes. In this chapter we introduce these and explain their usage. The Java library deliverables are found as two jar files (`ncs.jar` and `conf-api.jar`).

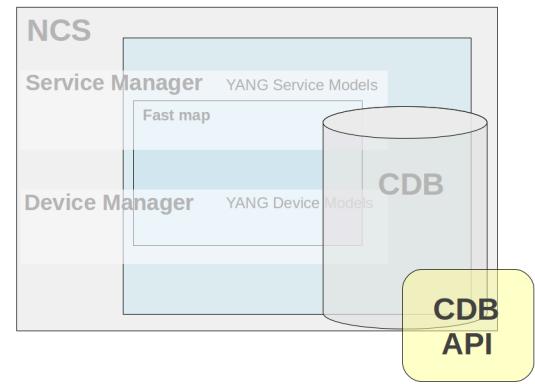
NSO Java library requires *Java SE 1.6* version or higher. NSO relies on log4j (<https://logging.apache.org>) for logging. No other dependencies exists for the NSO Java library. For convenience the Java build tool Apache ant (<https://ant.apache.org/>) is used to run all of the examples. However this tool is not a requirement for NSO.

General for all APIs are that they communicate with NSO using tcp sockets. This makes it possible to use all APIs from a remote location. The following APIs are included in the library:

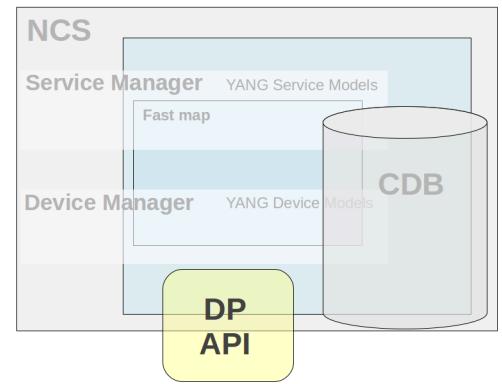
MAPI - (Management Agent API) Northbound interface that is transactional and user session based. Using this interface both configuration and operational data can be read. Configuration data can be written and committed as one transaction. The API is complete in the way that it is possible to write a new northbound agent using only this interface. It is also possible to attach to ongoing transactions in order to read uncommitted changes and/or modify data in these transactions.



CDB API - Southbound interface provides access to the CDB configuration database. Using this interface configuration data can be read. In addition, operational data that is stored in CDB can be read and written. This interface has a subscription mechanism to subscribe to changes. A subscription is specified on an path that points to an element in a YANG model or an instance in the instance tree. Any change under this point will trigger the subscription. CDB has also functions to iterate through the configuration changes when a subscription has triggered.

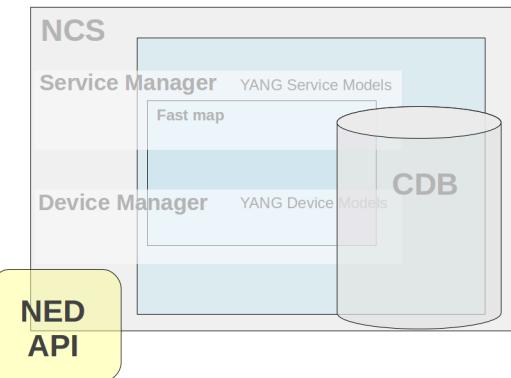


DP API - Southbound interface that enables callbacks, hooks and transforms. This API makes it possible to provide the service callbacks that handles service to device mapping logic. Other usual cases are external data providers for operational data or action callback implementations. There are also transaction and validation callbacks, etc. Hooks are callbacks that are fired when certain data is written and the hook is expected to do additional modifications of data. Transforms are callbacks that are used when complete mediation between two different models is necessary.

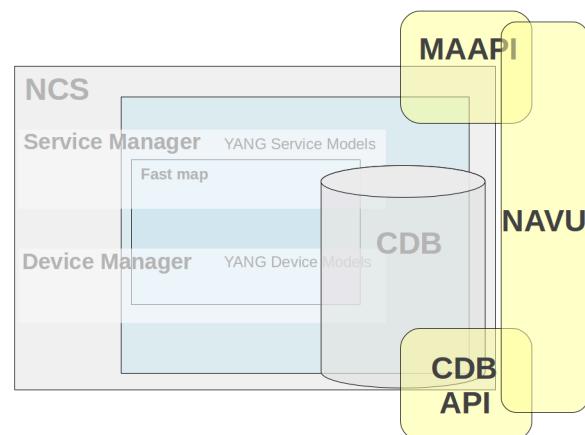


NED API - (Network Equipment Driver)

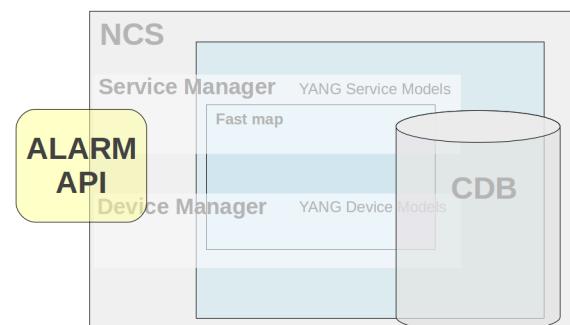
Southbound interface that mediate communication for devices that do not speak neither NETCONF nor SNMP. All prepackaged NEDs for different devices are written using this interface. It is possible to use the same interface to write your own NED. There are two types of NEDs, CLI NEDs and Generic NEDs. CLI NEDs can be used for devices that can be controlled by a Cisco style CLI syntax, in this case the NED is developed primarily by building a YANG model and a relatively small part in Java. In other cases the Generic NED can be used for any type of communication protocol.



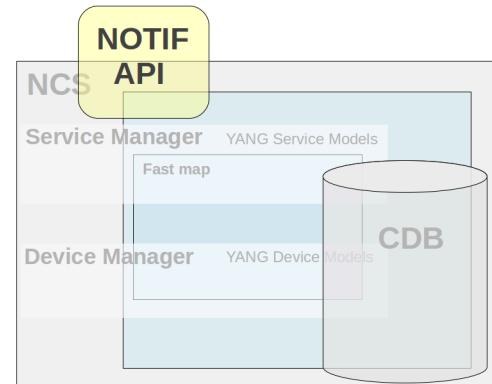
NAVU API - (Navigation Utilities) API that resides on top of the Maapi and Cdb APIs. It provides schema model navigation and instance data handling (read/write). Uses either a Maapi or Cdb context as data access and incorporates a subset of functionality from these (navigational and data read/write calls). Its major use is in service implementations which normally is about navigating device models and setting device data.



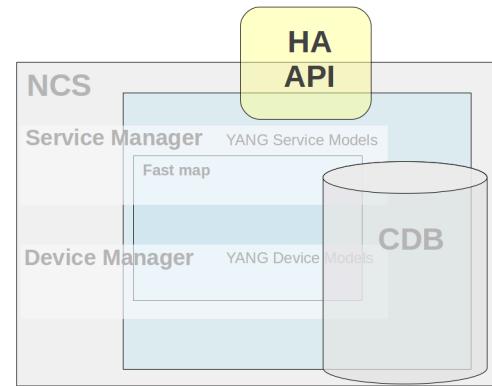
ALARM API - Eastbound api used both to consume and produce alarms in alignment with the NSO Alarm model. To consume alarms the AlarmSource interface is used. To produce a new alarm the AlarmSink interface is used. There is also a possibility to buffer produced alarms and make asynchronous writes to CDB to improve alarm performance.



NOTIF API - Northbound api used to subscribe on system events from NSO. These events are generated for audit log events, for different transaction states, for HA state changes, upgrade events, user sessions etc.



HA API - (High Availability) Northbound api used to manage a High Availability cluster of NSO instances. An NSO instance can be in one of three states NONE, MASTER or SLAVE. With the HA API the state can be queried and changed for NSO instances in the cluster.



In addition the *Conf API* framework contains utility classes for data types, keypaths, etc.

MAAPI

The Management Agent API (MAAPI) provides an interface to the Transaction engine in NSO. As such it is very versatile. Here are some examples on how the MAAPI interface can be used.

- Read and write configuration data stored by NSO or in an external database.
- Write our own north bound interface.
- We could access data inside a not yet committed transaction, e.g. as validation logic where our java code can attach itself to a running transaction and read through the not yet committed transaction and validate the proposed configuration change.
- During database upgrade we can access and write data to a special upgrade transaction.

The first step of a typical sequence of MAAPI API calls when writing a management application would be to create a user session. To create a user session is the equivalent of establishing an SSH connection from an NETCONF manager. It is up to the MAAPI application to authenticate users. The TCP connection

between MAAPI and NSO is neither encrypted, nor authenticated. The Maapi Java package does however include an `authenticate()` method that can be used by the application to hook into the AAA framework of NSO and let NSO authenticate the user.

Example 71. Establish a MAAPI connection

```
Socket socket = new Socket("localhost",Conf.NCS_PORT);
Maapi maapi = new Maapi(socket);
```

When a Maapi socket has been created the next step is to create a user session and supply the relevant information about the user for authentication.

Example 72. Starting a user session

```
maapi.startUserSession("admin",
                      InetAddress.getByName("localhost"),
                      "maapi",
                      new String[] {"admin"},
                      MaapiUserSessionFlag.PROTO_TCP);
```

When the user has been authenticated and a user session has been created the Maapi reference is now ready to establish a new transaction towards a data store. The following code snippet starts a read/write transaction towards running data store.

Example 73. Start a read/write transaction towards running

```
int th = maapi.startTrans(Conf.DB_RUNNING,
                         Conf.MODE_READ_WRITE);
```

The `startTrans(int db, int mode)` method of the Maapi class returns an integer which represents a transaction handle. This transaction handle is used when invoking the various Maapi methods.

An example of a typical transactional method is the `getElem()` method:

Example 74. Maapi.getElem()

```
public ConfValue getElem(int tid,
                        String fmt,
                        Object... arguments)
```

The `getElem(int th, String fmt, Object ... arguments)` first parameter is the transaction handle which is the integer that was returned by the `startTrans()` method. The `fmt` is a path that leads to a leaf in the data model. The path is expressed as a format string that contain fixed text with zero to many embedded format specifiers. For each specifier one argument in the variable argument list is expected.

The currently supported format specifiers in the Java API is:

- %d - requiring an integer parameter (type int) to be substituted.

- %s - requiring a `java.lang.String` parameter to be substituted.
- %x - requiring subclasses of type `com.tailf.conf.ConfValue` to be substituted.

```
ConfValue val = maapi.getElem(th,
    "/hosts/host{%-x}/interfaces{%-x}/ip",
    new ConfBuf("host1"),
    new ConfBuf("eth0"));
```

The return value `val` contains a reference to a `ConfValue` which is a superclass of all the `ConfValues` that maps to the specific yang data type. If the yang data type ip in the yang model is `ietf-inet-types:ipv4-address` we can narrow it to the subclass which is the corresponding `com.tailf.conf.ConfIPv4`

```
ConfIPv4 ipv4addr = (ConfIPv4)val;
```

The opposite operation of the `getElem()` is the `setElem()` method which set a leaf with a specific value.

```
maapi.setElem(th ,
    new ConfUInt16(1500),
    "/hosts/host{%-x}/interfaces{%-x}/ip/mtu",
    new ConfBuf("host1"),
    new ConfBuf("eth0"));
```

We have not yet committed the transaction so no modification is permanent. The data is only visible inside the current transaction. To commit the transaction we call:

```
maapi.applyTrans(th)
```

The method `applyTrans()` commits the current transaction to the running datastore.

Example 75. Commit a transaction

```
int th = maapi.startTrans(Conf.DB_RUNNING, Conf.MODE_READ_WRITE);
try {
    maapi.lock(Conf.DB_RUNNING);
    // make modifications to th
    maapi.setElem(th, ....);
    maapi.applyTrans(th);
    maapi.finishTrans(th);
} catch(Exception e) {
    maapi.finishTrans(th);
} finally {
    maapi.unLock(Conf.DB_RUNNING);
}
```

It is also possible to run the code above without `lock(Conf.DB_RUNNING)`.

The MAAPI is also intended to attach to already existing NSO transaction to inspect not yet committed data for example if we want to implement validation logic in Java. See [Example 82, “Attach Maapi to the current transaction”](#).

CDB API

This API provides an interface to the CDB Configuration database which stores all configuration data. With this API the user can:

- Start a CDB Session to read configuration data.

- Subscribe to changes in CDB - The subscription functionality makes it possible to receive events/ notifications when changes occur in CDB.

CDB can also be used to store operational data, i.e. data which is designated with a "config false" statement in the YANG data model. Operational data is read/write through the CDB API. NETCONF and the other northbound agents can only read operational data.

Java CDB API is intended to be fast and lightweight and the CDB read Sessions are expected to be short lived and fast. The NSO transaction manager is surpassed by CDB and therefore write operations on configurational data is prohibited. If operational data is stored in CDB both read and write operations on this data is allowed.

CDB is always locked for the duration of the session. It is therefore the responsibility of the programmer to make CDB interactions short in time and assure that all CDB sessions are closed when interaction has finished.

To initialize the CDB API a CDB socket has to be created and passed into the API base class `com.tailf.cdb.Cdb`:

Example 76. Establish a connection to CDB

```
Socket socket = new Socket("localhost", Conf.NCS_PORT);
Cdb cdb = new Cdb("MyCdbSock", socket);
```

After the cdb socket has been established a user could either start a CDB Session or start a subscription of changes in CDB:

Example 77. Establish a CDB Session

```
CdbSession session = cdb.startSession(CdbDbType.RUNNING);

/*
 * Retrieve the number of children in the list and
 * loop over these children
 */
for(int i = 0; i < session.numInstances("/servers/server"); i++) {
    ConfBuf name =
        (ConfBuf) session.getElem("/servers/server[%d]/hostname", i);
    ConfIPv4 ip =
        (ConfIPv4) session.getElem("/servers/server[%d]/ip", i);
}
```

We can refer to an element in a model with an expression like "/servers/server". This type of string reference to an element is called keypath or just path. To refer to element underneath a list, we need to identify which instance of the list elements that is of interest.

This can be performed either by pinpointing the sequence number in the ordered list, starting from 0. For instance the path: /servers/server[2]/port refers to the "port" leaf of the third server in the configuration. This numbering is only valid during the current CDB session. Note, the database is locked during this session.

We can also refer to list instances using the key values for the list. Remember that we specify in the data model which leaf or leafs in list that constitute the key. In our case a server has the "name" leaf as key. The syntax for keys is a space separated list of key values enclosed within curly brackets: { Key1 Key2 ...}. So /servers/server{www}/ip refers to the ip leaf of the server whose name is "www".

A YANG list may have more than one key for example the keypath: /dhcp/subNets/subNet{192.168.128.0, 255.255.255.0}/routers refers to the routers list of the subNet which has key "192.168.128.0", "255.255.255.0".

The keypath syntax allows for formatting characters and accompanying substitution arguments. For example `getElem("server[%d]/ifc{%s}/mtu", 2, "eth0")` is using a keypath with a mix of sequence number and keyvalues with formatting characters and argument. Expressed in text the path will reference the MTU of the third server instance's interface named "eth0".

The CdbSession java class have a number of methods to control current position in the model.

- CdbSession.getcwd() to get current position.
- CdbSession.cd() to change current position.
- CdbSession.pushd() to change and push a new position to a stack.
- CdbSession.popd() to change back to an stacked position.

Using relative paths and e.g. `CdbSession.pushd()` it is possible to write code that can be re-used for common sub-trees.

The current position also includes the namespace. If an element of another namespace should be read, then the prefix of that namespace should be set in the first tag of the keypath, like: `/smp:servers/server` where "smp" is the prefix of the namespace. It is also possible to set the default namespace for the CDB session with the method `CdbSession.setNamespace(ConfNamespace)`.

Example 78. Establish a CDB subscription

```
CdbSubscription sub = cdb.newSubscription();
int subid = sub.subscribe(1, new servers(), "/servers/server/");

// tell CDB we are ready for notifications
sub.subscribeDone();

// now do the blocking read
while (true) {
    int[] points = sub.read();
    // now do something here like diffIterate
    ....
}
```

The CDB subscription mechanism allows an external Java program to be notified when different parts of the configuration changes. For such a notification it is also possible to iterate through the change set in CDB for that notification.

Subscriptions are primarily to the running data store. Subscriptions towards the operational data store in CDB is possible, but the mechanism is slightly different see below.

The first thing to do is to register in CDB which paths should be subscribed to. This is accomplished with the `CdbSubscription.subscribe(...)` method. Each registered path returns a subscription point identifier. Each subscriber can have multiple subscription points, and there can be many different subscribers.

Every point is defined through a path - similar to the paths we use for read operations, with the difference that instead of fully instantiated paths to list instances we can choose to use tag paths i.e. leave out key value parts to be able to subscribe on all instances. We can subscribe either to specific leaves, or entire sub trees. Assume a YANG data model on the form of:

```

container servers {
    list server {
        key name;
        leaf name { type string; }
        leaf ip { type inet:ip-address; }
        leaf port type inet:port-number; }
    ....
}

```

Explaining this by example we get:

```
/servers/server/port
```

A subscription on a leaf. Only changes to this leaf will generate a notification.

```
/servers
```

Means that we subscribe to any changes in the sub tree rooted at /servers. This includes additions or removals of server instances, as well as changes to already existing server instances.

```
/servers/server{www}/ip
```

Means that we only want to be notified when the server "www" changes its ip address.

```
/servers/server/ip
```

Means we want to be notified when the leaf ip is changed in any server instance.

When adding a subscription point the client must also provide a priority, which is an integer. As CDB is changed, the change is part of a transaction. For example the transaction is initiated by a commit operation from the CLI or an edit-config operation in NETCONF resulting in the running database being modified. As the last part of the transaction CDB will generate notifications in lock-step priority order. First all subscribers at the lowest numbered priority are handled, once they all have replied and synchronized by calling `sync(CdbSubscriptionSyncType syncType)` the next set - at the next priority level - is handled by CDB. Not until all subscription points have been acknowledged is the transaction complete.

This implies that if the initiator of the transaction was for example a commit command in the CLI, the command will hang until notifications have been acknowledged.

Note that even though the notifications are delivered within the transaction it is not possible for a subscriber to reject the changes (since this would break the two-phase commit protocol used by the NSO back plane towards all data-providers).

When a client is done subscribing it needs to inform NSO it is ready to receive notifications. This is done by first calling `subscribeDone()`, after which the subscription socket is ready to be polled.

As a subscriber has read its subscription notifications using `read()` it can iterate through the changes that caused the particular subscription notification using the `diffIterate()` method.

It is also possible to start a new read-session to the CDB_PRE_COMMIT_RUNNING database to read the running database as it was before the pending transaction.

Subscriptions towards the operational data in CDB are similar to the above, but due to the fact that the operational data store is designed for light-weight access, and thus does not have transactions and normally avoids the use of any locks, there are several differences - in particular:

- Subscription notifications are only generated if the writer obtains the "subscription lock", by using the `startSession()` with the `CdbLockType.LOCKREQUEST`. In addition when starting a session

towards the operation data we need to pass the `CdbDBType.CDB_OPERATIONAL` when starting a CDB session:

```
CdbSession sess =
    cdb.startSession(CdbDBType.CDB_OPERATIONAL,
                     EnumSet.of(CdbLockType.LOCK_REQUEST));
```

- No priorities are used.
- Neither the writer that generated the subscription notifications nor other writers to the same data are blocked while notifications are being delivered. However the subscription lock remains in effect until notification delivery is complete.
- The previous value for modified leaf is not available when using the `diffIterate()` method.

Essentially a write operation towards the operational data store, combined with the subscription lock, takes on the role of a transaction for configuration data as far as subscription notifications are concerned. This means that if operational data updates are done with many single-element write operations, this can potentially result in a lot of subscription notifications. Thus it is a good idea to use the multi-element `setObject()` taking an array of `ConfValues` which sets a complete container or `setValues()` taking an array of `ConfXMLParam` and potent of setting an arbitrary part of the model. This to keep down notifications to subscribers when updating operational data.

For write operations that do not attempt to obtain the subscription lock are allowed to proceed even during notification delivery. Therefore it is the responsibility of the programmer to obtain the lock as needed when writing to the operational data store. E.g. if subscribers should be able to reliably read the exact data that resulted from the write that triggered their subscription, the subscription lock must always be obtained when writing that particular set of data elements. One possibility is of course to obtain the lock for all writes to operational data, but this may have an unacceptable performance impact.

To view registered subscribers use the `ncs --status` command. For details on how to use the different subscription functions see the javadoc for NSO Java API.

The code in the example `${NCS_DIR}/examples.ncs/getting-started/developing-with-ncs/1-cdb`. illustrates three different type of CDB subscribers.

- A simple Cdb config subscriber that utilizes the low level Cdb API directly to subscribe to changes in subtree of the configuration.
- Two Navu Cdb subscribers, one subscribing to configuration changes, and one subscribing to changes in operational data.

DP API

The DP API makes it possible to create callbacks which are called when certain events occur in NSO. As the name of the API indicates it is possible to write data provider callbacks that provide data to NSO that is stored externally. However this is only one of several callback types provided by this API. There exist callback interfaces for the following types:

- Service Callbacks - invoked for a service callpoints in the the YANG model. Implements service to device information mappings. See for example `${NCS_DIR}/examples.ncs/getting-started/developing-with-ncs/4-rfs-service`
- Action Callbacks - invoked for a certain action in the YANG model which is defined with a callpoint directive.
- Authentication Callbacks - invoked for external authentication functions.
- Authorization Callbacks - invoked for external authorization of operations and data. Note, avoid this callback if possible since performance will otherwise be affected.

- Data Callbacks - invoked for data provision and manipulation for certain data elements in the YANG model which is defined with a callpoint directive.
- DB Callbacks - invoked for external database stores.
- Range Action Callbacks - A variant of action callback where ranges are defined for the key values.
- Range Data Callbacks - A variant of data callback where ranges are defined for the data values.
- Snmp Inform Response Callbacks - invoked for response on Snmp inform requests on a certain element in the Yang model which is defined by a callpoint directive.
- Transaction Callbacks - invoked for external participants in the two-phase commit protocol.
- Transaction Validation Callbacks - invoked for external transaction validation in the validation phase of a two phase commit.
- Validation Callbacks - invoked for validation of certain elements in the YANG Model which is designed with a callpoint directive.

The callbacks are methods in ordinary java POJOs. These methods are adorned with a specific *Java Annotations* syntax for that callback type. The annotation makes it possible to add meta data information to NSO about the supplied method. The annotation includes information of which `callType` and, when necessary, which `callpoint` the method should be invoked for.



Note

Only one Java object can be registered on one and the same `callpoint`. Therefore, when a new Java object register on a `callpoint` which already has been registered, the earlier registration (and Java object) will be silently removed.

Transaction and Data Callbacks

By default NSO stores all configuration data in its CDB data store. We may wish to store and configure other data in NSO than what is defined by the NSO built-in YANG models, alternatively we may wish to store parts of the NSO tree outside NSO (CDB) i.e. in an external database. Say for example that we have our customer database stored in a relational database disjunct from NSO. To implement this, we must do a number of things: We must define a `callpoint` somewhere in the configuration tree, and we must implement what is referred to as a data provider. Also NSO executes all configuration changes inside transactions and if we want NSO (CDB) and our external database to participate in the same two-phase commit transactions we must also implement a transaction callback. All together it will appear as if the external data is part of the overall NSO configuration, thus the service model data can refer directly into this external data - typically in order to validate service instances.

The basic idea for a data provider, is that it participates entirely in each NSO transaction, and it is also responsible for reading and writing all data in the configuration tree below the callpoint. Before explaining how to write a data provider and what the responsibilities of a data provider are, we must explain how the NSO transaction manager drives all participants in a lock step manner through the phases of a transaction.

A transaction has a number of phases, the external data provider gets called in all the different phases. This is done by implementing a *Transaction callback* class and then registering that class. We have the following distinct phases of a NSO transaction:

- `init()` In this phase the Transaction callback class `init()` methods gets invoked. We use annotation on the method to indicate that it's the `init()` method as in:

```
public class MyTransCb {
    @TransCallback(callType=TransCBType.INIT)
    public void init(DpTrans trans) throws DpCallbackException {
```

```
        return;
    }
```

Each different callback method we wish to register, must be annotated with an annotation from `TransCBType`

The callback is invoked when a transaction starts, but NSO delays the actual invocation as an optimization. For a data provider providing configuration data, `init()` is invoked just before the first data-reading callback, or just before the `transLock()` callback (see below), whichever comes first. When a transaction has started, it is in a state we refer to as READ. NSO will, while the transaction is in the READ state, execute a series of read operations towards (possibly) different callpoints in the data provider.

Any write operations performed by the management station are accumulated by NSO and the data provider doesn't see them while in the READ state.

- `transLock()` - This callback gets invoked by NSO at the end of the transaction. NSO has accumulated a number of write operations and will now initiate the final write phases. Once the `transLock()` callback has returned, the transaction is in the VALIDATEstate. In the VALIDATE state, NSO will (possibly) execute a number of read operations in order to validate the new configuration. Following the read operations for validations comes the invocation of one of the `writeStart()` or `transUnlock()` callbacks.
- `transUnlock()` - This callback gets invoked by NSO if the validation failed or if the validation was done separate from the commit (e.g. by giving a `validate` command in the CLI). Depending on where the transaction originated, the behavior after a call to `transUnlock()` differs. If the transaction originated from the CLI, the CLI reports to the user that the configuration is invalid and the transaction remains in the READ state whereas if the transaction originated from a NETCONF client, the NETCONF operation fails and a NETCONF `rpc` error is reported to the NETCONF client/manager.
- `writeStart()` - If the validation succeeded, the `writeStart()` callback will be called and the transaction enters the WRITE state. While in WRITE state, a number of calls to the write data callbacks `setElem()`, `create()` and `remove()` will be performed.

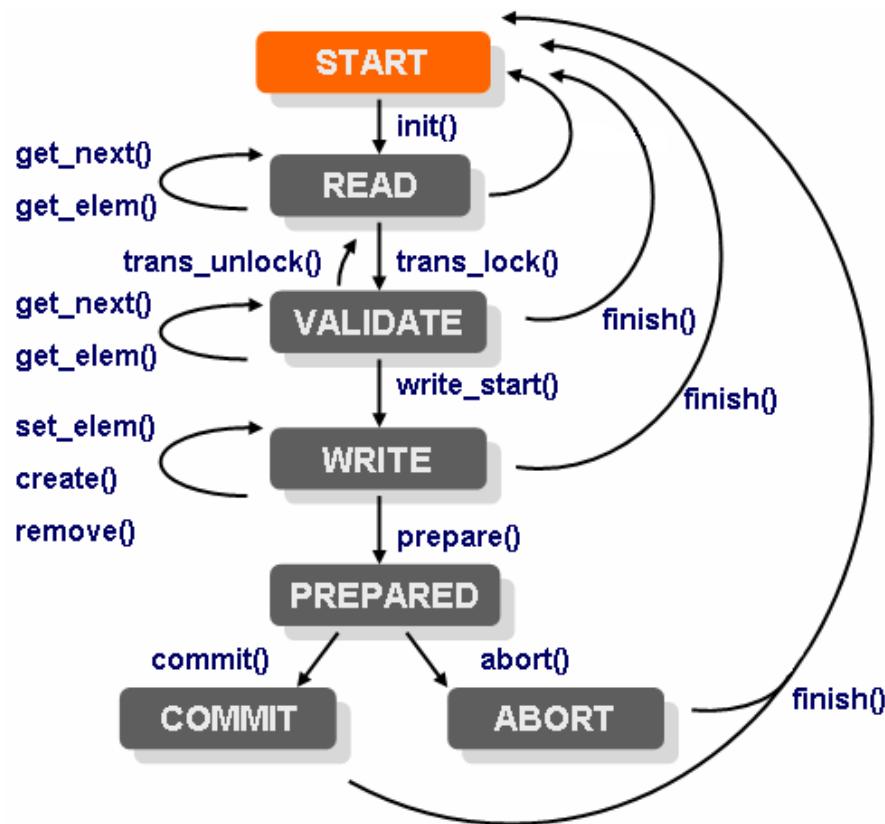
If the underlying database supports real atomic transactions, this is a good place to start such a transaction.

The application should not modify the real running data here. If, later, the `abort()` callback is called, all write operations performed in this state must be undone.

- `prepare()` - Once all write operations are executed, the `prepare()` callback is executed. This callback ensures that all participants have succeeded in writing all elements. The purpose of the callback is merely to indicate to NSO that the data provider is ok, and has not yet encountered any errors.
- `abort()` - If any of the participants die or fail to reply in the `prepare()` callback, the remaining participants all get invoked in the `abort()` callback. All data written so far in this transaction should be disposed of.
- `commit()` - If all participants successfully replied in their respective `prepare()` callbacks, all participants get invoked in their respective `commit()` callbacks. This is the place to make all data written by the write callbacks in WRITE state permanent.
- `finish()` - And finally, the `finish()` callback gets invoked at the end. This is a good place to deallocate any local resources for the transaction.

The `finish()` callback can be called from several different states.

The following picture illustrates the conceptual state machine a NSO transaction goes through.



NSO transaction state machine

All callbacks methods are optional. If a callback method is not implemented, it is the same as having an empty callback which simply returns.

Similar to how we have to register Transaction callbacks, we must also register data callbacks. The transaction callbacks cover the life span of the transaction, and the data callbacks are used to read and write data inside a transaction. The data callbacks have access to what is referred to as the transaction context in the form of a `DpTrans` object.

We have the following data callbacks:

- `getElem()` This callback is invoked by NSO when NSO needs to read the actual value of a leaf element. We must also implement the `getElem()` callback for the keys. NSO invokes `getElem()` on a key as an existence test.

We define the `getElem` callback inside a class as:

```

public static class DataCb {
    @DataCallback(callPoint="foo", callType=DataCBType.GET_ELEM)
    public ConfValue getElem(DpTrans trans, ConfObject[] kp)
        throws DpCallbackException {
        ....
    }
}

```

- `existsOptional()` This callback is called for all type less and optional elements, i.e. presence containers and leafs of type empty. If we have presence containers or leafs of type empty we cannot use the `getElem()` callback to read the value of such a node, since it does not have a type. An example of a data model could be:

```

container bs {
    presence "";
    tailf:callpoint bcp;
    list b {
        key name;
        max-elements 64;
        leaf name {
            type string;
        }
        container opt {
            presence "";
            leaf ii {
                type int32;
            }
        }
        leaf foo {
            type empty;
        }
    }
}

```

The above YANG fragment has 3 nodes that may or may not exist and that do not have a type. If we do not have any such elements, nor any operational data lists without keys (see below), we do not need to implement the existsOptional() callback.

If we have the above data model, we must implement the existsOptional(), and our implementation must be prepared to reply on calls of the function for the paths /bs, /bs/b/opt, and /bs/b/foo. The leaf /bs/b/opt/ii is not mandatory, but it does have a type namely int32, and thus the existence of that leaf will be determined through a call to the getElem() callback.

The existsOptional() callback may also be invoked by NSO as "existence test" for an entry in an operational data list without keys. Normally this existence test is done with a getElem() request for the first key, but since there are no keys, this callback is used instead. Thus if we have such lists, we must also implement this callback, and handle a request where the keypath identifies a list entry.

- `iterator()` and `getKey()` This pair of callback is used when NSO wants to traverse a YANG list. The job of the `iterator()` callback is to return a `Iterator` object that is invoked by the library. For each `Object` returned by the `iterator`, the NSO library will invoke the `getKey()` callback on the returned object. The `getKey` callback shall return a `ConfKey` value.

An alternative to the `getKey()` callback is to register the optional `getObject()` callback whose job it is to return not just the key, but the entire YANG list entry. It is possible to register both `getKey()` and `getObject()` or either. If the `getObject()` is registered, NSO will attempt to use it only when bulk retrieval is executed.

We also have two additional optional callbacks that may be implemented for efficiency reasons.

- `getObject()` If this optional callback is implemented, the work of the callback is to return an entire object, i.e. a list instance. This is not the same `getObject()` as the one that is used in combination with the `iterator()`
- `numInstances()` When NSO needs to figure out how many instances we have of a certain element, by default NSO will repeatedly invoke the `iterator()` callback. If this callback is installed, it will be called instead.

The following example illustrates an external data provider. The example is possible to run from the examples collection. It resides under `${NCS_DIR}/examples.ncs/getting-started/developing-with-ncs/6-extern-db`.

The example comes with a tailor made database - MyDb. That source code is provided with the example but not shown here. However the functionality will be obvious from the method names like newItem(), lock(), save() etc.

Two classes are implemented, one for the Transaction callbacks and another for the Data callbacks.

The data model we wish to incorporate into NSO is a trivial list of work items. It looks like:

Example 79. work.yang

```
module work {
    namespace "http://example.com/work";
    prefix w;
    import ietf-yang-types {
        prefix yang;
    }
    import tailf-common {
        prefix tailf;
    }
    description "This model is used as a simple example model
                  illustrating how to have NCS configuration data
                  that is stored outside of NCS - i.e not in CDB";
    revision 2010-04-26 {
        description "Initial revision.";
    }

    container work {
        tailf:callpoint workPoint;
        list item {
            key key;
            leaf key {
                type int32;
            }
            leaf title {
                type string;
            }
            leaf responsible {
                type string;
            }
            leaf comment {
                type string;
            }
        }
    }
}
```

Note the callpoint directive in the model, it indicates that an external Java callback must register itself using that name. That callback will be responsible for all data below the callpoint.

To compile the `work.yang` data model and then also to generate Java code for the data model we invoke `make all` in the example package `src` directory. The Makefile will compile the `yang` files in the package, generate Java code for those data models and then also invoke `ant` in the Java `src` directory.

The Data callback class looks as follows:

Example 80. DataCb class

```
@DataCallback(callPoint=work.callpoint_workPoint,
              callType=DataCBType.ITERATOR)
public Iterator<Object> iterator(DpTrans trans,
                                    ConfObject[] keyPath)
throws DpCallbackException {
```

```

        return MyDb.iterator();
    }

    @DataCallback(callPoint=work.callpoint_workPoint,
                 callType=DataCBType.GET_NEXT)
    public ConfKey getKey(DpTrans trans, ConfObject[] keyPath,
                          Object obj)
        throws DpCallbackException {
        Item i = (Item) obj;
        return new ConfKey( new ConfObject[] { new ConfInt32(i.key) } );
    }

    @DataCallback(callPoint=work.callpoint_workPoint,
                 callType=DataCBType.GET_ELEM)
    public ConfValue getElem(DpTrans trans, ConfObject[] keyPath)
        throws DpCallbackException {

        ConfInt32 kv = (ConfInt32) ((ConfKey) keyPath[1]).elementAt(0);
        Item i = MyDb.findItem( kv.intValue() );
        if (i == null) return null; // not found

        // switch on xml elem tag
        ConfTag leaf = (ConfTag) keyPath[0];
        switch (leaf.getTagHash()) {
        case work._key:
            return new ConfInt32(i.key);
        case work._title:
            return new ConfBuf(i.title);
        case work._responsible:
            return new ConfBuf(i.responsible);
        case work._comment:
            return new ConfBuf(i.comment);
        default:
            throw new DpCallbackException("xml tag not handled");
        }
    }

    @DataCallback(callPoint=work.callpoint_workPoint,
                 callType=DataCBType.SET_ELEM)
    public int setElem(DpTrans trans, ConfObject[] keyPath,
                       ConfValue newval)
        throws DpCallbackException {
        return Conf.REPLY_ACCUMULATE;
    }

    @DataCallback(callPoint=work.callpoint_workPoint,
                 callType=DataCBType.CREATE)
    public int create(DpTrans trans, ConfObject[] keyPath)
        throws DpCallbackException {
        return Conf.REPLY_ACCUMULATE;
    }

    @DataCallback(callPoint=work.callpoint_workPoint,
                 callType=DataCBType.REMOVE)
    public int remove(DpTrans trans, ConfObject[] keyPath)
        throws DpCallbackException {
        return Conf.REPLY_ACCUMULATE;
    }

    @DataCallback(callPoint=work.callpoint_workPoint,
                 callType=DataCBType.NUM_INSTANCES)

```

```

public int numInstances(DpTrans trans, ConfObject[] keyPath)
    throws DpCallbackException {
    return MyDb.numItems();
}

@DataCallback(callPoint=work.callpoint_workPoint,
              callType=DataCBType.GET_OBJECT)
public ConfValue[] getObject(DpTrans trans, ConfObject[] keyPath)
    throws DpCallbackException {
    ConfInt32 kv = (ConfInt32) ((ConfKey) keyPath[0]).elementAt(0);
    Item i = MyDb.findItem( kv.intValue() );
    if (i == null) return null; // not found
    return getObject(trans, keyPath, i);
}

@DataCallback(callPoint=work.callpoint_workPoint,
              callType=DataCBType.GET_NEXT_OBJECT)
public ConfValue[] getObject(DpTrans trans, ConfObject[] keyPath,
                             Object obj)
    throws DpCallbackException {
    Item i = (Item) obj;
    return new ConfValue[] {
        new ConfInt32(i.key),
        new ConfBuf(i.title),
        new ConfBuf(i.responsible),
        new ConfBuf(i.comment)
    };
}

```

First we see how the Java annotations are used to declare the type of callback for each method. Secondly, we see how the `getElem()` callback inspects the `keyPath` parameter passed to it to figure out exactly which element NSO wants to read. The `keyPath` is an array of `ConfObject` values. Keypaths are central to the understanding of the NSO Java library since they are used to denote objects in the configuration. A keypath uniquely identifies an element in the instantiated configuration tree.

Furthermore, the `getElem()` switches on the tag `keyPath[0]` which is a `ConfTag` using symbolic constants from the class "work". The "work" class was generated through the call to `ncsc --emit-java`

The three write callbacks, `setElem()`, `create()` and `remove()` all return the value `Conf.REPLY_ACCUMULATE`. If our backend database has real support to abort transactions, it is a good idea to initiate a new backend database transaction in the Transaction callback `init()` (more on that later), whereas if our backend database doesn't support proper transactions, we can fake real transactions by returning `Conf.REPLY_ACCUMULATE` instead of actually writing the data. Since the final verdict of the NSO transaction as a whole may very well be to abort the transaction, we must be prepared to undo all write operations. The `Conf.REPLY_ACCUMULATE` return value means that we ask the library to cache the write for us.

The Transaction callback class, looks like:

Example 81. TransCb class

```

@TransCallback(callType=TransCBType.INIT)
public void init(DpTrans trans) throws DpCallbackException {
    return;
}

@TransCallback(callType=TransCBType.TRANS_LOCK)
public void transLock(DpTrans trans) throws DpCallbackException {
    MyDb.lock();
}

```

```

    }

@TransCallback(callType=TransCBType.TRANS_UNLOCK)
public void transUnlock(DpTrans trans) throws DpCallbackException {
    MyDb.unlock();
}

@TransCallback(callType=TransCBType.PREPARE)
public void prepare(DpTrans trans) throws DpCallbackException {
    Item i;
    ConfInt32 kv;
    for (Iterator<DpAccumulate> it = trans.accumulated();
         it.hasNext(); ) {
        DpAccumulate ack= it.next();
        // check op
        switch (ack.getOperation()) {
        case DpAccumulate.SET_ELEM:
            kv = (ConfInt32) ((ConfKey) ack.getKP()[1]).elementAt(0);
            if ((i = MyDb.findItem( kv.intValue())) == null)
                break;
            // check leaf tag
            ConfTag leaf = (ConfTag) ack.getKP()[0];
            switch (leaf.getTagHash()) {
            case work._title:
                i.title = ack.getValue().toString();
                break;
            case work._responsible:
                i.responsible = ack.getValue().toString();
                break;
            case work._comment:
                i.comment = ack.getValue().toString();
                break;
            }
            break;
        case DpAccumulate.CREATE:
            kv = (ConfInt32) ((ConfKey) ack.getKP()[0]).elementAt(0);
            MyDb newItem(new Item(kv.intValue()));
            break;
        case DpAccumulate.REMOVE:
            kv = (ConfInt32) ((ConfKey) ack.getKP()[0]).elementAt(0);
            MyDb.removeItem(kv.intValue());
            break;
        }
    }
    try {
        MyDb.save("running.prep");
    } catch (Exception e) {
        throw
            new DpCallbackException("failed to save file: running.prep",
                                   e);
    }
}

@TransCallback(callType=TransCBType.ABORT)
public void abort(DpTrans trans) throws DpCallbackException {
    MyDb.restore("running.DB");
    MyDb.unlink("running.prep");
}

@TransCallback(callType=TransCBType.COMMIT)
public void commit(DpTrans trans) throws DpCallbackException {
    try {

```

```

        MyDb.rename( "running.prep", "running.DB" );
    } catch (DpCallbackException e) {
        throw new DpCallbackException("commit failed");
    }
}

@TransCallback(callType=TransCBType.FINISH)
public void finish(DpTrans trans) throws DpCallbackException {
    ;
}
}

```

We can see how the `prepare()` callback goes through all write operations and actually execute them towards our database `MyDb`.

Service and Action Callbacks

Both service and action callbacks are fundamental in NSO.

Implementing a service callback is one way of creating a service type. This and other ways of creating service types are in depth described in the [Chapter 13, Package Development](#) chapter.

Action callbacks are used to implement arbitrary operations in java. These operations can be basically anything, e.g. downloading a file, performing some test, resetting alarms, etc, but they should not modify the modeled configuration.

The actions are defined in the YANG model by means of `rpc` or `tailf:action` statements. Input and output parameters can optionally be defined via `input` and `output` statements in the YANG model. To specify that the `rpc` or `action` is implemented by a callback, the model uses a `tailf:actionpoint` statement.

The action callbacks are:

- `init()` Similar to the transaction `init()` callback. However note that unlike the case with transaction and data callbacks, both `init()` and `action()` are registered for each `actionpoint` (i.e. different action points can have different `init()` callbacks), and there is no `finish()` callback - the action is completed when the `action()` callback returns.
- `action()` This callback is invoked to actually execute the `rpc` or `action`. It receives the input parameters (if any) and returns the output parameters (if any).

In the `examples.ncs/service-provider/mpls-vpn` example we can define a *self-test* action. In the `packages/l3vpn/src/yang/l3vpn.yang` we locate the service callback definition:

```
uses ncs:service-data;
ncs:servicepoint vlanspnt;
```

Beneath the service callback definition we add a action callback definition so the resulting YANG looks like the following:

```
uses ncs:service-data;
ncs:servicepoint vlanspnt;

tailf:action self-test {
    tailf:info "Perform self-test of the service";
    tailf:actionpoint vlanselftest;
    output {
        leaf success {
            type boolean;
        }
    }
}
```

```

        leaf message {
            type string;
            description
                "Free format message.";
        }
    }
}

```

The packages/l3vpn/src/java/src/com/example/l3vpnRFS.java already contains an action implementation but it has been suppressed since no *actionpoint* with the corresponding name has been defined in the YANG model, before now.

```

/**
 * Init method for selftest action
 */
@ActionCallback(callPoint="l3vpn-self-test",
callType=ActionCBType.INIT)
public void init(DpActionTrans trans) throws DpCallbackException {

}

/**
 * Selftest action implementation for service
 */
@ActionCallback(callPoint="l3vpn-self-test", callType=ActionCBType.ACTION)
public ConfXMLParam[] selftest(DpActionTrans trans, ConTag name,
                                ConfObject[] kp, ConfXMLParam[] params)
throws DpCallbackException {
    try {
        // Refer to the service yang model prefix
        String nsPrefix = "l3vpn";
        // Get the service instance key
        String str = ((ConfKey)kp[0]).toString();

        return new ConfXMLParam[] {
            new ConfXMLParamValue(nsPrefix, "success", new ConfBool(true)),
            new ConfXMLParamValue(nsPrefix, "message", new ConfBuf(str))};
    } catch (Exception e) {
        throw new DpCallbackException("self-test failed", e);
    }
}
}

```

Validation Callbacks

In the VALIDATE state of a transaction, NSO will validate the new configuration. This consists of verification that specific YANG constraints such as `min-elements`, `unique`, etc, as well as arbitrary constraints specified by `must` expressions, are satisfied. The use of `must` expressions is the recommended way to specify constraints on relations between different parts of the configuration, both due to its declarative and concise form and due to performance considerations, since the expressions are evaluated internally by the NSO transaction engine.

In some cases it may still be motivated to implement validation logic via callbacks in code. The YANG model will then specify a *validation point* by means of a `tailf:validate` statement. By default the callback registered for a validation point will be invoked whenever a configuration is validated, since the callback logic will typically be dependent on data in other parts of the configuration, and these dependencies are not known by NSO. Thus it is important from a performance point of view to specify the actual dependencies by means of `tailf:dependency` substatements to the `validate` statement.

Validation callbacks use the MAAPI API to attach to the current transaction. This makes it possible to read the configuration data that is to be validated, even though the transaction is not committed yet. The view of

the data is effectively the pre-existing configuration "shadowed" by the changes in the transaction, and thus exactly what the new configuration will look like if it is committed.

Similar to the case of transaction and data callbacks, there are transaction validation callbacks that are invoked when the validation phase starts and stops, and validation callbacks that are invoked for the specific validation points in the YANG model.

The transaction validation callbacks are:

- `init()` This callback is invoked when the validation phase starts. It will typically attach to the current transaction:

Example 82. Attach Maapi to the current transaction

```
public class SimpleValidator implements DpTransValidateCallback{
    ...
    @TransValidateCallback(callType=TransValidateCBType.INIT)
    public void init(DpTrans trans) throws DpCallbackException{
        try {
            th = trans.thandle;
            maapi.attach(th, new MyNamesapce().hash(), trans.uinfo.usid);
            ...
        } catch(Exception e) {
            throw new DpCallbackException("failed to attach via maapi: "+
                e.getMessage());
        }
    }
}
```

- `stop()` This callback is invoked when the validation phase ends. If `init()` attached to the transaction, `stop()` should detach from it.

The actual validation logic is implemented in a validation callback:

- `validate()` This callback is invoked for a specific validation point.

Transforms

Transforms implement a mapping between one part of the data model - the front-end of the transform - and another part - the back-end of the transform. Typically the front-end is visible to northbound interfaces, while the back-end is not, but for operational data (`config false` in the data model), a transform may implement a different view (e.g. aggregation) of data that is also visible without going through the transform.

The implementation of a transform uses techniques already described in this section: Transaction and data callbacks are registered and invoked when the front-end data is accessed, and the transform uses the MAAPi API to attach to the current transaction, and accesses the back-end data within the transaction.

To specify that the front-end data is provided by a transform, the data model uses the `tailf:callpoint` statement with a `tailf:transform true` substatement. Since transforms do not participate in the two-phase commit protocol, they only need to register the `init()` and `finish()` transaction callbacks. The `init()` callback attaches to the transaction, and `finish()` detaches from it. Also, a transform for operational data only needs to register the data callbacks that read data, i.e. `getElem()`, `existsOptional()`, etc.

Hooks

Hooks make it possible have changes to the configuration trigger additional changes. In general this should only be done when the data that is written by the hook is not visible to northbound interfaces,

since otherwise the additional changes will make it difficult for e.g. EMS or NMS systems to manage the configuration - the complete configuration resulting from a given change can not be predicted. However one use case in NSO for hooks that trigger visible changes is precisely to model managed devices that have this behavior: hooks in the device model can emulate what the device does on certain configuration changes, and thus the device configuration in NSO remains in sync with the actual device configuration.

The implementation technique for a hook is very similar to that for a transform. Transaction and data callbacks are registered, and the MAAPI API is used to attach to the current transaction and write the additional changes into the transaction. As for transforms, only the `init()` and `finish()` transaction callbacks need to be registered, to do the MAAPI attach and detach. However only data callbacks that write data, i.e. `setElem()`, `create()`, etc need to be registered, and depending on which changes should trigger the hook invocation, it is possible to register only a subset of those. For example, if the hook is registered for a leaf in the data model, and only changes to the value of that leaf should trigger invocation of the hook, it is sufficient to register `setElem()`.

To specify that changes to some part of the configuration should trigger a hook invocation, the data model uses the `tailf:callpoint` statement with a `tailf:set-hook` or `tailf:transaction-hook` substatement. A set-hook is invoked immediately when a north bound agent requests a write operation on the data, while a transaction-hook is invoked when the transaction is committed. For the NSO-specific use case mentioned above, a set-hook should be used. The `tailf:set-hook` and `tailf:transaction-hook` statements take an argument specifying the extent of the data model the hook applies to.

NED API

NSO can speak southbound to an arbitrary management interface. This is of course not entirely automatic like with NETCONF or SNMP, and depending on the type of interface the device has for configuration, this may involve some programming. Devices with a Cisco style CLI can however be managed by writing YANG models describing the data in the CLI, and a relatively thin layer of Java code to handle the communication to the devices. Refer to (Chapter 2, *Network Element Drivers (NEDs)* in *NSO 5.7 NED Development*) for more information.

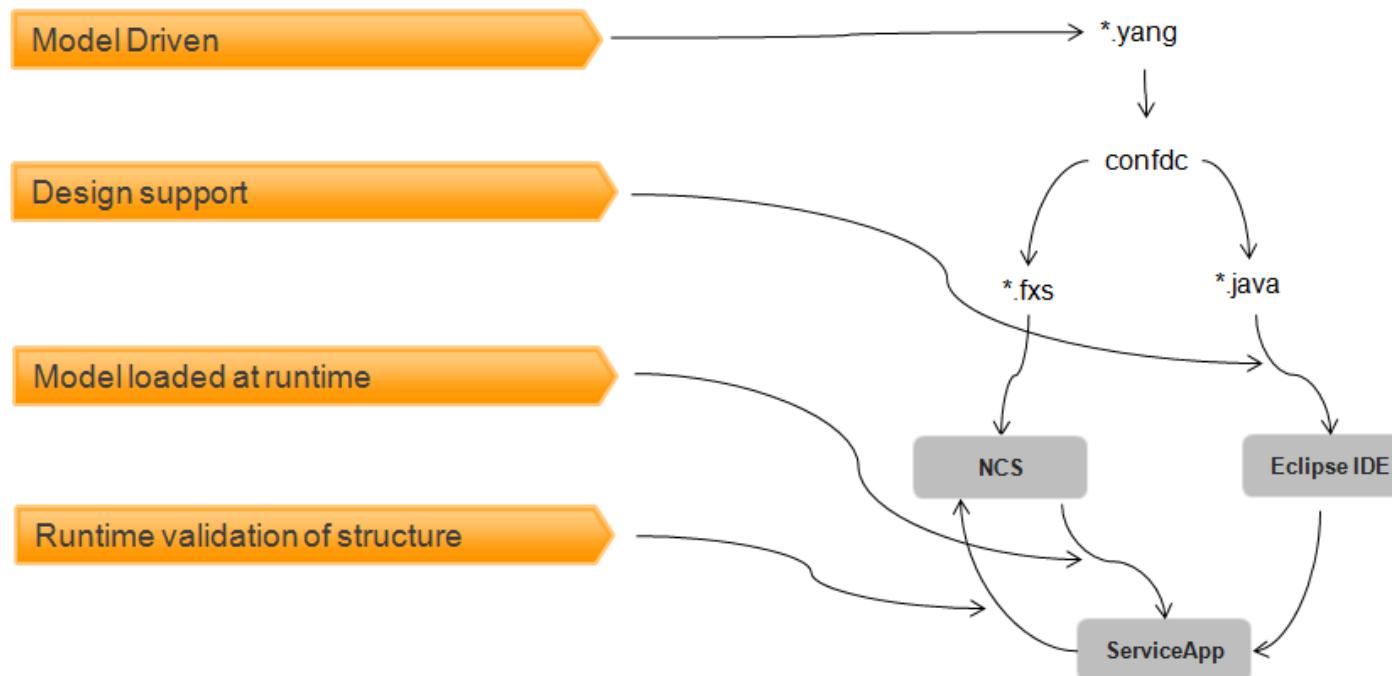
NAVU API

The NAVU API provides a DOM driven approach to navigate the NSO service and device models. The main features of the NAVU API is dynamic schema loading at start up and lazy loading of instance data. The navigation model is based on the YANG language structure. In addition to navigation and reading of values NAVU also provides methods to modify the data model. Furthermore, it supports execution of actions modelled in the service model.

By using NAVU it is easy to drill down through tree structures with minimal effort using the node by node navigation primitives. Alternatively, we can use the NAVU search feature. This feature is especially useful when we need find information deep down in the model structures.

NAVU requires all models i.e. the complete NSO service model with all its augmented sub models. This is loaded at runtime from NSO. NSO has in turn acquired these from loaded .fxs files. The .fxs files are a product from the ncsc tool with compiles these from the .yang files.

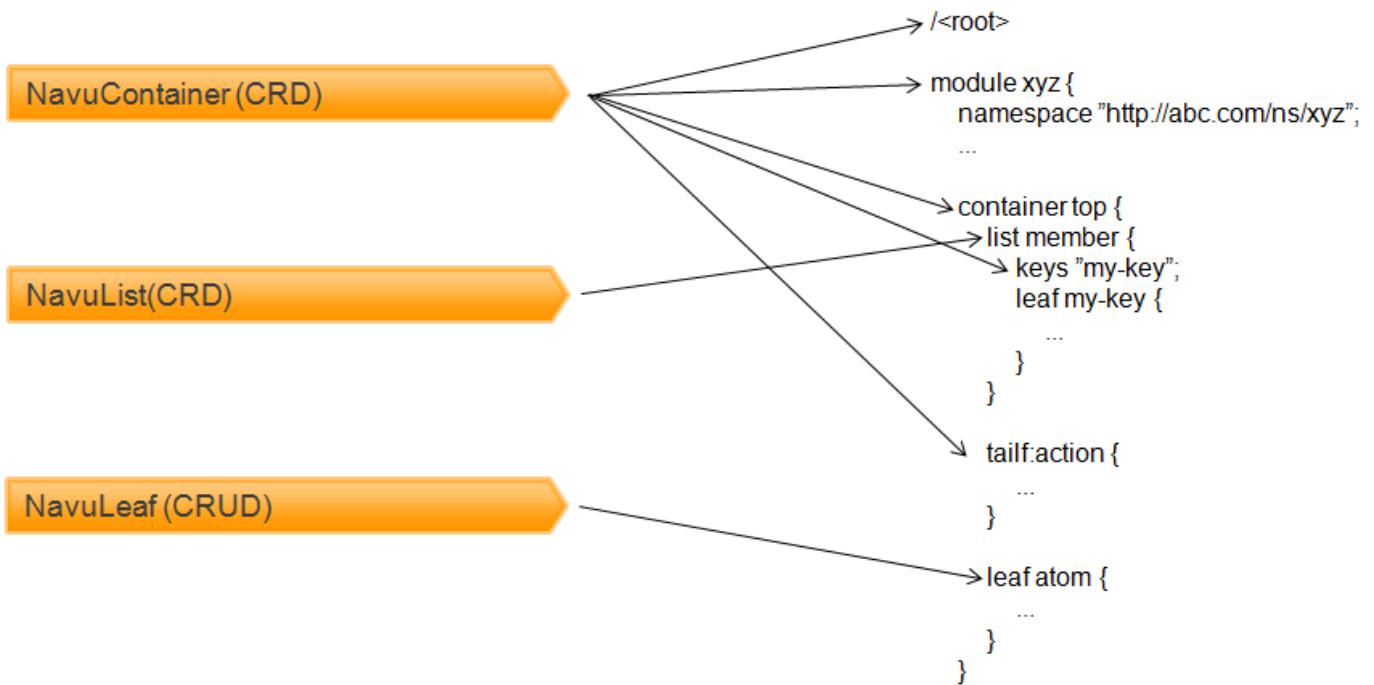
The ncsc tool can also generate java classes from the .yang files. These files, extending the ConfNamespace baseclass, are the java representation of the models and contains all defined nametags and their corresponding hash values. These java classes can, optionally, be used as help classes in the service applications to make NAVU navigation type safe, e.g. eliminating errors from misspelled model container names.

Figure 83. NAVU Design Support

The service models are loaded at start up and are always the latest version. The models are always traversed in a *lazy* fashion i.e. data is only loaded when it is needed. This is in order to minimize the amount of data transferred between NSO and the service applications.

The most important classes of NAVU are the classes implementing the YANG node types. These are used to navigate the DOM. These classes are as follows.

- NavuContainer - the NavuContainer is a container representing either the root of model, a YANG module root, a YANG container.
- NavuList - the NavuList represents a YANG list node.
- NavuListEntry - list node entry.
- NavuLeaf - the NavuLeaf represents a YANG leaf node.

Figure 84. NAVU YANG Structure

The remaining part of this section will guide us through the most useful features of the NAVU. Should further information be required, please refer to the corresponding Javadoc pages.

NAVU relies on MAAPI as an underlying interfaces to access NSO. The starting point in NAVU configuration is to create a NavuContext instance using the `NavuContext(Maapi maapi)` constructor. To read and/or write data a transaction has to be started in Maapi. There are methods in the `NavuContext` class to start and handle this transaction.

If data has to be written the Navu transaction has to be started differently depending on the data being configuration or operational data. Such a transaction is started by the methods `NavuContext.startRunningTrans()` or `NavuContext.startOperationalTrans()` respectively. The Javadoc describes this in more details.

When navigating using NAVU we always start by creating a `NavuContainer` and passing in the `NavuContext` instance, this is a base container from which navigation can be started. Furthermore we need to create a root `NavuContainer` which is the top of the YANG module in which to navigate down. This is done by using the `NavuContainer.container(int hash)` method. Here the argument is the hash value for the modules namespace.

Example 85. NSO Module

```

module tailf-ncs {
    namespace "http://tail-f.com/ns/ncs";
    ...
}
  
```

Example 86. NSO NavuContainer Instance

.....

```

NavuContext context = new NavuContext(maapi);
context.startRunningTrans(Conf.MODE_READ);
// This will be the base container "/"
NavuContainer base = new NavuContainer(context);

// This will be the ncs root container "/ncs"
NavuContainer root = base.container(new Ncs().hash());
.....
// This method finishes the started read transaction and
// clears the context from this transaction.
context.finishClearTrans();

```

NAVU maps the YANG node types; *container*, *list*, *leaf* and *leaf-list* into its own structure. As mentioned previously NavuContainer is used to represent both the *module* and the *container* node type. The NavuListEntry is also used to represent a *list* node instance (actually NavuListEntry extends NavuContainer). I.e. an element of a list node.

Consider the YANG excerpt below.

Example 87. NSO List Element

```

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

        list device {
            key name;

            leaf name {
                type string;
            }
            .....
        }
        .....
    }
}

```

If the purpose is to directly access a list node we would typically do a direct navigation to the list element using the NAVU primitives.

Example 88. NAVU List Direct Element Access

```

.....
NavuContext context = new NavuContext(maapi);
context.startRunningTrans(Conf.MODE_READ);

NavuContainer base = new NavuContainer(context);
NavuContainer ncs = base.container(new Ncs().hash());
NavuContainer dev = ncs.container("devices").
    list("device").
    elem(key);

NavuListEntry devEntry = (NavuListEntry)dev;
.....
context.finishClearTrans();

```

Or if we want to iterate over all elements of a list we could do as follows.

Example 89. NAVU List Element Iterating

```
.....
NavuContext context = new NavuContext(maapi);
context.startRunningTrans(Conf.MODE_READ);

NavuContainer base = new NavuContainer(context);
NavuContainer ncs = base.container(new Ncs().hash());
NavuList listOfDevs = ncs.container("devices").
list("device");

for (NavuContainer dev: listOfDevs.elements()) {
    .....
}

.....
context.finishClearTrans();
```

The above example uses the `select()` which uses a recursive regexp match against its children.

Or alternatively, if the purpose is to drill down deep into a structure we should use `select()`. The `select()` offers a wild card based search. The search is relative and can be performed from any node in the structure.

Example 90. NAVU Leaf Access

```
.....
NavuContext context = new NavuContext(maapi);
context.startRunningTrans(Conf.MODE_READ);

NavuContainer base = new NavuContainer(context);
NavuContainer ncs = base.container(new Ncs().hash());

for (NavuNode node: ncs.container("devices").select("dev.*/*")) {
    NavuContainer dev = (NavuContainer)node;
    .....
}
.....
context.finishClearTrans();
```

All of the above are valid ways of traversing the lists depending on the purpose. If we know what we want, we use direct access. If we want to apply something to a large amount of nodes, we use `select()`.

An alternative method is to use the `xPathSelect()` where a XPath query could be issued instead.

Example 91. NAVU Leaf Access

```
.....
NavuContext context = new NavuContext(maapi);
context.startRunningTrans(Conf.MODE_READ);

NavuContainer base = new NavuContainer(context);
NavuContainer ncs = base.container(new Ncs().hash());

for (NavuNode node: ncs.container("devices").xPathSelect("device/*")) {
    NavuContainer devs = (NavuContainer)node;
    .....
}
.....
context.finishClearTrans();
```

`NavuContainer` and `NavuList` are structural nodes with NAVU. I.e. they have no values. Values are always kept by `NavuLeaf`. A `NavuLeaf` represents the YANG node types *leaf*. A `NavuLeaf` can

be both read and set. NavuLeafList represents the YANG node type *leaf-list* and has some features in common from both NavuLeaf (which it inherits from) and NavuList.

Example 92. NSO Leaf

```
module tailf-ncs {
    namespace "http://tail-f.com/ns/ncs";
    ...
    container ncs {
        .....

        list service {
            key object-id;

            leaf object-id {
                type string;
            }
            ....

            leaf reference {
                type string;
            }
            .....

        }
    }

    .....
}
```

To read and update a leaf we simply navigate to the leaf and request the value. And in the same manner we can update the value.

Example 93. NAVU List Element Iterating

```
.....
NavuContext context = new NavuContext(maapi);
context.startRunningTrans(Conf.MODE_READ);

NavuContainer base = new NavuContainer(context);
NavuContainer ncs = base.container(new Ncs().hash());

for (NavuNode node: ncs.select("sm/ser.*/*")) {
    NavuContainer rfs = (NavuContainer)node;
    if (rfs.leaf(Ncs._description_).value()==null) {
        /*
         * Setting dummy value.
         */
        rfs.leaf(Ncs._description_).set(new ConfBuf("Dummy value"));
    }
}
.....
context.finishClearTrans();
```

In addition to the YANG standard node types NAVU also supports the Tailf proprietary node type *action*. An action is considered being a *NavuAction*. It differs from an ordinary container in that it can be executed using the *call()* primitive. Input and output parameters are represented as ordinary nodes. The action extension of YANG allows an arbitrary structure to be defined both for input and output parameters.

Consider the excerpt below. It represents a module on a managed device. When connected and synchronized to the NSO, the module will appear in the /devices/device/config container.

Example 94. YANG Action

```
module interfaces {
    namespace "http://router.com/interfaces";
    prefix i;
    .....

    list interface {
        key name;
        max-elements 64;

        tailf:action ping-test {
            description "ping a machine ";
            tailf:exec "/tmp/mpls-ping-test.sh" {
                tailf:args "-c $(context) -p $(path)";
            }

            input {
                leaf ttl {
                    type int8;
                }
            }

            output {
                container rcon {
                    leaf result {
                        type string;
                    }
                    leaf ip {
                        type inet:ipv4-address;
                    }
                    leaf ival {
                        type int8;
                    }
                }
            }
        }
    }
}

.....
}

.....
}
```

To execute the action below we need to access a device with this module loaded. This is done in a similar way to non action nodes.

Example 95. NAVU Action Execution

```
.....
NavuContext context = new NavuContext(maapi);
context.startRunningTrans(Conf.MODE_READ);

NavuContainer base = new NavuContainer(context);
NavuContainer ncs = base.container(new Ncs().hash());

/*
 * Execute ping on all devices with the interface module.

```

```

        */
for (NavuNode node: ncs.container(Ncs._devices_).
            select("device/.*/config/interface/.*")) {
    NavuContainer if = (NavuContainer)node;

    NavuAction ping = if.action(interfaces.i_ping_test_);

    /*
     * Execute action.
     */
    ConfXMLParamResult[] result = ping.call(new ConfXMLParam[] {
        new ConfXMLParamValue(new interfaces().hash(),
            interfaces._ttl,
            new ConfInt64(64)));
}

//or we could execute it with XML-String

result = ping.call("<if:ttl>64</if:ttl>");
/*
 * Output the result of the action.
 */
System.out.println("result_ip: "+
((ConfXMLParamValue)result[1]).getValue().toString());

System.out.println("result_ivalue:" +
((ConfXMLParamValue)result[2]).getValue().toString());
}
.....
context.finishClearTrans();

```

Or we could do it with xPathSelect()

Example 96. NAVU Action Execution 2

```

.....
NavuContext context = new NavuContext(maapi);
context.startRunningTrans(Conf.MODE_READ);

NavuContainer base = new NavuContainer(context);
NavuContainer ncs = base.container(new Ncs().hash());

/*
 * Execute ping on all devices with the interface module.
 */
for (NavuNode node: ncs.container(Ncs._devices_).
            xPathSelect("device/config/interface")) {
    NavuContainer if = (NavuContainer)node;

    NavuAction ping = if.action(interfaces.i_ping_test_);

    /*
     * Execute action.
     */
    ConfXMLParamResult[] result = ping.call(new ConfXMLParam[] {
        new ConfXMLParamValue(new interfaces().hash(),
            interfaces._ttl,
            new ConfInt64(64)));
}

//or we could execute it with XML-String

result = ping.call("<if:ttl>64</if:ttl>");


```

```

/*
 * Output the result of the action.
 */
System.out.println("result_ip: " +
((ConfXMLParamValue)result[1]).getValue().toString());

System.out.println("result_ival: " +
((ConfXMLParamValue)result[2]).getValue().toString());
}
.....
context.finishClearTrans();

```

The examples above have described how to attach to the NSO module and navigate through the data model using the NAVU primitives. When using NAVU in the scope of the NSO Service manager, we normally don't have to worry about attaching the NavuContainer to the NSO data model. NSO does this for us providing NavuContainer nodes pointing at the nodes of interest.

ALARM API

Since this API is potent of both producing and consuming alarms, this becomes an API that can be used both north and east bound. It adheres to the NSO Alarm model.

For more information see (Chapter 6, *The Alarm Manager in NSO 5.7 User Guide*)

The `com.tailf.ncs.alarmman.consumer.AlarmSource` class is used to subscribe on alarms. This class establishes a listener towards an alarm subscription server called `com.tailf.ncs.alarmman.consumer.AlarmSourceCentral`. The `AlarmSourceCentral` needs to be instantiated and started prior to the instantiation of the `AlarmSource` listener. The NSO java-vm takes care of starting the `AlarmSourceCentral` so any use of the ALARM API inside the NSO java-vm can expect this server to be running.

For situations where alarm subscription outside of the NSO java-vm is desired, the starting the `AlarmSourceCentral` is performed by opening a Cdb socket, pass this Cdb to the `AlarmSourceCentral` class and then call the `start()` method.

```

// Set up a CDB socket
Socket socket = new Socket("127.0.0.1",Conf.NCS_PORT);
Cdb cdb = new Cdb("my-alarm-source-socket", socket);

// Get and start alarm source - this must only be done once per JVM
AlarmSourceCentral source =
    AlarmSourceCentral.getAlarmSource(10000, cdb);
source.start();

```

To retrieve alarms from the `AlarmSource` listener, a initial `startListening()` is required. Then either a blocking `takeAlarm()` or a timeout based `pollAlarm()` can be used to retrieve the alarms. The first method will wait indefinitely for new alarms to arrive while the second will timeout if an alarm has not arrived in the stipulated time. When a listener no longer is needed then a `stopListening()` call should be issued to deactivate it.

```

AlarmSource mySource = new AlarmSource();
try {
    mySource.startListening();
    // Get an alarms.
    Alarm alarm = mySource.takeAlarm();

    while (alarm != null){
        System.out.println(alarm);

```

```

        for (Attribute attr: alarm.getCustomAttributes()){
            System.out.println(attr);
        }

        alarm = mySource.takeAlarm();
    }

} catch (Exception e) {
    e.printStackTrace();
} finally {
    mySource.stopListening();
}

```

Both the `takeAlarm()` and the `pollAlarm()` method returns a `Alarm` object from which all alarm information can be retrieved.

The `com.tailf.ncs.alarmman.producer.AlarmSink` is used to persistently store alarms in NSO. This can be performed either directly or by the use of a alarm storage server called `com.tailf.ncs.alarmman.producer.AlarmSinkCentral`.

To directly store alarms an `AlarmSink` instance is created using the `AlarmSink(Maapi maapi)` constructor.

```

/*
// Maapi socket used to write alarms directly.
//
Socket socket = new Socket("127.0.0.1",Conf.NCS_PORT);
Maapi maapi = new Maapi(socket);
maapi.startUserSession("system", InetAddress.getByName(host),
                      "system", new String[] {}, 
                      MaapiUserSessionFlag.PROTO_TCP);

AlarmSink sink = new AlarmSink(maapi);

```

On the other hand if the alarms is to be stored using the `AlarmSinkServer` then the `AlarmSink()` constructor without arguments is used.

```
AlarmSink sink = new AlarmSink();
```

However this case requires that the `AlarmSinkServer` is started prior to the instantiation of the `AlarmSink`. The NSO java-vm will take care of starting this server so any use of the ALARM API inside the java-vm can expect this server to be running. If it is desired to store alarms in an application outside of the NSO java vm the `AlarmSinkServer` needs to be started like the following example:

```

/*
// You will need a Maapi socket to write you alarms.
//
Socket socket = new Socket("127.0.0.1",Conf.NCS_PORT);
Maapi maapi = new Maapi(socket);
maapi.startUserSession("system", InetAddress.getByName(host),
                      "system", new String[] {}, 
                      MaapiUserSessionFlag.PROTO_TCP);

AlarmSinkCentral sinkCentral = AlarmSinkCentral.getAlarmSink(1000, maapi);
sinkCentral.start();

```

To store an alarm using the `AlarmSink` an `Alarm` instance must be created. This alarm alarm instance is then stored by a call to the `submitAlarm()` method.

```

ArrayList<AlarmId> idList = new ArrayList<AlarmId>();

ConfIdentityRef alarmType =

```

```

        new ConfidentialityRef(NcsAlarms.hash,
                               NcsAlarms._ncs_dev_manager_alarm);

ManagedObject managedObject1 =
    new ManagedObject("/ncs:devices/device{device0}/config/root1");
ManagedObject managedObject2 =
    new ManagedObject("/ncs:devices/device{device0}/config/root2");

idList.add(new AlarmId(new ManagedDevice("device0"),
                       alarmType,
                       managedObject1));
idList.add(new AlarmId(new ManagedDevice("device0"),
                       alarmType,
                       managedObject2));

ManagedObject managedObject3 =
    new ManagedObject("/ncs:devices/device{device0}/config/root3");

Alarm myAlarm =
    new Alarm(new ManagedDevice("device0"),
              managedObject3,
              alarmType,
              PerceivedSeverity.WARNING,
              false,
              "This is a warning",
              null,
              idList,
              null,
              ConfDatetime.getConfDatetime(),
              new AlarmAttribute(myAlarm.hash,
                                 myAlarm._custom_alarm_attribute_,
                                 new ConfBuf("An alarm attribute")),
              new AlarmAttribute(myAlarm.hash,
                                 myAlarm._custom_status_change_,
                                 new ConfBuf("A status change")));
}

sink.submitAlarm(myAlarm);

```

NOTIF API

Applications can subscribe to certain events generated by NSO. The event types are defined by the `com.tailf.notif.NotificationType` enumeration. The following notification can be subscribed on:

- `NotificationType.NOTIF_AUDIT` - all audit log events are sent from NSO on the event notification socket.
- `NotificationType.NOTIF_COMMIT_SIMPLE` - an event indicating that a user has somehow modified the configuration.
- `NotificationType.NOTIF_COMMIT_DIFF` - an event indicating that a user has somehow modified the configuration. The main difference between this event and the above mentioned `NOTIF_COMMIT_SIMPLE` is that this event is synchronous, i.e. the entire transaction hangs until we have explicitly called `Notif.diffNotificationDone()`. The purpose of this event is to give the applications a chance to read the configuration diffs from the transaction before it commits. A user subscribing to this event can use the MAAPI api to attach `Maapi.attach()` to the running transaction and use `Maapi.diffIterate()` to iterate through the diff.
- `NotificationType.NOTIF_COMMIT_FAILED` - This event is generated when a data provider fails in its commit callback. NSO executes a two-phase commit procedure towards all data providers when committing transactions. When a provider fails in commit, the system is an unknown state. If the

provider is "external", the name of failing daemon is provided. If the provider is another NETCONF agent, the IP address and port of that agent is provided.

- `NotificationType.NOTIF_COMMIT_PROGRESS` - This event provides progress information about the commit of a transaction.
- `NotificationType.NOTIF_PROGRESS` - This event provides progress information about the commit of a transaction or an action being applied. Subscribing to this notification type means that all notifications of the type `NotificationType.NOTIF_COMMIT_PROGRESS` are subscribed to as well.
- `NotificationType.NOTIF_CONFIRMED_COMMIT` - This event is generated when a user has started a confirmed commit, when a confirming commit is issued, or when a confirmed commit is aborted; represented by `ConfirmNotification.confirm_type`. For a confirmed commit, the timeout value is also present in the notification.
- `NotificationType.NOTIF_FORWARD_INFO` - This event is generated whenever the server forwards (proxies) a northbound agent.
- `NotificationType.NOTIF_HA_INFO` - an event related to NSOs perception of the current cluster configuration.
- `NotificationType.NOTIF_HEARTBEAT` - This event can be used by applications that wish to monitor the health and liveness of the server itself. It needs to be requested through a Notif instance which has been constructed with a `heartbeat_interval`. The server will continuously generate heartbeat events on the notification socket. If the server fails to do so, the server is hung. The timeout interval is measured in milli seconds. Recommended value is 10000 milli seconds to cater for truly high load situations. Values less than 1000 are changed to 1000.
- `NotificationType.NOTIF_SNMPA` - This event is generated whenever an SNMP pdu is processed by the server. The application receives an `SnmpaNotification` with a list of all varbinds in the pdu. Each varbind contains subclasses that are internal to the `SnmpaNotification`.
- `NotificationType.NOTIF_SUBAGENT_INFO` - only sent if NSO runs as a primary agent with subagents enabled. This event is sent when the subagent connection is lost or reestablished. There are two event types, defined in `SubagentNotification.subagent_info_type`: "subagent up" and "subagent down".
- `NotificationType.NOTIF_DAEMON` - all log events that also goes to the `/NCSConf/logs/NSCLog` log are sent from NSO on the event notification socket.
- `NotificationType.NOTIF_NETCONF` - all log events that also goes to the `/NCSConf/logs/netconfLog` log are sent from NSO on the event notification socket.
- `NotificationType.NOTIF_DEVEL` - all log events that also goes to the `/NCSConf/logs/develLog` log are sent from NSO on the event notification socket.
- `NotificationType.NOTIF_TAKEOVER_SYSLOG` - If this flag is present, NSO will stop syslogging. The idea behind the flag is that we want to configure syslogging for NSO in order to let NSO log its startup sequence. Once NSO is started we wish to subsume the syslogging done by NSO. Typical applications that use this flag want to pick up all log messages, reformat them and use some local logging method. Once all subscriber sockets with this flag set are closed, NSO will resume to syslog.
- `NotificationType.NOTIF_UPGRADE_EVENT` - This event is generated for the different phases of an in-service upgrade, i.e. when the data model is upgraded while the server is running. The application receives an `UpgradeNotification` where the `UpgradeNotification.event_type` gives the specific upgrade event. The events correspond to the invocation of the Maapi functions that drive the upgrade.
- `NotificationType.NOTIF_USER_SESSION` - an event related to user sessions. There are 6 different user session related event types, defined in `UserSessNotification.user_sess_type`: session starts/stops, session locks/unlocks database, session starts/stop database transaction.

To receive events from the NSO the application opens a socket and passes it to the notification base class `com.tailf.notif.Notif` together with an `EnumSet` of `NotificationType` for all types of notifications

that should be received. Looping over the `Notif.read()` method will read and deliver notification which are all subclasses of the `com.tailf.notif.Notification` base class.

```
Socket sock = new Socket("localhost", Conf.NCS_PORT);
EnumSet notifSet = EnumSet.of(NotificationType.NOTIF_COMMIT_SIMPLE,
                                NotificationType.NOTIF_AUDIT);
Notif notif = new Notif(sock, notifSet);

while (true) {
    Notification n = notif.read();

    if (n instanceof CommitNotification) {
        // handle NOTIF_COMMIT_SIMPLE case
        ....
    } else if (n instanceof AuditNotification) {
        // handle NOTIF_AUDIT case
        ....
    }
}
```

HA API

The HA API is used to setup and control High Availability cluster nodes. This package is used to connect to the High Availability (HA) subsystem. Configuration data can then be replicated on several nodes in a cluster. (Chapter 7, *High Availability* in *NSO 5.7 Administration Guide*)

The following example configures three nodes in a HA cluster. One is set as primary and the other two as secondaries.

Example 97. HA cluster setup

```
.....

Socket s0 = new Socket("host1", Conf.NCS_PORT);
Socket s1 = new Socket("host2", Conf.NCS_PORT);
Socket s2 = new Socket("host3", Conf.NCS_PORT);

Ha ha0 = new Ha(s0, "clus0");
Ha ha1 = new Ha(s1, "clus0");
Ha ha2 = new Ha(s2, "clus0");

ConfHaNode primary =
    new ConfHaNode(new ConfBuf("node0"),
                  new ConfIPv4(InetAddress.getByName("localhost")));

ha0.beMaster(primary.nodeid);

ha1.beSlave(new ConfBuf("node1"), primary, true);

ha2.beSlave(new ConfBuf("node2"), primary, true);

HaStatus status0 = ha0.status();
HaStatus status1 = ha1.status();
HaStatus status2 = ha2.status();

....
```

Java API Conf Package

This section describes the types and how these types maps to various YANG types and java classes.

All types inherits the baseclass `com.tailf.conf.ConfObject`.

Following the type hierarchy of `ConfObject` subclasses are distinguished by:

- *Value* - A concrete value classes which inherits `ConfValue` that in turn is a subclass of `ConfObject`.
- *TypeDescriptor* - a class representing the type of a `ConfValue`. A type-descriptor is represented as an instance of `ConfTypeDescriptor`. Usage is primarily to be able to map a `ConfValue` to its internal integer value representation or vice versa.
- *Tag* - A tag is representation of an element in the YANG model. A Tag is represented as an instance of `com.tailf.conf.Tag`. The primary usage of tags are in the representation of keypaths.
- *Key* - a key is a representation of the instance key for a element instance. A key is represented as an instance of `com.tailf.conf.ConfKey`. A `ConfKey` is constructed from an array of values (`ConfValue[]`). The primary usage of keys are in the representation of keypaths.
- *XMLParam* - subclasses of `ConfXMLParam` which are used to represent a, possibly instantiated, subtree of a YANG model. Useful in several APIs where multiple values can be set or retrieved in one function call.

The class `ConfObject` defines public int constants for the different value types. Each value type are mapped to a specific YANG type and are also represented by a specific subtype of `ConfValue`. Having a `ConfValue` instance it is possible to retrieve its integer representation by the use of the static method `getConfTypeDescriptor()` in class `ConfTypeDescriptor`. This functions returns a `ConfTypeDescriptor` instance representing the value from which the integer representation can be retrieved. The values represented as integers are:

Table 98. ConfValue types

Constant	YANG type	ConfValue	Description
J_STR	string	ConfBuf	Human readable string
J_BUF	string	ConfBuf	Human readable string
J_INT8	int8	ConfInt8	8-bit signed integer
J_INT16	int16	ConfInt16	16-bit signed integer
J_INT32	int32	ConfInt32	32-bit signed integer
J_INT64	int64	ConfInt64	64-bit signed integer
J_UINT8	uint8	ConfUInt8	8-bit unsigned integer
J_UINT16	uint16	ConfUInt16	16-bit unsigned integer
J_UINT32	uint32	ConfUInt32	32-bit unsigned integer
J_UINT64	uint64	ConfUInt64	64-bit unsigned integer
J_IPV4	inet:ipv4-address	ConfIPv4	64-bit unsigned
J_IPV6	inet:ipv6-address	ConfIPv6	IP v6 Address
J_BOOL	boolean	ConfBoolean	Boolean value
J_QNAME	xs:QName	ConfQName	A namespace/tag pair
J_DATETIME	yang:date-and-time	ConfDateTime	Date and Time Value

Constant	YANG type	ConfValue	Description
J_DATE	xs:date	ConfDate	XML schema Date
J_ENUMERATION	enum	ConfEnumeration	An enumeration value
J_BIT32	bits	ConfBit32	32 bit value
J_BIT64	bits	ConfBit64	64 bit value
J_LIST	leaf-list	-	-
J_INSTANCE_IDENTIFIER	instance-identifier	ConfObjectRef	yang builtin
J_OID	tailf:snmp-oid	ConfOID	-
J_BINARY	tailf:hex-list, tailf:octet-list	ConfBinary, ConfHexList	-
J_IPV4PREFIX	inet:ipv4-prefix	ConfIPv4Prefix	-
J_IPV6PREFIX	-	ConfIPv6Prefix	-
J_IPV6PREFIX	inet:ipv6-prefix	ConfIPv6Prefix	-
J_DEFAULT	-	ConfDefault	default value indicator
J_NOEXISTS	-	ConfNoExists	no value indicator
J_DECIMAL64	decimal64	ConfDecimal64	yang builtin
J_IDENTITYREF	identityref	ConfIdentityRef	yang builtin

An important class in the `com.tailf.conf` package, not inheriting `ConfObject`, is `ConfPath`. `ConfPath` is used to represent a keypath which can point to any element in an instantiated model. As such it is constructed from an array of `ConfObject[]` instances where each element is expected to be either a `ConfTag` or a `ConfKey`.

As an example take the keypath `/ncs:devices/device{d1}/iosxr:interface/Loopback{lo0}`. The following code snippets shows the instantiating of a `ConfPath` object representing this keypath:

```
ConfPath keyPath = new ConfPath(new ConfObject[] {
    new ConfTag("ncs", "devices"),
    new ConfTag("ncs", "device"),
    new ConfKey(new ConfObject[] {
        new ConfBuf("d1")}),
    new ConfTag("iosxr", "interface"),
    new ConfTag("iosxr", "Loopback"),
    new ConfKey(new ConfObject[] {
        new ConfBuf("lo0")})
});
```

Another, more commonly used option is to use the format string + arguments constructor from `ConfPath`. Where `ConfPath` parses and creates the `ConfTag/ConfKey` representation from the string representation instead.

```
// either this way
ConfPath key1 = new ConfPath("/ncs:devices/device{d1}"+
                           "/iosxr:interface/Loopback{lo0}")
// or this way
ConfPath key2 = new ConfPath("/ncs:devices/device{%s}"+
                           "/iosxr:interface/Loopback{%s}",
                           new ConfBuf("d1"),
```

```
new ConfBuf("100"));
```

The usage of ConfXMLParam is in tagged value arrays ConfXMLParam[] of subtypes of ConfXMLParam. These can in collaboration represent an arbitrary YANG model subtree. It does not view a node as a path but instead it behaves as an XML instance document representation. We have 4 subtypes of ConfXMLParam:

- ConfXMLParamStart - Represents an opening tag. Opening node of a container or list entry.
- ConfXMLParamStop - Represents an closing tag. Closing tag of a container or a list entry.
- ConfXMLParamValue - Represent a value and a tag. Leaf tag with the corresponding value.
- ConfXMLParamLeaf - Represents a leaf tag without the leafs value.

Each element in the array is associated with the the node in the data model.

The array corresponding to the /servers/server{www} which is representation of the instance XML document:

```
<servers>
  <server>
    <name>www</name>
  </server>
</servers>
```

The list entry above could be populated as:

```
ConfXMLParam[] tree = new ConfXMLParam[] {
    new ConfXMLParamStart(ns.hash(), ns._servers),
    new ConfXMLParamStart(ns.hash(), ns._server),
    new ConfXMLParamValue(ns.hash(), ns._name),
    new ConfXMLParamStop(ns.hash(), ns._server),
    new ConfXMLParamStop(ns.hash(), ns._servers)};
```

Namespace classes and the loaded schema

A namespace class represents the namespace for a YANG module. As such if maps the symbol name of each element in the YANG module to its corresponding hash value.

A namespace class is a subclass of ConfNamespace and comes in one of two shapes. Either created at compile time using the ncsc compiler or created at runtime with the use of Maapi.loadSchemas. These two types also indicates two main usages of namespace classes. The first is in programming where the symbol name are used e.g. in Navu navigation. This is where the compiled namespaces are used. The other is for internal mapping between symbol names and hash values. This is were the runtime type normally are used, however compiled namespace classes can be used for these mappings too.

The compiled namespace classes are generated from compiled .fxs files through ncsc,(ncsc --emit-java).

```
ncsc --java-disable-prefix --java-package \
      com.example.app.namespaces \
      --emit-java \
      java/src/com/example/app/namespaces/foo.java \
      foo.fxs
```

Runtime namespace classes are created by calling Maapi.loadschema(). Thats it, the rest is dynamic. All namespaces known by NSO are downloaded and runtime namespace classes are created. these can be retrieved by calling Maapi.getAutoNsList()

```
Socket s = new Socket("localhost", Conf.NCS_PORT);
Maapi maapi = new Maapi(s);
```

Namespace classes and the loaded schema

```
maapi.loadSchemas();
ArrayList<ConfNamespace> nsList = maapi.getAutoNsList();
```

The schema information is loaded automatically at first connect of the NSO server so no manually method call to `Maapi.loadSchemas()` is needed.

With all schemas loaded, the java engine can make mappings between hash codes and symbol names on the fly. Also the `ConfPath` class can find and add namespace information when parsing keypaths provided that the namespace prefixes are added in the start element for each namespace.

```
ConfPath key1 = new ConfPath("/ncs:devices/device{d1}/iosxr:interface");
```

As an option, several APIs e.g. MAAPi have the possibility to set the default namespace which will be the expected namespace for paths without prefixes. For example if the namespace class smp is generated with the legal path "/smp:servers/server" an option in maapi could be the following:

```
Socket s = new Socket("localhost", Conf.NCS_PORT);
Maapi maapi = new Maapi(s);
int th = maapi.startTrans(Conf.DB_CANDIDATE,
                           Conf.MODE_READ_WRITE);

// Because we will use keypaths without prefixes
maapi.setNamespace(th, new smp().uri());

ConfValue val = maapi.getElem(th, "/devices/device{d1}/address");
```



CHAPTER 11

Python API Overview

- [Introduction, page 173](#)
- [Python API overview, page 174](#)
- [Python scripting, page 175](#)
- [High-level MAAPI API, page 175](#)
- [Maagic API, page 176](#)
- [Maagic examples, page 182](#)
- [PlanComponent, page 185](#)
- [Python packages, page 186](#)
- [Low-level APIs, page 189](#)

Introduction

The NSO Python library contains a variety of APIs for different purposes. In this chapter we introduce these and explain their usage. The NSO Python modules deliverables are found in two variants, the low-level APIs and the high-level APIs.

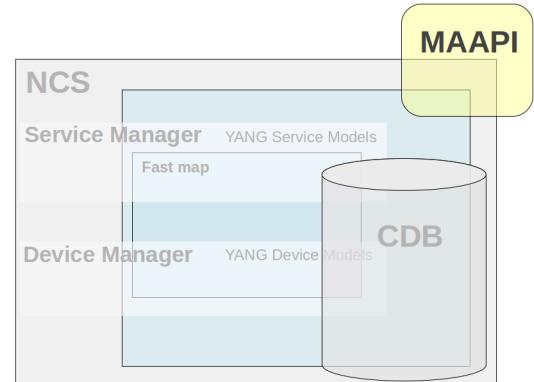
The low-level APIs is a direct mapping of the NSO C APIs, CDB and MAAPI. These will follow the evolution of the C APIs. See **man confd_lib_lib** for further information.

The high-level APIs is an abstraction layer on top of the low-level APIs to make them easier to use, to improve code readability and development rate for common use cases. E.g. services and action callbacks and common scripting towards NSO.

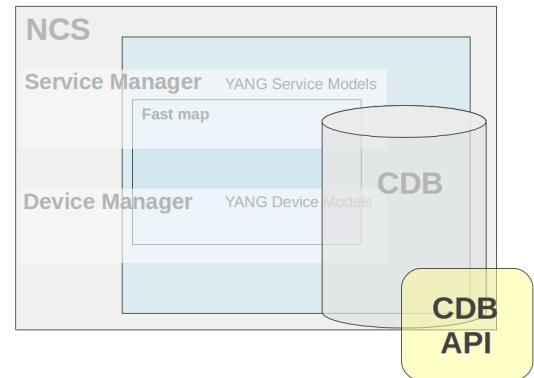
Python versions 3.7 or higher are supported. There are no dependencies to external modules.

Python API overview

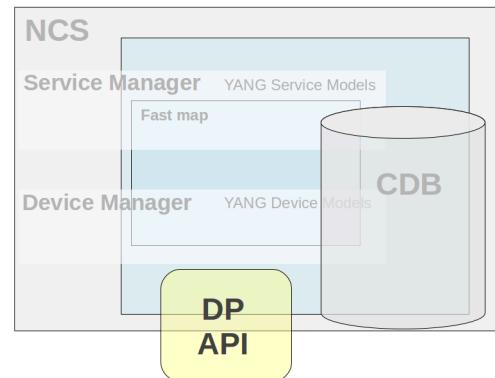
MAAPI - (Management Agent API) Northbound interface that is transactional and user session based. Using this interface, both configuration and operational data can be read. Configuration and operational data can be written and committed as one transaction. The API is complete in the way that it is possible to write a new northbound agent using only this interface. It is also possible to attach to ongoing transactions in order to read uncommitted changes and/or modify data in these transactions.



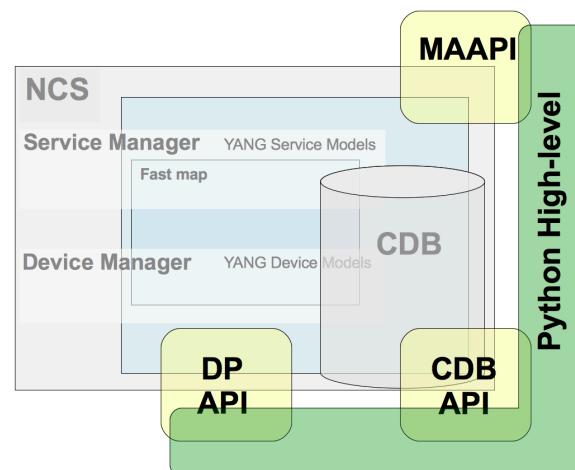
Python low-level CDB API - Southbound interface provides access to the CDB configuration database. Using this interface, configuration data can be read. In addition, operational data that is stored in CDB can be read and written. This interface has a subscription mechanism to subscribe to changes. A subscription is specified on an path that points to an element in a YANG model or an instance in the instance tree. Any change under this point will trigger the subscription. CDB has also functions to iterate through the configuration changes when a subscription has triggered.



Python low-level DP API - Southbound interface that enables callbacks, hooks and transforms. This API makes it possible to provide the service callbacks that handles service to device mapping logic. Other usual cases are external data providers for operational data or action callback implementations. There are also transaction and validation callbacks, etc. Hooks are callbacks that are fired when certain data is written and the hook is expected to do additional modifications of data. Transforms are callbacks that are used when complete mediation between two different models is necessary.



Python high-level API - API that resides on top of the MAAPI, CDB, and DP APIs. It provides schema model navigation and instance data handling (read/write). Uses a MAAPI context as data access and incorporates its functionality. It is used in service implementations, action handlers and Python scripting.



Python scripting

Scripting in Python is a very easy and powerful way of accessing NSO. This document has several examples of scripts showing various ways in accessing data and requesting actions in NSO.

The examples are directly executable with the python interpreter after sourcing the `ncsrc` file in the NSO installation directory. This sets up the `PYTHONPATH` environment variable, which enables access to the NSO Python modules.

Edit a file and execute it directly on the command line like this:

```
$ python3 script.py
```

High-level MAAPI API

The Python high-level MAAPI API provides an easy to use interface for accessing NSO. Its main targets is to encapsulate the sockets, transaction handles, data type conversions and the possibility to use the Python `with` statement for proper resource cleanup.

The simplest way to access NSO is to use the `single_transaction` helper. It creates a MAAPI context and a transaction in one step.

This example shows its usage, connecting as user 'admin' and 'python' as AAA context:

Example 99. Single transaction helper

```
import ncs

with ncs.maapi.single_write_trans('admin', 'python') as t:
    t.set_elem2('Kilroy was here', '/ncs:devices/device{ce0}/description')
    t.apply()

with ncs.maapi.single_read_trans('admin', 'python') as t:
    desc = t.get_elem('/ncs:devices/device{ce0}/description')
    print("Description for device ce0 = %s" % desc)
```

A common use case is to create a MAAPI context and re-use it for several transactions. This reduces the latency and increases the transaction throughput, especially for back-end applications. For scripting the lifetime is shorter and there is no need to keep the MAAPI contexts alive.

This example shows how to keep a MAAPI connection alive between transactions:

Example 100. Reading of configuration data using high-level MAAPI

```
import ncs

with ncs.maapi.Maapi() as m:
    with ncs.maapi.Session(m, 'admin', 'python'):

        # The first transaction
        with m.start_read_trans() as t:
            address = t.get_elem('/ncs:devices/device{ce0}/address')
            print("First read: Address = %s" % address)

        # The second transaction
        with m.start_read_trans() as t:
            address = t.get_elem('/ncs:devices/device{ce1}/address')
            print("Second read: Address = %s" % address)
```

Maagic API

Maagic is a module provided as part of the NSO Python APIs. It reduces the complexity of programming towards NSO, is used on top of the MAAPI high-level API and addresses areas which require more programming. First it helps in navigating in the model, using standard Python object dot notation, giving very clear and easily read code. The context handlers removes the need to close sockets, user sessions and transactions and the problems when they are forgotten and kept open. Finally it removes the need to know the data types of the leafs, helping you to focus on the data to be set.

When using Maagic you still do the same procedure of starting a transaction.

```
with ncs.maapi.Maapi() as m:
    with ncs.maapi.Session(m, 'admin', 'python'):
        with m.start_write_trans() as t:
            # Read/write/request ...
```

To use the Maagic functionality you get access to a Maagic object either pointing to the root of the CDB:

```
root = ncs.maagic.get_root(t)
```

In this case it is a `ncs.maagic.Node` object with a `ncs.maapi.Transaction` back-end.

From here you can navigate in the model. In the table you can see examples how to navigate.

Table 101. Maagic object navigation

Action	Returns
<code>root.devices</code>	Container
<code>root.devices.device</code>	List
<code>root.devices.device['ce0']</code>	ListElement
<code>root.devices.device['ce0'].device_type.cli</code>	PresenceContainer
<code>root.devices.device['ce0'].address</code>	str
<code>root.devices.device['ce0'].port</code>	int

You can also get a Maagic object from a keypath:

```
node = ncs.maagic.get_node(t, '/ncs:devices/device{ce0}')
```

Namespaces

Maagic handles namespaces by a prefix to the names of the elements. This is optional, but recommended to avoid future side effects.

The syntax is to prefix the names with the namespace name followed by two underscores, e.g. `ns_name__name`.

Examples how to use namespaces:

```
# The examples are equal unless there is a namespace collision.  
# For the ncs namespace it would look like this:
```

```
root.ncs__devices.ncs__device['ce0'].ncs__address  
# equals  
root.devices.device['ce0'].address
```

In cases where there is a name collision, the namespace prefix is required to access an entity from a module, except for the module that was first loaded. Namespace is always required for root entities when there is a collision. The module load order is found in the ncs log file: logs/ncs.log.

```
# This example have three namespaces referring to a leaf, value, with the same  
# name and this load order: /ex/a:value=11, /ex/b:value=22 and /ex/c:value=33  
  
root.ex.value # returns 11  
root.ex.a__value # returns 11  
root.ex.b__value # returns 22  
root.ex.c__value # returns 33
```

Reading data

Reading data using Maagic is straight forward. You will just specify the leaf you are interested in and the data is retrieved. The data is returned in the nearest available Python data type.

For non-existing leafs, `None` is returned.

```
dev_name = root.devices.device['ce0'].name # 'ce0'  
dev_address = root.devices.device['ce0'].address # '127.0.0.1'  
dev_port = root.devices.device['ce0'].port # 10022
```

Writing data

Writing data using Maagic is straightforward. You will just specify the leaf you are interested in and assign a value. Any data type can sent as input, as the `str` function is called, converting it to a string. The format is depending on the data type. If the type validation fails an `Error` exception is thrown.

```
root.devices.device['ce0'].name = 'ce0'
root.devices.device['ce0'].address = '127.0.0.1'
root.devices.device['ce0'].port = 10022
root.devices.device['ce0'].port = '10022' # Also valid

# This will raise an Error exception
root.devices.device['ce0'].port = 'netconf'
```

Deleting data

Data is deleted the Python way of using the `del` function:

```
del root.devices.device['ce0'] # List element
del root.devices.device['ce0'].name # Leaf
del root.devices.device['ce0'].device_type.cli # Presence container
```

Some entities have a delete method, this is explained under the corresponding type.



The delete mechanism in Maagic is implemented using the `__delattr__` method on the `Node` class. This means that executing the `del` function on a local or global variable will only delete the object from the Python local or global namespaces. E.g. `del obj`.

Containers

Containers are addressed using standard Python dot notation: `root.container1.container2`

Presence containers

A presence container is created using the `create` method:

```
pc = root.container.presence_container.create()
```

Existence is checked with the `exists` or `bool` functions:

```
root.container.presence_container.exists() # Returns True or False
bool(root.container.presence_container) # Returns True or False
```

A presence container is deleted with the `del` or `delete` functions:

```
del root.container.presence_container
root.container.presence_container.delete()
```

Choices

The case of a choice is checked by addressing the name of the choice in the model:

```
ne_type = root.devices.device['ce0'].device_type.ne_type
if ne_type == 'cli':
    # Handle CLI
elif ne_type == 'netconf':
    # Handle NETCONF
elif ne_type == 'generic':
```

```

    # Handle generic
else:
    # Don't handle

```

Changing a choice is done by setting a value in any of the other cases:

```

root.devices.device['ce0'].device_type.netconf.create()
str(root.devices.device['ce0'].device_type.ne_type) # Returns 'netconf'

```

Lists and List elements

List elements are created using the `create` method on the `List` class:

```

# Single value key
ce5 = root.devices.device.create('ce5')

# Multiple values key
o = root.container.list.create('foo', 'bar')

```

The objects `ce5` and `o` above are of type `ListElement` which is actually an ordinary `Container` object with a different name.

Existence is checked with the `exists` or `bool` functions `List` class:

```
'ce0' in root.devices.device # Returns True or False
```

A list element is deleted with the python `del` function:

```

# Single value key
del root.devices.device['ce5']

# Multiple values key
del root.container.list['foo', 'bar']

```

To delete the whole list use the python `del` function or `delete()` on the list.

```

# use Python's del function
del root.devices.device

# use List's delete() method
root.container.list.delete()

```

Unions

Unions are not handled in any specific way, you just read or write to the leaf and the data is validated according to the model.

Enumeration

Enumerations are returned as an `Enum` object, giving access to both the integer and string values.

```

str(root.devices.device['ce0'].state.admin_state) # May return 'unlocked'
root.devices.device['ce0'].state.admin_state.string # May return 'unlocked'
root.devices.device['ce0'].state.admin_state.value # May return 1

```

Writing values to enumerations accepts both string and integer values.

```

root.devices.device['ce0'].state.admin_state = 'locked'
root.devices.device['ce0'].state.admin_state = 0

# This will raise an Error exception
root.devices.device['ce0'].state.admin_state = 3 # Not a valid enum

```

Leafref

Leafrefs are read as regular leafs and the returned data type corresponds to the referred leaf.

```
# /model/device is a leafref to /devices/device/name
dev = root.model.device # May return 'ce0'
```

Leafrefs are set as the leaf it refers. Data type is validated as it is set. The reference is validated when the transaction is committed.

```
# /model/device is a leafref to /devices/device/name
root.model.device = 'ce0'
```

Identityref

Identityrefs are read and written as string values. Writing an identityref without prefix is possible, but doing so is error prone and may stop working if another model is added which also has an identity with the same name. The recommendation is to always use prefix when writing identityrefs. Reading an identityref will always return a prefixed string value.

```
# Read
root.devices.device['ce0'].device_type.cli.ned_id # May return 'ios-id:cisco-ios'

# Write when identity cisco-ios is unique throughout the system (not recommended)
root.devices.device['ce0'].device_type.cli.ned_id = 'cisco-ios'

# Write with unique identity
root.devices.device['ce0'].device_type.cli.ned_id = 'ios-id:cisco-ios'
```

Instance-identifier

Instance-identifiers are read as xpath formatted string values.

```
# /model/iref is an instance-identifier
root.model.iref # May return "/ncs:devices/ncs:device[ncs:name='ce0']"
```

Instance-identifiers are set as xpath formatted strings. The string is validated as it is set. The reference is validated when the transaction is committed.

```
# /model/iref is an instance-identifier
root.devices.device['ce0'].device_type.cli.ned_id = "/ncs:devices/ncs:device[ncs:name='ce0']"
```

Leaf-list

A leaf-list is represented by a *LeafList* object. This object behaves very much like a Python list. You may iterate it, check for existence of a specific element using *in*, remove specific items using the *del* operator. See examples below.

N.B. From NSO version 4.5 and onwards a yang leaf-list is represented differently than before. Reading a leaf-list using Maagic used to result in an ordinary Python list (or None if the leaf-list was non-existent). Now, reading a leaf-list will give back a LeafList object whether it exists or not. The LeafList object may be iterated like a Python list and you may check for existence using the *exists()* method or the *bool()* operator. A Maagic leaf-list node may be assigned using a Python list, just like before, and you may convert it to a Python list using the *as_list()* method or by doing *list(my_leaf_list_node)*.

You should update your code to cope with the new behaviour. If you for any reason are unable to do so you can instruct Maagic to behave as in previous versions by setting the environment variable `DEPRECATED_MAAGIC_WANT_LEAF_LIST_AS_LEAF` to 'true', 'yes' or '1' before starting your Python process (or NSO).

Please note that this environment variable is deprecated and will go away in the future.

```
# /model/ll is a leaf-list with the type string

# read a LeafList object
ll = root.model.ll

# iteration
for item in root.model.ll:
    do_stuff(item)

# check if the leaf-list exists (i.e. is non-empty)
if root.model.ll:
    do_stuff()
if root.model.ll.exists():
    do_stuff()

# check the leaf-list contains a specific item
if 'foo' in root.model.ll:
    do_stuff()

# length
len(root.model.ll)

# create a new item in the leaf-list
root.model.ll.create('bar')

# set the whole leaf-list in one operation
root.model.ll = ['foo', 'bar', 'baz']

# remove a specific item from the list
del root.model.ll['bar']
root.model.ll.remove('baz')

# delete the whole leaf-list
del root.model.ll
root.model.ll.delete()

# get the leaf-list as a Python list
root.model.ll.as_list()
```

Binary

Binary values are read and written as byte strings.

```
# Read
root.model.bin # May return '\x00foo\x01bar'

# Write
root.model.bin = b'\x00foo\x01bar'
```

Bits

Reading a bits leaf will give a Bits object back (or None if the bits leaf is non-existent). To get some useful information out of the Bits object you can either use the `bytarray()` method to get a Python `bytarray` object in return or the Python `str()` operator to get a space separated string containing the bit names.

```
# read a bits leaf - a Bits object may be returned (None if non-existent)
root.model.bits

# get a bytearray
root.model.bits bytearray()

# get a space separated string with bit names
str(root.model.bits)
```

There are four ways of setting a bits leaf. One is to set it using a string with space separated bit names, the other one is to set it using a bytearray, the third by using a Python binary string and as a last option it may be set using a Bits object. Note that updating a Bits object does not change anything in the database - for that to happen you need to assign it to the Maagic node.

```
# set a bits leaf using a string of space separated bit names
root.model.bits = 'turboMode enableEncryption'

# set a bits leaf using a Python bytearray
root.model.bits = bytearray(b'\x11')

# set a bits leaf using a Python binary string
root.model.bits = b'\x11'

# read a bits leaf, update the Bits object and set it
b = x.model.bits
b.clr_bit(0)
x.model.bits = b
```

Empty leaf

An empty leaf is created using the create method:

```
pc = root.container.empty_leaf.create()
```

Existence is checked with the exists or bool functions:

```
root.container.empty_leaf.exists() # Returns True or False
bool(root.container.empty_leaf) # Returns True or False
```

An empty leaf is deleted with the del or delete functions:

```
del root.container.empty_leaf
root.container.empty_leaf.delete()
```

Maagic examples

Action requests

Requesting an action may not require an ongoing transaction and this example shows how to use Maapi as a transactionless back-end for Maagic.

Example 102. Action request without transaction

```
import ncs

with ncs.maapi.Maapi() as m:
    with ncs.maapi.Session(m, 'admin', 'python'):
        root = ncs.maagic.get_root(m)
```

```

        output = root.devices.check_sync()

    for result in output.sync_result:
        print('sync-result {')
        print('    device %s' % result.device)
        print('    result %s' % result.result)
        print('}')

```

This example shows how to request an action that require an ongoing transaction. It is also valid to request an action that does not require an ongoing transaction.

Example 103. Action request with transaction

```

import ncs

with ncs.maapi.Maapi() as m:
    with ncs.maapi.Session(m, 'admin', 'python'):
        with m.start_read_trans() as t:
            root = ncs.maagic.get_root(t)

            output = root.devices.check_sync()

        for result in output.sync_result:
            print('sync-result {')
            print('    device %s' % result.device)
            print('    result %s' % result.result)
            print('}')

```

Providing parameters to an action with Maagic is very easy. You request an input object, with `get_input` from the Maagic action object and sets the desired (or required) parameters as defined in the model specification.

Example 104. Action request with input parameters

```

import ncs

with ncs.maapi.Maapi() as m:
    with ncs.maapi.Session(m, 'admin', 'python'):
        root = ncs.maagic.get_root(m)

        input = root.action.double.get_input()
        input.number = 21
        output = root.action.double(input)

        print(output.result)

```

If you have a leaf-list you need to prepare the input parameters

Example 105. Action request with leaf-list input parameters

```

import ncs

with ncs.maapi.Maapi() as m:
    with ncs.maapi.Session(m, 'admin', 'python'):
        root = ncs.maagic.get_root(m)

        input = root.leaf_list_action.llist.get_input()
        input.args = ['testing action']
        output = root.leaf_list_action.llist(input)

```

```
    print(output.result)
```

A common use case is to script creation of devices. With the Python APIs this is easily done without the need to generate set commands and execute them in the CLI.

Example 106. Create device, fetch host keys and synchronize configuration

```
import argparse
import ncs

def parseArgs():
    parser = argparse.ArgumentParser()
    parser.add_argument('--name', help="device name", required=True)
    parser.add_argument('--address', help="device address", required=True)
    parser.add_argument('--port', help="device address", type=int, default=22)
    parser.add_argument('--desc', help="device description",
                        default="Device created by maagic_create_device.py")
    parser.add_argument('--auth', help="device authgroup", default="default")
    return parser.parse_args()

def main(args):
    with ncs.maapi.Maapi() as m:
        with ncs.maapi.Session(m, 'admin', 'python'):
            with m.start_write_trans() as t:
                root = ncs.maagic.get_root(t)

                print("Setting device '%s' configuration..." % args.name)

                # Get a reference to the device list
                device_list = root.devices.device

                device = device_list.create(args.name)
                device.address = args.address
                device.port = args.port
                device.description = args.desc
                device.authgroup = args.auth
                dev_type = device.device_type.cli
                dev_type.ned_id = 'cisco-ios-cli-3.0'
                device.state.admin_state = 'unlocked'

                print('Committing the device configuration...')
                t.apply()
                print("Committed")

                # This transaction is no longer valid

                #
                # fetch-host-keys and sync-from does not require a transaction
                # continue using the Maapi object
                #
                root = ncs.maagic.get_root(m)
                device = root.devices.device[args.name]

                print("Fetching SSH keys...")
                output = device.ssh.fetch_host_keys()
                print("Result: %s" % output.result)

                print("Syncing configuration...")
                output = device.sync_from()
```

```

        print("Result: %s" % output.result)
        if not output.result:
            print("Error: %s" % output.info)

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

```

PlanComponent

This class is a helper to support service progress reporting using plan-data as part of a Reactive FASTMAP service. More info about plan-data is found in [the section called “Progress reporting using plan-data”](#).

The interface of the PlanComponent is identical to the corresponding Java class and supports the setup of plans and setting the transition states.

```

class PlanComponent(object):
    """Service plan component.

    The usage of this class is in conjunction with a service that
    uses a reactive FASTMAP pattern.
    With a plan the service states can be tracked and controlled.

    A service plan can consist of many PlanComponent's.
    This is operational data that is stored together with the service
    configuration.
    """

    def __init__(self, service, name, component_type):
        """Initialize a PlanComponent."""

    def append_state(self, state_name):
        """Append a new state to this plan component.

        The state status will be initialized to 'ncs:not-reached'.
        """

    def set_reached(self, state_name):
        """Set state status to 'ncs:reached'."""

    def set_failed(self, state_name):
        """Set state status to 'ncs:failed'."""

    def set_status(self, state_name, status):
        """Set state status."""

```

See [pydoc3 ncs.application.PlanComponent](#) for further information about the Python class.

The pattern is to add an overall plan (self) for the service and separate plans for each component that builds the service.

```

self_plan = PlanComponent(service, 'self', 'ncs:self')
self_plan.append_state('ncs:init')
self_plan.append_state('ncs:ready')
self_plan.set_reached('ncs:init')

route_plan = PlanComponent(service, 'router', 'myserv:router')
route_plan.append_state('ncs:init')
route_plan.append_state('myserv:syslog-initialized')
route_plan.append_state('myserv:ntp-initialized')
route_plan.append_state('myserv:dns-initialized')

```

```
route_plan.append_state('ncs:ready')
route_plan.set_reached('ncs:init')
```

When appending a new state to a plan the initial state is set to ncs:not-reached. At completion of a plan the state is set to ncs:ready. In this case when the service is completely setup:

```
self_plan.set_reached('ncs:ready')
```

Python packages

Action handler

The Python high-level API provides an easy way to implement an action handler for your modeled actions. The easiest way to create a handler is to use the **ncs-make-package** command. It creates some ready to use skeleton code.

```
$ cd packages
$ ncs-make-package --service-skeleton python pyaction --component-class
  action.Action \
  --action-example
```

The generated package skeleton:

```
$ tree pyaction
pyaction/
+-- README
+-- doc/
+-- load-dir/
+-- package-meta-data.xml
+-- python/
|   +-- pyaction/
|   |   +-- __init__.py
|   |   +-- action.py
+-- src/
|   +-- Makefile
|   +-- yang/
|       +-- action.yang
+-- templates/
```

This example action handler takes a number as input, doubles it, and returns the result.

When debugging Python packages refer to [the section called “Debugging of Python packages”](#).

Example 107. Action server implementation

```
# -*- mode: python; python-indent: 4 -*-
from ncs.application import Application
from ncs.dp import Action

# -----
# ACTIONS EXAMPLE
# -----
class DoubleAction(Action):
    @Action.action
    def cb_action(self, uinfo, name, kp, input, output):
        self.log.info('action name: ', name)
        self.log.info('action input.number: ', input.number)

        output.result = input.number * 2
```

```

class LeafListAction(Action):
    @Action.action
    def cb_action(self, uinfo, name, kp, input, output):
        self.log.info('action name: ', name)
        self.log.info('action input.args: ', input.args)
        output.result = [ w.upper() for w in input.args]

# -----
# COMPONENT THREAD THAT WILL BE STARTED BY NCS.
# -----
class Action(Application):
    def setup(self):
        self.log.info('Worker RUNNING')
        self.register_action('action-action', DoubleAction)
        self.register_action('llist-action', LeafListAction)

    def teardown(self):
        self.log.info('Worker FINISHED')

```

Test the action by doing a request from the NSO CLI:

```

admin@ncs> request action double number 21
result 42
[ok][2016-04-22 10:30:39]

```

The input and output parameters are the most commonly used parameters of the action callback method. They provide the access objects to the data provided to the action request and the returning result.

They are `maagic.Node` objects, which provide easy access to the modeled parameters.

Table 108. Action handler callback parameters

Parameter	Type	Description
<code>self</code>	<code>ncs.dp.Action</code>	The action object.
<code>uinfo</code>	<code>ncs.UserInfo</code>	User information of the requester.
<code>name</code>	<code>string</code>	The tail:action name.
<code>kp</code>	<code>ncs.HKeypathRef</code>	The keypath of the action.
<code>input</code>	<code>ncs.maagic.Node</code>	An object containing the parameters of the input section of the action yang model.
<code>output</code>	<code>ncs.maagic.Node</code>	The object where to put the output parameters as defined in the output section of the action yang model.

Service handler

The Python high-level API provides an easy way to implement a service handler for your modeled services. The easiest way to create a handler is to use the `ncs-make-package` command. It creates some skeleton code.

```

$ cd packages
$ ncs-make-package --service-skeleton python pyservice \
--component-class service.Service

```

The generated package skeleton:

```
$ tree pyservice
```

```
pyservice/
+-- README
+-- doc/
+-- load-dir/
+-- package-meta-data.xml
+-- python/
|   +-- pyservice/
|       +-- __init__.py
|       +-- service.py
+-- src/
|   +-- Makefile
|   +-- yang/
|       +-- service.yang
+-- templates/
```

This example has some code added for the service logic, including a service template.

When debugging Python packages refer to the section called “[Debugging of Python packages](#)”.

Example 109. High-level python service implementation

Add some service logic to the cb_create:

```
# -*- mode: python; python-indent: 4 -*-
from ncs.application import Application
from ncs.application import Service
import ncs.template

# -----
# SERVICE CALLBACK EXAMPLE
# -----
class ServiceCallbacks(Service):
    @Service.create
    def cb_create(self, tctx, root, service, proplist):
        self.log.info('Service create(service=', service._path, ')')

        # Add this service logic >>>>
        vars = ncs.template.Variables()
        vars.add('MAGIC', '42')
        vars.add('CE', service.device)
        vars.add('INTERFACE', service.unit)
        template = ncs.template.Template(service)
        template.apply('pyservice-template', vars)

        self.log.info('Template is applied')

        dev = root.devices.device[service.device]
        dev.description = "This device was modified by %s" % service._path
        # <<<<<< service logic

    @Service.pre_lock_create
    def cb_pre_lock_create(self, tctx, root, service, proplist):
        self.log.info('Service plcreate(service=', service._path, ')')

    @Service.pre_modification
    def cb_pre_modification(self, tctx, op, kp, root, proplist):
        self.log.info('Service premod(service=', kp, ')')

    @Service.post_modification
    def cb_post_modification(self, tctx, op, kp, root, proplist):
        self.log.info('Service premod(service=', kp, ')')
```

```

# -----
# COMPONENT THREAD THAT WILL BE STARTED BY NCS.
# -----
class Service(Application):
    def setup(self):
        self.log.info('Worker RUNNING')
        self.register_service('service-servicepoint', ServiceCallbacks)

    def teardown(self):
        self.log.info('Worker FINISHED')

```

Add a template to packages/pyservice/templates/service.template.xml:

```

<config-template xmlns="http://tail-f.com/ns/config/1.0">
    <devices xmlns="http://tail-f.com/ns/ncs">
        <device tags="nocreate">
            <name>{$CE}</name>
            <config tags="merge">
                <interface xmlns="urn:ios">
                    <FastEthernet>
                        <name>0/{$INTERFACE}</name>
                        <description>The maagic: {$MAGIC}</description>
                    </FastEthernet>
                </interface>
            </config>
        </device>
    </devices>
</config-template>

```

Table 110. Service handler callback parameters

Parameter	Type	Description
<i>self</i>	ncs.application.Service	The service object.
<i>tctx</i>	ncs.TransCtxRef	Transaction context.
<i>root</i>	ncs.maagic.Node	An object pointing to the root with the current transaction context, using shared operations (<i>create</i> , <i>set_elem</i> , ...) for configuration modifications.
<i>service</i>	ncs.maagic.Node	An object pointing to the service with the current transaction context, using shared operations (<i>create</i> , <i>set_elem</i> , ...) for configuration modifications.
<i>propList</i>	list(tuple(str, str))	The opaque object for the service configuration used to store hidden state information between invocations. It is updated by returning a modified list.

Low-level APIs

The Python low-level APIs are a direct mapping of the C-APIs. A C call has a corresponding Python function entry. From a programmers point of view it wraps the C data structures into Python objects and handles the related memory management when requested by the Python garbage collector. Any errors are reported as `error.Error`.

The low-level APIs will not be described in detail in this document, but you will find a few examples showing its usage in the coming sections.

See `pydoc3 _ncs` and `man confd_lib.lib` for further information.

Low-level MAAPI API

This API is a direct mapping of the NSO MAAPI C API. See [pydoc3 _ncs.maapi](#) and [man confd_lib_maapi](#) for further information.

This example is a script to read and de-crypt a password using the Python low-level MAAPI API.

Example 111. Setting of configuration data using MAAPI

```
import socket
import _ncs
from _ncs import maapi

sock_maapi = socket.socket()

maapi.connect(sock_maapi,
              ip='127.0.0.1',
              port=_ncs.NCS_PORT)

maapi.load_schemas(sock_maapi)

maapi.start_user_session(
    sock_maapi,
    'admin',
    'python',
    [],
    '127.0.0.1',
    _ncs.PROTO_TCP)

maapi.install_crypto_keys(sock_maapi)

th = maapi.start_trans(sock_maapi, _ncs.RUNNING, _ncs.READ)

path = "/devices/authgroups/group{default}/umap{admin}/remote-password"
encrypted_password = maapi.get_elem(sock_maapi, th, path)

decrypted_password = _ncs.decrypt(str(encrypted_password))

maapi.finish_trans(sock_maapi, th)
maapi.end_user_session(sock_maapi)
sock_maapi.close()

print("Default authgroup admin password = %s" % decrypted_password)
```

This example is a script to do a **check-sync** action request using the low-level MAAPI API.

Example 112. Action request

```
import socket
import _ncs
from _ncs import maapi

sock_maapi = socket.socket()

maapi.connect(sock_maapi,
              ip='127.0.0.1',
              port=_ncs.NCS_PORT)

maapi.load_schemas(sock_maapi)

_ncs.maapi.start_user_session(
```

```

        sock_maapi,
        'admin',
        'python',
        [ ],
        '127.0.0.1',
        _ncs.PROTO_TCP)

ns_hash = _ncs.str2hash("http://tail-f.com/ns/ncs")

results = maapi.request_action(sock_maapi, [], ns_hash, "/devices/check-sync")
for result in results:
    v = result.v
    t = v.conf_type()
    if t == _ncs.C_XMLBEGIN:
        print("sync-result {")
    elif t == _ncs.C_XMLEND:
        print("}")
    elif t == _ncs.C_BUF:
        tag = result.tag
        print("    %s %s" % (_ncs.hash2str(tag), str(v)))
    elif t == _ncs.C_ENUM_HASH:
        tag = result.tag
        text = v.val2str((ns_hash, '/devices/check-sync/sync-result/result'))
        print("    %s %s" % (_ncs.hash2str(tag), text))

maapi.end_user_session(sock_maapi)
sock_maapi.close()

```

Low-level CDB API

This API is a direct mapping of the NSO CDB C API. See [pydoc3 _nes.cdb](#) and [man confd_lib_cdb](#) for further information.

Setting of operational data has historically been done using one of the CDB API:s (Python, Java, C). This example shows how set a value and trigger subscribers for operational data using the Python low-level API. API.

Example 113. Setting of operational data using CDB API

```

import socket
import _ncs
from _ncs import cdb

sock_cdb = socket.socket()

cdb.connect(
    sock_cdb,
    type=cdb.DATA_SOCKET,
    ip='127.0.0.1',
    port=_ncs.NCS_PORT)

cdb.start_session2(sock_cdb, cdb.OPERATIONAL, cdb.LOCK_WAIT | cdb.LOCK_REQUEST)

path = "/operdata/value"
cdb.set_elem(sock_cdb, _ncs.Value(42, _ncs.C_UINT32), path)

new_value = cdb.get(sock_cdb, path)

cdb.end_session(sock_cdb)
sock_cdb.close()

```

```
print("/operdata/value is now %s" % new_value)
```



CHAPTER 12

NSO Packages

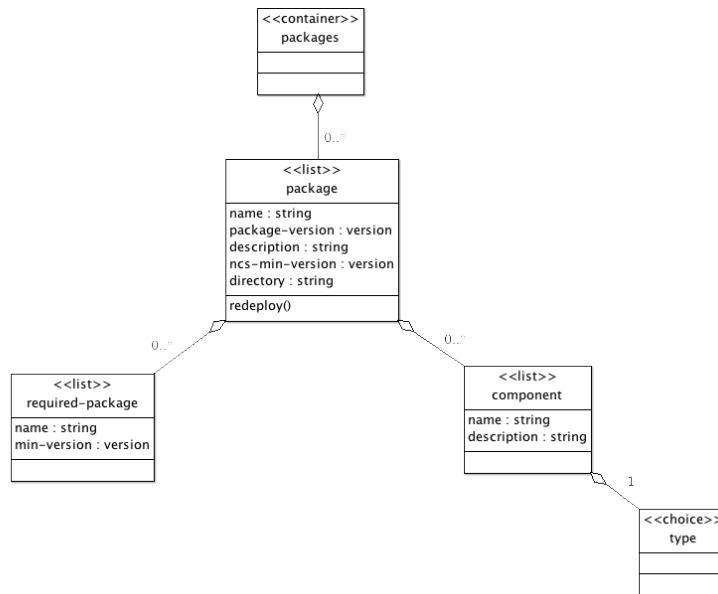
- [Package Overview, page 193](#)
- [An Example Package, page 195](#)
- [The package-meta-data.xml file, page 195](#)
- [Components, page 198](#)
- [Creating Packages , page 201](#)

Package Overview

All user code that needs to run in NSO must be part of a package. A package is basically a directory of files with a fixed file structure. A package consists of code, YANG modules, custom Web UI widgets etc., that are needed in order to add an application or function to NSO. Packages is a controlled way to manage loading and versions of custom applications.

A package is a directory where the package name is the same as the directory name. At the toplevel of this directory a file called `package-meta-data.xml` must exist. The structure of that file is defined by the YANG model `$NCS_DIR/src/ncs/yang/tailf-ncs-packages.yang`. A package may also be a tar archive with the same directory layout. The tar archive can be either uncompressed with suffix `.tar`, or gzip-compressed with suffix `.tar.gz` or `.tgz`. The archive file should also follow some naming conventions. There are two acceptable naming conventions for archive files, one is that after the introduction of CDM in the NSO 5.1, it can be named by `ncs-<ncs-version>-<package-name>-<package-version>.<suffix>`, e.g. `ncs-5.3-my-package-1.0.tar.gz` and the other is `<package-name>-<package-version>.<suffix>`, e.g. `my-package-1.0.tar.gz`.

- `package-name` - should use letters, digits and may include underscores (`_`) or dashes (`-`), but no additional punctuation and digits can not follow underscores or dashes immediately.
- `package-version` - should use numbers and dot (`.`).

Figure 114. Package Model

Packages are composed of components. The following types of components are defined: NED, Application, and Callback.

The file layout of a package is:

```

<package-name>/package-meta-data.xml
    load-dir/
    shared-jar/
    private-jar/
    webui/
    templates/
    src/
    doc/
    netsim/
  
```

The `package-meta-data.xml` defines several important aspects of the package, such as the name, dependencies on other packages, the package's components etc. This will be thoroughly described later in this chapter.

When NSO starts, it needs to search for packages to load. The `ncs.conf` parameter `/ncs-config/load-path` defines a list of directories. At initial startup, NSO searches these directories for packages, copies the packages to a private directory tree in the directory defined by the `/ncs-config/state-dir` parameter in `ncs.conf`, and loads and starts all the packages found. All `.fxs` (compiled YANG files) and `.ccl` (compiled CLI spec files) files found in the directory `load-dir` in a package are loaded. On subsequent startups, NSO will by default only load and start the copied packages - see the section called “[Loading Packages](#)” for different ways to get NSO to search the load path for changed or added packages.

A package usually contains Java code. This Java code is loaded by a class loader in the NSO Java VM. A package that contains Java code must compile the Java code so that the compilation results are divided into jar files where code that is supposed to be shared among multiple packages is compiled into one set of jar files, and code that is private to the package itself is compiled into another set of jar files. The shared

and the common jar files shall go into the `shared-jar` directory and the `private-jar` directory, respectively. By putting for example the code for a specific service in a private jar, NSO can dynamically upgrade the service without affecting any other service.

The optional `webui` directory contains webui customization files.

An Example Package

The NSO example collection for developers contains a number of small self-contained examples. The collection resides at `$NCS_DIR/examples.ncs/getting-started/developing-with-ncs`. Each of these examples defines a package. Let's take a look at some of these packages. The example `3-aggregated-stats` has a package `./packages/stats`. The `package-meta-data.xml` file for that package looks like:

Example 115. An Example Package

```
<ncs-package xmlns="http://tail-f.com/ns/ncs-packages">
    <name>stats</name>
    <package-version>1.0</package-version>
    <description>Aggregating statistics from the network</description>
    <ncs-min-version>3.0</ncs-min-version>
    <required-package>
        <name>router-nc-1.0</name>
    </required-package>
    <component>
        <name>stats</name>
        <callback>
            <java-class-name>com.example.stats.Stats</java-class-name>
        </callback>
    </component>
</ncs-package>
```

The file structure in the package looks like:

```
----package-meta-data.xml
----private-jar
----shared-jar
----src
    |----Makefile
    |----yang
    |    |----aggregate.yang
    |----java
    |    |----build.xml
    |    |----src
    |        |----com
    |            |----example
    |                |----stats
    |                    |----namespaces
    |                    |----Stats.java
----doc
----load-dir
```

The `package-meta-data.xml` file

The `package-meta-data.xml` file defines the name of the package, additional settings, and one *component*. Its settings are defined by the `$NCS_DIR/src/ncs/yang/tailf-ncs-packages.yang` YANG model, where the *package* list name gets renamed to *ncs-package*. See the

The package-meta-data.xml file

`tailf-ncs-packages.yang` module where all options are described in more detail. To get an overview, use the IETF RFC 8340 based YANG tree diagram.

```
$ yanger -f tree tailf-ncs-packages.yang

submodule: tailf-ncs-packages (belongs-to tailf-ncs)
  +-ro packages
    +-ro package* [name] <-- renamed to "ncs-package" in package-meta-data.xml
      +-ro name                      string
      +-ro package-version           version
      +-ro description?             string
      +-ro ncs-min-version*        version
      +-ro python-package!
        | +-ro vm-name?            string
        | +-ro callpoint-model?   enumeration
        +-ro directory?            string
        +-ro templates*            string
        +-ro template-loading-mode? enumeration
        +-ro supported-ned-id*    union
        +-ro required-package* [name]
          | +-ro name              string
          | +-ro min-version?     version
        +-ro component* [name]
          +-ro name                string
          +-ro description?       string
          +-ro entitlement-tag?   string
          +-ro (type)
            +---(ned)
              | +-ro ned
                +-ro (ned-type)
                  | +---(netconf)
                    |   +-ro netconf
                    |     +-ro ned-id?   identityref
                  +---:(snmp)
                    |   +-ro snmp
                    |     +-ro ned-id?   identityref
                  +---:(cli)
                    |   +-ro cli
                    |     +-ro ned-id      identityref
                    |     +-ro java-class-name string
                  +---:(generic)
                    +-ro generic
                      +-ro ned-id      identityref
                      +-ro java-class-name string
            +-ro device
              | +-ro vendor          string
              | +-ro product-family? string
            +-ro option* [name]
              +-ro name            string
              +-ro value?          string
            +---:(upgrade)
              +-ro upgrade
                +-ro (type)
                  +---:(java)
                    |   +-ro java-class-name?   string
                  +---:(python)
                    +-ro python-class-name? string
            +---:(callback)
              +-ro callback
                +-ro java-class-name*   string
            +---:(application)
              +-ro application
```

```

    +-+ro (type)
    |   +-+:(java)
    |   |   +-+ro java-class-name      string
    |   |   +-+:(python)
    |   |       +-+ro python-class-name  string
    +-+ro start-phase?          enumeration

```



Note The order of the XML entries in a package-meta-data.xml must be in the same order as the model shown above.

A sample package configuration taken from the \$NCS_DIR/examples.ncs/development-guide/nano-services/netsim-vrouterexample:

```

$ ncs_load -o -Fp -p /packages

<config xmlns="http://tail-f.com/ns/config/1.0">
  <packages xmlns="http://tail-f.com/ns/ncs">
    <package>
      <name>router-nc-1.1</name>
      <package-version>1.1</package-version>
      <description>Generated netconf package</description>
      <nscs-min-version>5.7</nscs-min-version>
      <directory>./state/packages-in-use/1/router</directory>
      <component>
        <name>router</name>
        <ned>
          <netconf>
            <ned-id xmlns:router-nc-1.1="http://tail-f.com/ns/ned-id/router-nc-1.1">
              router-nc-1.1:router-nc-1.1</ned-id>
            </netconf>
            <device>
              <vendor>Acme</vendor>
            </device>
          </ned>
        </component>
        <oper-status>
          <up/>
        </oper-status>
      </package>
      <package>
        <name>vrouter</name>
        <package-version>1.0</package-version>
        <description>Nano services netsim virtual router example</description>
        <nscs-min-version>5.7</nscs-min-version>
        <python-package>
          <vm-name>vrouter</vm-name>
          <callpoint-model>threading</callpoint-model>
        </python-package>
        <directory>./state/packages-in-use/1/vrouter</directory>
        <templates>vrouter-configured</templates>
        <template-loading-mode>strict</template-loading-mode>
        <supported-ned-id xmlns:router-nc-1.1="http://tail-f.com/ns/ned-id/router-nc-1.1">
          router-nc-1.1:router-nc-1.1</supported-ned-id>
        <required-package>
          <name>router-nc-1.1</name>
          <min-version>1.1</min-version>
        </required-package>
        <component>
          <name>nano-app</name>
          <description>Nano service callback and post-actions example</description>

```

```

<application>
  <python-class-name>vrouter.nano_app.NanoApp</python-class-name>
  <start-phase>phase2</start-phase>
</application>
</component>
<oper-status>
  <up/>
</oper-status>
</package>
</packages>
</config>

```

Below is a brief list of the configurables in the `tailf-ncs-packages.yang` YANG model that applies to meta data file. A more detailed description can be found in the YANG model:

- name - the name of the package. All packages in the system must have unique names.
- package-version - the version of the package. This is for administrative purposes only, NSO cannot simultaneously handle two versions of the same package.
- ncs-min-version - the oldest known NSO version where the package works.
- python-package - Python specific package data.
 - vm-name - the Python VM name for the package. Default is the package vm-name. Packages with the same vm-name run in the same Python VM. Applicable only when callpoint-model = threading.
 - callpoint-model - A Python package run Services, Nano Services, and Actions in the same OS process. If the callpoint-model is set to multiprocessing each will get a separate worker process. Running Services, Nano Services, and Actions in parallel can, depending on the application, improve the performance at the cost of complexity. See [the section called “The application component”](#) for details.
- directory - the path to the directory of the package.
- templates - the templates defined by the package.
- template-loading-mode - control if the templates are interpreted in strict or relaxed mode.
- supported-ned-id - the list of ned-ids supported by this package. An example of the expected format taken from the `$NCS_DIR/examples.ncs/development-guide/nano-services/netsim-vrouter` example:


```
<supported-ned-id xmlns:router-nc-1.1="http://tail-f.com/ns/ned-id/router-nc-1.1">
  router-nc-1.1:router-nc-1.1</supported-ned-id>
```
- required-package - a list of names of other packages that are required for this package to work.
- component - Each package defines zero or more components.

Components

Each component in a package has a name. The names of all the components must be unique within the package. The YANG model for packages contain:

```

.....
list component {
  key name;
  leaf name {
    type string;
  }
  ...
choice type {
  mandatory true;
  case ned {

```

```

    ...
}
case callback {
    ...
}
case application {
    ...
}
case upgrade {
    ...
}
...
}
...

```

Lots of additional information can be found in the YANG module itself. The mandatory choice that defines a component must be one of *ned*, *callback*, *application* or *upgrade*. We have:

Component types

ned

A Network Element Driver component is used southbound of NSO to communicate with managed devices (described in Chapter 2, *Network Element Drivers (NEDs)* in *NSO 5.7 NED Development*). The easiest NED to understand is the NETCONF NED which is built in into NSO.

There are 4 different types of NEDs:

- *netconf* - used for NETCONF enabled devices such as Juniper routers, ConfD powered devices or any device that speaks proper NETCONF and also has YANG models. Plenty of packages in the NSO example collection have NETCONF NED components, for example `$NCS_DIR/examples.ncs/getting-started/developing-with-ncs/0-router-network/packages/router`.
- *snmp* - used for SNMP devices.

The example `$NCS_DIR/examples.ncs/snmp-ned/basic` has a package which has an SNMP NED component.

- *cli* - used for CLI devices. The package `$NCS_DIR/packages/neds/cisco-ios` is an example of a package that has a CLI NED component.
- *generic* - used for generic NED devices. The example `$NCS_DIR/examples.ncs/generic-ned/xmlrpc-device` has a package called `xml-rpc` which defines a NED component of type *generic*

A CLI NED and a generic NED component must also come with additional user written Java code, whereas a NETCONF NED and an SNMP NED have no Java code.

callback

This defines component with one or many java classes that implements callbacks using the Java callback annotations.

If we look at the component in the `stats` package above we have:

```

<component>
    <name>stats</name>
    <callback>
        <java-class-name>
            com.example.stats.Stats
        </java-class-name>
    </callback>
</component>

```

The *Stats* class here implements a read-only data provider. See [the section called “DP API”](#).

The *callback* type of component is used for a wide range of callback type Java applications, where one of the most important are the Service Callbacks. The following list of Java callback annotations apply to callback components.

- *ServiceCallback* - to implement service to device mappings. See example: \$NCS_DIR/examples.ncs/getting-started/developing-with-ncs/4-rfs-service See [Chapter 14, Developing NSO Services](#) for a thorough introduction to services.
- *ActionCallback* - to implement user defined tailf:actions or YANG RPCs. See example: \$NCS_DIR/examples.ncs/getting-started/developing-with-ncs/2-actions
- *DataCallback* - to implement the data getters and setters for a data provider. See example \$NCS_DIR/examples.ncs/getting-started/developing-with-ncs/3-aggregated-stats
- *TransCallback* to implement the transaction portions of a data provider callback. See example \$NCS_DIR/examples.ncs/getting-started/developing-with-ncs/3-aggregated-stats
- *DBCallback* - to implement an external database. See example: \$NCS_DIR/examples.ncs/getting-started/developing-with-ncs/6-extern-db
- *SnmpInformResponseCallback* to implement an SNMP listener - See example \$NCS_DIR/examples.ncs/snmp-notification-receiver
- *TransValidateCallback*, *ValidateCallback* - to implement a user defined validation hook that gets invoked on every commit.
- *AuthCallback* - to implement a user hook that gets called whenever a user is authenticated by the system.
- *AuthorizationCallback* - to implement a authorization hook that allow/disallow users to do operations and/or access data. Note, this callback should normally be avoided since, by nature, invoking a callback for any operation and/or data element is an performance impairment.

A package that has a *callback* component usually has some YANG code and then also some Java code that relates to that YANG code. By convention the YANG and the Java code resides in a src directory in the component. When the source of the package is built, any resulting fxs files (compiled YANG files) must reside in the *load-dir* of the package and any resulting Java compilation results must reside in the *shared-jar* and *private-jar* directories. Study the *3-aggregated-stats* example to see how this is achieved.

application	Used to cover Java applications that do not fit into the <i>callback</i> type. Typically this is functionality that should be running in separate threads and work autonomously. The example \$NCS_DIR/examples.ncs/getting-started/developing-with-ncs/1-cdb contains three components that are of type <i>application</i> . These components must also contain a <i>java-class-name</i> element. For application components, that Java class must implement the <i>ApplicationComponent</i> Java interface.
upgrade	Used to migrate data for packages where the yang model has changed and the automatic cdb upgrade is not sufficient. The upgrade component consists of a java class with a main method that is expected to run one time only.

The example `$NCS_DIR/examples.ncs/getting-started/developing-with-ncs/14-upgrade-service` illustrates user cdb upgrades using *upgrade* components.

Creating Packages

NSO ships with a tool *ncs-make-package* that can be used to create packages. [Chapter 13, Package Development](#) discusses in depth how to develop a package.

Creating a NETCONF NED Package

This use case applies if we have a set of YANG files that define a managed device. If we wish to develop an EMS solution for an existing device *and* that device has YANG files and also speaks NETCONF, we need to create a package for that device in order to be able to manage it. Assuming all YANG files for the device are stored in `./acme-router-yang-files`, we can create a package for the router as:

```
$ ncs-make-package --netconf-ned ./acme-router-yang-files acme
$ cd acme/src; make
```

The above command will create a package called *acme* in `./acme`. The *acme* package can be used for two things; managing real acme routers and also be used as input to the *ncs-netsim* tool to simulate a network of *acme* routers.

In the first case, managing real acme routers, all we really need to do is to put the newly generated package in the load-path of NSO, start NSO with package reload (see [the section called “Loading Packages”](#)), and then add one or more acme routers as managed devices to NSO. The *ncs-setup* tool can be used to do this:

```
$ ncs-setup --ned-package ./acme --dest ./ncs-project
```

The above command generates a directory `./ncs-project` which is suitable for running NSO. Assume we have an existing router at IP address 10.2.3.4 and that we can log into that router over the NETCONF interface using user name bob, and password secret. The following session shows how to setup NSO to manage this router:

```
$ cd ./ncs-project
$ ncs
$ ncs_cli -u admin
> configure
> set devices authgroups group southbound-bob umap admin \
   remote-name bob remote-password secret
> set devices device acmeli authgroup southbound-bob address 10.2.3.4
> set devices device acmeli device-type netconf
> commit
```

We can also use the newly generated acme package to simulate a network of acme routers. During development this is especially useful. The *ncs-netsim* tool can create a simulated network of acme routers as:

```
$ ncs-netsim create-network ./acme 5 a --dir ./netsim
$ ncs-netsim start
DEVICE a0 OK STARTED
DEVICE a1 OK STARTED
DEVICE a2 OK STARTED
DEVICE a3 OK STARTED
DEVICE a4 OK STARTED
$
```

And finally, *ncs-setup* can be used to initialize an environment where NSO is used to manage all devices in an *ncs-netsim* network:

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

Creating an SNMP NED Package

Similarly, if we have a device that has a set of MIB files, we can use *ncs-make-package* to generate a package for that device. An SNMP NED package can, similarly to a NETCONF NED package, be used to both manage real devices and also be fed to *ncs-netsim* to generate a simulated network of SNMP devices.

Assuming we have a set of MIB files in `./mibs`, we can generate a package for a device with those mibs as:

```
$ ncs-make-package --snmp-ned ./mibs acme
$ cd acme/src; make
```

Creating a CLI NED Package or a Generic NED Package

For CLI NEDs and Generic NEDs, we cannot (yet) generate the package. Probably the best option for such packages is to start with one of the examples. A good starting point for a CLI NED is `$NCS_DIR/packages/neds/cisco-ios` and a good starting point for a Generic NED is the example `$NCS_DIR/examples.ncs/generic-ned/xmlrpc-device`

Creating a Service Package or a Data Provider Package

The *ncs-make-package* can be used to generate empty skeleton packages for a data provider and a simple service. The flags `--service-skeleton` and `--data-provider-skeleton`

Alternatively one of the examples can be modified to provide a good starting point. For example `$NCS_DIR/examples.ncs/getting-started/developing-with-ncs/4-rfs-service`



CHAPTER 13

Package Development

- [Developing a Service Package, page 203](#)
- [ncs-setup, page 205](#)
- [The netsim Part of a NED Package, page 206](#)
- [Plug-and-play scripting, page 208](#)
- [Creating a Service Package, page 208](#)
- [Java Service Implementation, page 209](#)
- [Developing our First Service Application, page 209](#)
- [Tracing Within the NSO Service Manager, page 212](#)
- [Controlling error messages info level from Java, page 215](#)
- [Loading Packages, page 216](#)
- [Debugging the Service and Using Eclipse IDE, page 216](#)
- [Working with ncs-project, page 220](#)

Developing a Service Package

When setting up an application project, there are a number of things to think about. A service package needs a service model, NSO configuration files and mapping code. Similarly, NED packages need YANG files and NED code. We can either copy an existing example and modify that, or we can use the tool `ncs-make-package` to create an empty skeleton for a package for us. The `ncs-make-package` tool provides a good starting point for a development project. Depending on the type of package, we use `ncs-make-package` to set up a working development structure.

As explained in [Chapter 12, NSO Packages](#), NSO runs all user Java code and also loads all data models through an NSO package. Thus a development project is the same as developing a package. Testing and running the package is done by putting the package in the NSO load-path and running NSO.

There are different kinds of packages; NED packages, service packages etc. Regardless of package type, the structure of the package as well as the deployment of the package into NSO is the same. The script `ncs-make-package` creates the following for us:

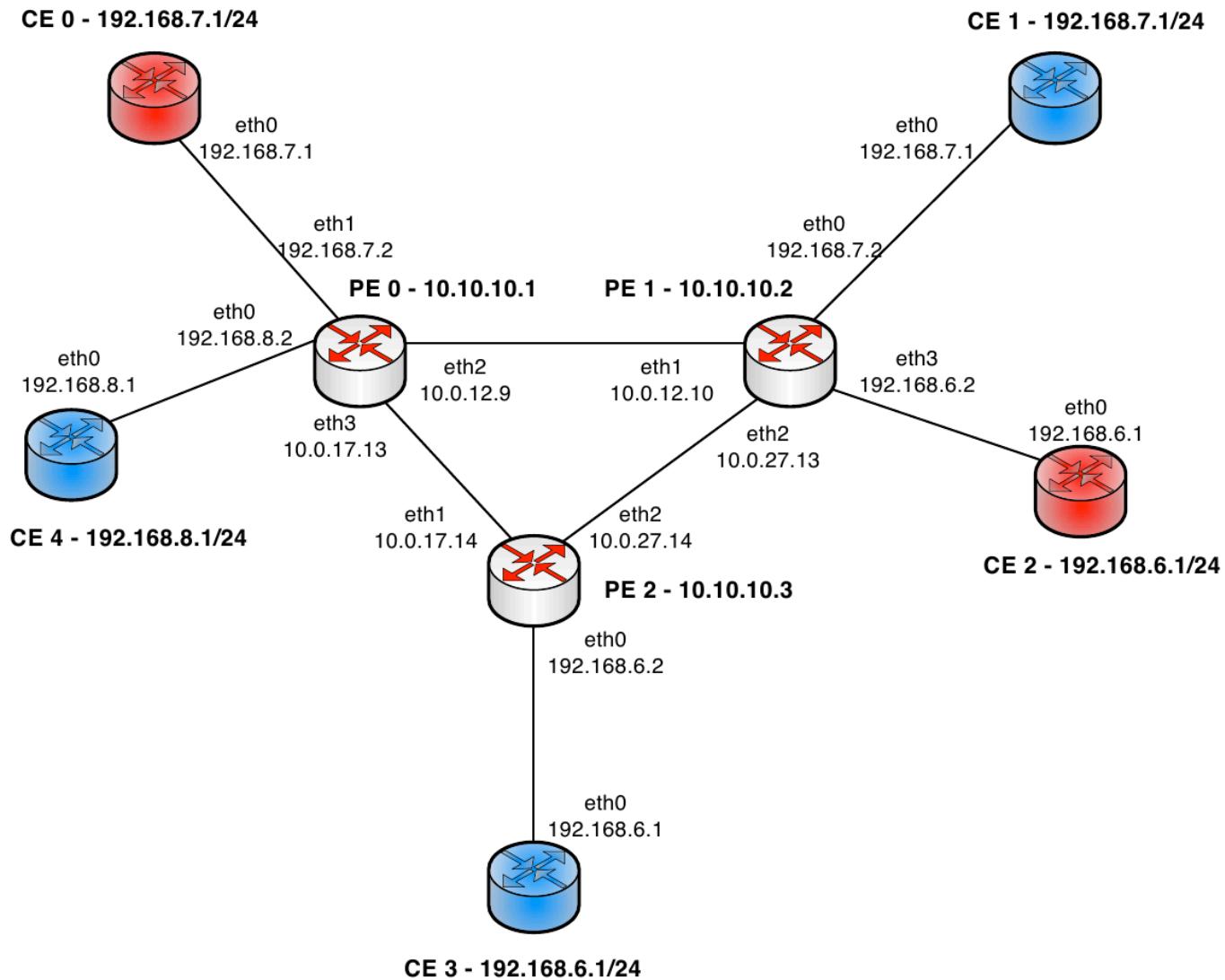
- A Makefile to build the source code of the package. The package contains source code and needs to be built.

- If it's a NED package, a `netsim` directory which is used by the `ncs-netsim` tool to simulate a network of devices.
- If it is a service package, skeleton YANG and Java files that can be modified are generated.

In this chapter we are going to develop an MPLS service for a network of provider edge routers (PE) and customer equipment routers (CE). The assumption is that the routers speak NETCONF and that we have proper YANG modules for the two types of routers. The techniques described here work equally well for devices that speak other protocols than NETCONF, such as Cisco CLI or SNMP.

The first thing we want to do is to create a simulation environment where ConfD is used as NETCONF server to simulate the routers in our network. We plan to create a network that looks like:

Figure 116. MPLS network



In order to create the simulation network, the first thing we need to do is to create NSO packages for the two router models. The packages is also exactly what NSO needs in order to manage the routers.

Assume that the yang files for the PE routers reside in `./pe-yang-files` and the YANG files for the CE routers reside in `./ce-yang-files`. The `ncs-make-package` tool is used to create two device packages, one called `pe` and the other `ce`.

```
$ ncs-make-package --netconf-ned ./pe-yang-files pe
$ ncs-make-package --netconf-ned ./ce-yang-files ce
$ (cd pe/src; make)
$ (cd ce/src; make)
```

At this point, we can use the `ncs-netsim` tool to create a simulation network. `ncs-netsim` will use the Tail-f ConfD daemon as a NETCONF server to simulate the managed devices, all running on localhost.

```
$ ncs-netsim create-network ./ce 5 ce create-network ./pe 3 pe
```

The above command creates a network with 8 routers, 5 running the YANG models for a CE router and 3 running a YANG model for the PE routers. `ncs-netsim` can be used to stop, start and manipulate this network. For example:

```
$ ncs-netsim start
DEVICE ce0 OK STARTED
DEVICE ce1 OK STARTED
DEVICE ce2 OK STARTED
DEVICE ce3 OK STARTED
DEVICE ce4 OK STARTED
DEVICE pe0 OK STARTED
DEVICE pe1 OK STARTED
DEVICE pe2 OK STARTED
```

ncs-setup

In the previous section, we described how to use `ncs-make-package` and `ncs-netsim` to setup a simulation network. Now, we want to use `ncs` to control and manage precisely the simulated network. We can use the `ncs-setup` tool setup a directory suitable for this. `ncs-setup` has a flag to setup NSO initialization files so that all devices in a `ncs-netsim` network are added as managed devices to NSO. If we do:

```
$ ncs-setup --netsim-dir ./netsim --dest NCS;
$ cd NCS
$ cat README.ncs
.....
$ ncs
```

The above commands, db, log etc directories and also creates an NSO XML initialization file in `./NCS/ncs-cdb/netsim_devices_init.xml`. The init file is important, it is created from the content of the netsim directory and it contains the IP address, port, auth credentials and NED type for all the devices in the netsim environment. There is a dependency order between `ncs-setup` and `ncs-netsim` since `ncs-setup` creates the XML init file based on the contents in the netsim environment, therefor we must run the `ncs-netsim create-network` command before we execute the `ncs-setup` command. Once `ncs-setup` has been run, and the init XML file has been generated it is possible to manually edit that file.

If we start the NSO CLI, we have for example :

```
$ ncs_cli -u admin
admin connected from 127.0.0.1 using console on zoe
admin@zoe> show configuration devices device ce0
address    127.0.0.1;
port       12022;
authgroup default;
device-type {
```

```

        netconf;
    }
    state {
        admin-state unlocked;
    }
}

```

The netsim Part of a NED Package

If we take a look at the directory structure of the generated NETCONF NED packages, we have in `./ce`

```

|----package-meta-data.xml
|----private-jar
|----shared-jar
|----netsim
|----|----start.sh
|----|----confd.conf.netsim
|----|----Makefile
|----src
|----|----ncsc-out
|----|----Makefile
|----|----yang
|----|----|----interfaces.yang
|----|----java
|----|----|----build.xml
|----|----|----src
|----|----|----|----com
|----|----|----|----example
|----|----|----|----|----ce
|----|----|----|----|----|----namespaces
|----doc
|----load-dir

```

It is a NED package, and it has a directory called `netsim` at the top. This indicates to the `ncs-netsim` tool that `ncs-netsim` can create simulation networks that contains devices running the YANG models from this package. This section describes the `netsim` directory and how to modify it. `ncs-netsim` uses ConfD to simulate network elements, and in order to fully understand how to modify a generated `netsim` directory, some knowledge of how ConfD operates may be required.

The `netsim` directory contains three files:

- `confd.conf.netsim` is a configuration file for the ConfD instances. The file will be `/bin/sed` substituted where the following list of variables will be substituted for the actual value for that ConfD instance:
 - 1 %IPC_PORT% for `/confdConfig/confdIpcAddress/port`
 - 2 %NETCONF_SSH_PORT% - for `/confdConfig/netconf/transport/ssh/port`
 - 3 %NETCONF_TCP_PORT% - for `/confdConfig/netconf/transport/tcp/port`
 - 4 %CLI_SSH_PORT% - for `/confdConfig/cli/ssh/port`
 - 5 %SNMP_PORT% - for `/confdConfig/snmpAgent/port`
 - 6 %NAME% - for the name of ConfD instance.
 - 7 %COUNTER% - for the number of the ConfD instance
- The `Makefile` should compile the YANG files so that ConfD can run them. The `Makefile` should also have an `install` target that installs all files required for ConfD to run one instance of a simulated network element. This is typically all `fxs` files.
- An optional `start.sh` file where additional programs can be started. A good example of a package where the `netsim` component contains some additional C programs is the `webserver` package in the NSO website example `$NCS_DIR/web-server-farm`.

Remember the picture of the network we wish to work with, there the routers, PE and CE, have IP address and some additional data. So far here, we have generated a simulated network with YANG models. The routers in our simulated network have no data in them, we can log in to one of the routers to verify that:

```
$ ncs-netsim cli pe0
admin connected from 127.0.0.1 using console on zoe
admin@zoe> show configuration interface
No entries found.
[ok][2012-08-21 16:52:19]
admin@zoe> exit
```

The ConfD devices in our simulated network all have a Juniper CLI engine, thus we can, using the command `ncs-netsim cli [devicename]` login to an individual router.

In order to achieve this, we need to have some additional XML initializing files for the ConfD instances. It is the responsibility of the `install` target in the netsim Makefile to ensure that each ConfD instance gets initialized with the proper init data. In the NSO example collection, the example `$NCS_DIR/examples.ncs/mpls` contains precisely the two above mentioned PE and CE packages, but modified so that the network elements in the simulated network gets initialized properly.

If we run that example in the NSO example collection we see

```
$ cd $NCS_DIR/examples.ncs/mpls/mpls-devices
$ make all
....
$ ncs-netsim start
.....
$ ncs
$ ncs_cli -u admin

admin connected from 127.0.0.1 using console on zoe
admin@zoe> show status packages package pe
package-version 1.0;
description      "Generated netconf package";
ncs-min-version 2.0;
component pe {
    ned {
        netconf;
        device {
            vendor "Example Inc.";
        }
    }
}
oper-status {
    up;
}
[ok][2012-08-22 14:45:30]
admin@zoe> request devices sync-from
sync-result {
    device ce0
    result true
}
sync-result {
    device cel
    result true
}
sync-result {
    .....
}
admin@zoe> show configuration devices device pe0 config if:interface
interface eth2 {
    ip    10.0.12.9;
```

```

        mask 255.255.255.252;
    }
    interface eth3 {
        ip   10.0.17.13;
        mask 255.255.255.252;
    }
    interface lo {
        ip   10.10.10.1;
        mask 255.255.0.0;
    }
}

```

A full simulated router network loaded into NSO, with ConfD simulating the 7 routers.

Plug-and-play scripting

With the scripting mechanism it is possible for an end-user to add new functionality to NSO in a plug-and-play like manner. See Chapter 7, *Plug-and-play scripting in NSO 5.7 User Guide* about the scripting concept in general. It is also possible for a developer of an NSO package to enclose scripts in the package.

Scripts defined in an NSO package works pretty much as system level scripts configured with the */ncs-config/scripts/dir* configuration parameter. The difference is that the location of the scripts is predefined. The scripts directory must be named `scripts` and must be located in the top directory of the package.

In this complete example `examples.ncs/getting-started/developing-with-ncs/11-scripting` there is a `README` file and a simple post-commit script `packages/scripting/scripts/post-commit/show_diff.sh` as well as a simple command script `packages/scripting/scripts/command/echo.sh`.

Creating a Service Package

So far we have only talked about packages that describe a managed device, i.e *ned* packages. There are also *callback*, *application* and *service* packages. A service package is a package with some YANG code that models an NSO service together with java code that implements the service. See [Chapter 14, Developing NSO Services](#).

We can generate a service package skeleton, using `ncs-make-package`, as:

```
$ ncs-make-package --service-skeleton java myrfs
$ cd test/src; make
```

make sure that package is part of the load-path, and we can then create *test* service instances - that do nothing.

```
admin@zoe> show status packages package myrfs
package-version 1.0;
description      "Skeleton for a resource facing service - RFS";
ncs-min-version 2.0;
component RFSSkeleton {
    callback {
        java-class-name [ com.example.myrfs.myrfs ];
    }
}
oper-status {
    up;
}
[ok][2012-08-22 15:30:13]
admin@zoe> configure
Entering configuration mode private
```

```
[ok][2012-08-22 15:32:46]

[edit]
admin@zoe% set services myrfs s1 dummy 3.4.5.6
[ok][2012-08-22 15:32:56]
```

`ncs-make-package` will generate skeleton files for our service models and for our service logic. The package is fully build-able and runnable even though the service models are empty. Both CLI and Webui can be run. In addition to this we also have a simulated environment with ConfD devices configured with YANG modules.

Calling `ncs-make-package` with the arguments above will create a service skeleton that is placed in the root in the generated service model. However services can be augmented anywhere or can be located in any YANG module. This can be controlled by giving the argument `--augment NAME` where NAME is the path to where the service should be augmented, or in the case of putting the service as a root container in the service YANG this can be controlled by giving the argument `--root-container NAME`.

Services created using `ncs-make-package` will be of type `list`. However it is possible to have services that are of type `container` instead. A container service need to be specified as a *presence* container.

Java Service Implementation

The service implementation logic of a service can be expressed using the Java language. For each such service a Java class is created. This class should implement the `create()` callback method from the `ServiceCallback` interface. This method will be called to implement the service to device mapping logic for the service instance.

We declare in the component for the package, that we have a *callback component*. In the `package-meta-data.xml` for the generated package, we have:

```
<component>
  <name>RFSSkeleton</name>
  <callback>
    <java-class-name>com.example.myrfs.myrfs</java-class-name>
  </callback>
</component>
```

When the package is loaded, the NSO Java VM will load the jar files for the package, and register the defined class as a callback class. When the user creates a service of this type, the `create()` method will be called.

Developing our First Service Application

In the following sections we are going to show how to write a service applications through a number of examples. The purpose of these examples are to illustrate the concepts described in previous chapters.

- Service Model - a model of the service you want to provide.
- Service Validation Logic - a set of validation rules incorporated into your model.
- Service Logic - a Java class mapping the service model operations onto the device layer.

If we take a look at the Java code in the service generated by `ncs-make-package`, first we have the `create()` which takes four parameters. The `ServiceContext` instance is a container for the current service transaction, with this e.g. the transaction timeout can be controlled. The container service is a `NavuContainer` holding a read/write reference to path in the instance tree containing the current service instance. From this point you can start accessing all nodes contained within created service. The `root`

container is a `NavuContainer` holding a reference to the NSO root. From here you can access the whole data model of the NSO. The `opaque` parameter contains a `java.util.Properties` object instance. This object may be used to transfer additional information between consecutive calls to the create callback. It is always null in the first callback method when a service is first created. This `Properties` object can be updated (or created if null) but should always be returned.

Example 117. Resource Facing Service Implementation

```

@ServiceCallback(servicePoint="myrfsspt",
    callType=ServiceCBType.CREATE)
public Properties create(ServiceContext context,
                        NavuNode service,
                        NavuNode root,
                        Properties opaque)
                        throws DpCallbackException {
    String servicePath = null;
    try {
        servicePath = service.getKeyPath();

        //Now get the single leaf we have in the service instance
        // NavuLeaf sServerLeaf = service.leaf("dummy");

        //...and its value (which is a ipv4-address )
        // ConfIPv4 ip = (ConfIPv4)sServerLeaf.value();

        //Get the list of all managed devices.
        NavuList managedDevices = root.container("devices").list("device");

        // iterate through all manage devices
        for(NavuContainer deviceContainer : managedDevices.elements()){

            // here we have the opportunity to do something with the
            // ConfIPv4 ip value from the service instance,
            // assume the device model has a path /xyz/ip, we could
            // deviceContainer.container("config").
            //         .container("xyz").leaf(ip).set(ip);
            //
            // remember to use NAVU sharedCreate() instead of
            // NAVU create() when creating structures that may be
            // shared between multiple service instances
        }
    } catch (NavuException e) {
        throw new DpCallbackException("Cannot create service " +
                                      servicePath, e);
    }
    return opaque;
}

```

The `opaque` object is extremely useful to pass information between different invocations of the `create()` method. The returned `Properties` object instance is stored persistently. If the `create` method computes something on its first invocation, it can return that computation in order to have it passed in as a parameter on the second invocation.

This is crucial to understand, the Mapping Logic *fastmap mode* relies on the fact that a *modification* of an existing service instance can be realized as a full deletion of what the service instance created when the service instance was first created, followed by yet another `create`, this time with slightly different parameters. The NSO transaction engine will then compute the minimal difference and send southbound to all involved managed devices. Thus a good service instance `create()` method will - when being modified - recreate exactly the same structures it created the first time.

The best way to debug this and to ensure that a modification of a service instance really only sends the minimal NETCONF diff to the south bound managed devices, is to turn on NETCONF trace in the NSO, modify a service instance and inspect the XML sent to the managed devices. A badly behaving `create()` method will incur large reconfigurations of the managed devices, possibly leading to traffic interruptions.

It is highly recommended to also implement a `selftest()` action in conjunction to a service. The purpose of the `selftest()` action is to trigger a test of the service. The `ncs-make-package` tool creates an `selftest()` action that takes no input parameters and have two output parameters.

Example 118. Selftest yang definition

```
tailf:action self-test {
    tailf:info "Perform self-test of the service";
    tailf:actionpoint myrfsselftest;
    output {
        leaf success {
            type boolean;
        }
        leaf message {
            type string;
            description
                "Free format message.";
        }
    }
}
```

The `selftest()` implementation is expected to do some diagnosis of the service. This can possibly include use of testing equipment or probes.

Example 119. Selftest action

```
/**
 * Init method for selftest action
 */
@ActionCallback(callPoint="myrfsselftest", callType=ActionCBType.INIT)
public void init(DpActionTrans trans) throws DpCallbackException {
}

/**
 * Selftest action implementation for service
 */
@ActionCallback(callPoint="myrfsselftest", callType=ActionCBType.ACTION)
public ConfXMLParam[] selftest(DpActionTrans trans, ConfTag name,
                               ConfObject[] kp, ConfXMLParam[] params)
throws DpCallbackException {
    try {
        // Refer to the service yang model prefix
        String nsPrefix = "myrfs";
        // Get the service instance key
        String str = ((ConfKey)kp[0]).toString();

        return new ConfXMLParam[] {
            new ConfXMLParamValue(nsPrefix, "success", new ConfBool(true)),
            new ConfXMLParamValue(nsPrefix, "message", new ConfBuf(str))};
    } catch (Exception e) {
        throw new DpCallbackException("selftest failed", e);
    }
}
```

Tracing Within the NSO Service Manager

The NSO Java VM logging functionality is provided using LOG4J. The logging is composed of a configuration file (`log4j2.xml`) where static settings are made i.e all settings that could be done for LOG4J (see <https://logging.apache.org/log4j/2.x> for more comprehensive log settings). There are also dynamically configurable log settings under `/java-vm/java-logging`.

When we start the NSO Java VM in `main()` the `log4j2.xml` log file is parsed by the LOG4J framework and it applies the static settings to the NSO Java VM environment. The file is searched for in the Java CLASSPATH.

NSO Java VM starts a number of internal processes or threads, one of these thread executes a service called NcsLogger which handles the dynamic configurations of the logging framework. When NcsLogger starts it initially reads all the configurations from `/java-vm/java-logging` and applies them, thus overwriting settings that was previously parsed by the LOG4J framework.

After it has applied the changes from the configuration it starts to listen to changes that are made under `/java-vm/java-logging`.

The LOG4J framework has 8 verbosity levels: ALL, DEBUG, ERROR, FATAL, INFO, OFF, TRACE and WARN. They have the following relations: ALL > TRACE > DEBUG > INFO > WARN > ERROR > FATAL > OFF. This means that the highest verbosity that we could have is level ALL and the lowest no traces at all, OFF. There are corresponding enumerations for each LOG4J verbosity level in `tailf-ncs.yang`, thus the NcsLogger does the mapping between the enumeration type: `log-level-type` and the LOG4J verbosity levels.

Example 120. tailf-ncs-java-vm.yang

```
typedef log-level-type {
    type enumeration {
        enum level-all {
            value 1;
        }
        enum level-debug {
            value 2;
        }
        enum level-error {
            value 3;
        }
        enum level-fatal {
            value 4;
        }
        enum level-info {
            value 5;
        }
        enum level-off {
            value 6;
        }
        enum level-trace {
            value 7;
        }
        enum level-warn {
            value 8;
        }
    }
    description
        "Levels of logging for Java packages in log4j.";
}
```

```

.....
container java-vm {
    ....
    container java-logging {
        tailf:info "Configure Java Logging";
        list logger {
            tailf:info "List of loggers";
            key "logger-name";
            description
                "Each entry in this list holds one representation of a logger with
                a specific level defined by log-level-type. The logger-name
                is the name of a Java package. logger-name can thus be for
                example com.tailf.maapi, or com.tailf etc.";

            leaf logger-name {
                tailf:info "The name of the Java package";
                type string;
                mandatory true;
                description
                    "The name of the Java package for which this logger
                    entry applies.";
            }
            leaf level {
                tailf:info "Log-level for this logger";
                type log-level-type;
                mandatory true;
                description
                    "Corresponding log-level for a specific logger.";
            }
        }
    }
}

```

To change a verbosity level one needs to create a logger. A logger is something that controls the logging of a certain parts of the NSO Java API.

The loggers in the system are hierarchically structured which means that there is one root logger that always exists. All descendants of the root logger inherits its settings from the root logger if the descendant logger don't overwrite its settings explicitly.

The LOG4J loggers are mapped to the package level in NSO Java API so the root logger that exists have a direct descendant that is the package: com and it has in turn a descendant com.tailf.

The com.tailf logger has direct descendant that corresponds to every package in the system for example: com.tailf.cdb, com.tailf.maapi etc.

As in the default case one could configure a logger in the static settings that is in a log4j2.properties file this would mean that we need to explicitly restart the NSO Java VM ,or one could alternatively configure a logger dynamically if an NSO restart is not desired.

Recall that if a logger is not configured explicitly then it will inherit its settings from its predecessors. To overwrite a logger setting we create a logger in NSO.

To create a logger, for example let say that one uses Maapi API to read and write configuration changes in NSO. We want to show all traces including INFO level traces. To enable INFO traces for Maapi classes (located in package com.tailf.maapi) during runtime we start for example a CLI session and create a logger called com.tailf.maapi.

```
ncs@admin% set java-vm java-logging logger com.tailf.maapi level level-info
[ok][2010-11-05 15:11:47]
```

```
ncs@admin% commit
Commit complete.
```

When we commit our changes to CDB the NcsLogger will notice that a change has been made under /java-vm/java-logging, it will then apply the logging settings to the logger com.tailf.maapi that we just created. We explicitly set the INFO level to that logger. All the descendents from com.tailf.maapi will automatically inherit its settings from that logger.

So where do the traces go? The default configuration (in log4j2.properties):
appender.dest1.type=Console the LOG4J framework forward all traces to stdout/stderr.

In NSO all stdout/stderr goes first through the service manager. The service manager has configuration under /java-vm/stdout-capture that controls where the stdout/stderr will end up.

The default setting is in a file called ./ncs-java-vm.log.

Example 121. stdout capture

```
container stdout-capture {
    tailf:info "Capture stdout and stderr";
    description
        "Capture stdout and stderr from the Java VM.

        Only applicable if auto-start is 'true'.";
    leaf enabled {
        tailf:info "Enable stdout and stderr capture";
        type boolean;
        default true;
    }
    leaf file {
        tailf:info "Write Java VM output to file";
        type string;
        default "./ncs-java-vm.log";
        description
            "Write Java VM output to filename.";
    }
    leaf stdout {
        tailf:info "Write output to stdout";
        type empty;
        description
            "If present write output to stdout, useful together
             with the --foreground flag to ncs.";
    }
}
```

It is important to consider that when creating a logger (in this case com.tailf.maapi) the name of the logger has to be an existing package known by NSO classloader.

One could also create a logger named com.tailf with some desired level. This would set all packages (com.tailf.*) to the same level. A common usage is to set com.tailf to level INFO which would set all traces, including INFO from all packages to level INFO.

If one would like to turn off all available traces in the system (quiet mode) then configure com.tailf or (com) to level OFF.

There are INFO level messages in all parts of the NSO Java API. ERROR levels when exception occurs and some warning messages (level WARN) for some places in packages.

There is also protocol traces between the Java API and NSO which could be enabled if we create a logger com.tailf.conf with DEBUG trace level.

Controlling error messages info level from Java

When processing in the java-vm fails the exception error message is reported back to Ncs. This can be more or less informative depending on how elaborate the message is in the thrown exception. Also the exception can be wrapped one or several times with the original exception indicated as root cause of the wrapped exception.

In debugging and error reporting these root cause messages can be valuable to understand what actually happens in the java code. On the other hand, in normal operations, just a top level message message without too much details are preferred. The exceptions are also always logged in the java-vm log but if this log is large it can be troublesome to correlate a certain exception to a specific action in Ncs. For this reason it is possible to configure the level of details shown by ncs for an java-vm exception. The leaf / ncs:java-vm/exception-error-message/verbosity takes one of three values:

- *standard* - Show the message from the top exception. This is the default
- *verbose* - Show all messages for the chain of cause exceptions, if any
- *trace* - Show messages for the chain of cause exceptions with exception class and the trace for the bottom root cause

Here is an example in how this can be used. In the web-site-service example we try to create a service without the necessary preparations:

Example 122. Setting error message verbosity

```
admin@ncs% set services web-site s1 ip 1.2.3.4 port 1111 url x.se
[ok][2013-03-25 10:46:46]

[edit]
admin@ncs% commit
Aborted: Service create failed
[error][2013-03-25 10:46:48]
```

This is a very generic error message with does not describe what really happens in the java code. Here the java-vm log has to be analyzed to find the problem. However, with this cli session open we can from another cli set the error reporting level to trace:

```
$ ncs_cli -u admin
admin@ncs> configure
admin@ncs% set java-vm exception-error-message verbosity trace
admin@ncs% commit
```

If we now in the original cli session issue the commit again we get the following error message that pinpoint the problem in the code:

```
admin@ncs% commit
Aborted: [com.tailf.dp.DpCallbackException] Service create failed
Trace : [java.lang.NullPointerException]
        com.tailf.conf.ConfKey.hashCode(ConfKey.java:145)
        java.util.HashMap.getEntry(HashMap.java:361)
        java.util.HashMap.containsKey(HashMap.java:352)
        com.tailf.navu.NavuList.refreshElem(NavuList.java:1007)
        com.tailf.navu.NavuList.elem(NavuList.java:831)
        com.example.webserviceservice.webserviceservice.WebSiteServiceRFS.crea...
        com.tailf.nsmux.NcsRfsDispatcher.applyStandardChange(NcsRfsDispa...
        com.tailf.nsmux.NcsRfsDispatcher.dispatch(NcsRfsDispatcher.java:...
        sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccesso...
```

```

sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethod...
java.lang.reflect.Method.invoke(Method.java:616)
com.tailf.dp.annotations.DataCallbackProxy.writeAll(DataCallback...
com.tailf.dp.DpTrans.protoCallback(DpTrans.java:1357)
com.tailf.dp.DpTrans.read(DpTrans.java:571)
com.tailf.dp.DpTrans.run(DpTrans.java:369)
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExec...
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExe...
java.lang.Thread.run(Thread.java:679)
com.tailf.dp.DpThread.run(DpThread.java:44)
[error][2013-03-25 10:47:09]

```

Loading Packages

NSO will first start take the packages found in the load path and copy these into a directory under supervision of NSO located at `./state/package-in-use`. Later starts of NSO will not take any new copies from the packages load-path so changes will not take effect by default. The reason for this is that in normal operation changing packages definition as a side-effect of a restart is an unwanted behavior. Instead these type of changes are part of an NSO installation upgrade.

During package development as opposed to operations it is usually desirable that all changes to package definitions in the package load-path takes effect immediately. There are two ways to make this happen. Either start ncs with the `--with-reload-packages` directive:

```
$ ncs --with-reload-packages
```

or set the environment variable `NCS_RELOAD_PACKAGES`, for example like this

```
$ export NCS_RELOAD_PACKAGES=true
```

It is a strong recommendation to use the `NCS_RELOAD_PACKAGES` environment variable approach since it guarantees that the packages are updated in all situations.

It is also possible to request a running NSO to reload all its packages.

```
admin@iron> request packages reload
```

This request can only be performed in operational mode, and the effect is that all packages will be updated, and any change in YANG models or code will be effectuated. If any YANG models are changed an automatic CDB data upgrade will be executed. If manual (user code) data upgrades are necessary the package should contain an *upgrade* component. This *upgrade* component will be executed as a part of the package reload. See the section called “[Writing an Upgrade Package Component](#)” for information how to develop an upgrade component.

If the change in a package does not affect the data model or shared Java code, there is another command

```
admin@iron> request packages package mypack redeploy
```

This will redeploy the private JARs in the Java VM for the Java package, restart Python VM for the Python package and reload the templates associated with the package. However this command will not be sensitive to changes in the YANG models or shared JARs for the Java package.

Debugging the Service and Using Eclipse IDE

By default, ncs will start the Java VM invoking the command `$NCS_DIR/bin/ncs-start-java-vm`. That script will invoke:

```
$ java com.tailf.ncs.NcsJVMLauncher
```

The class NcsJVMLauncher contains the main() method. The started java VM will automatically retrieve and deploy all java code for the packages defined in the load-path in the ncs.conf file. No other specification than the package-meta-data.xml for each package is needed.

In the NSO CLI, there exist a number of settings and actions for the NSO Java VM, if we do:

```
$ ncs_cli -u admin

admin connected from 127.0.0.1 using console on iron.local
admin@iron> show configuration java-vm | details
stdout-capture {
    enabled;
    file    ./logs/ncs-java-vm.log;
}
connect-time           30;
initialization-time   20;
synchronization-timeout-action log-stop;
java-thread-pool {
    pool-config {
        cfg-core-pool-size      5;
        cfg-keep-alive-time    60;
        cfg-maximum-pool-size 256;
    }
}
jmx {
    jndi-address 127.0.0.1;
    jndi-port     9902;
    jmx-address   127.0.0.1;
    jmx-port      9901;
}
[ok][2012-07-12 10:45:59]
```

We see some of the settings that are used to control how the NSO Java VM run. In particular here we're interested in /java-vm/stdout-capture/file

The NSO daemon will, when it starts, also start the NSO Java VM, and it will capture the stdout output from the NSO Java VM and send it to the file ./logs/ncs-java-vm.log. For more detail on the Java VM settings see [Chapter 5, The NSO Java VM](#).

Thus if we tail -f that file, we get all the output from the Java code. That leads us to the first and most simple way of developing the Java code. If we now:

- 1 Edit our Java code.
- 2 Recompile that code in the package, e.g cd ./packages/myrfs/src; make
- 3 Restart the Java code, either through telling NSO to restart the entire NSO Java VM from the NSO CLI (Note, this requires env variable NCS_RELOAD_PACKAGES=true):

```
admin@iron% request java-vm restart
result Started
[ok][2012-07-12 10:57:08]
```

Or instructing NSO to just redeploy the package we're currently working on.

```
admin@iron% request packages package stats redeploy
result true
[ok][2012-07-12 10:59:01]
```

We can then do tail -f logs/ncs-java-vm.log in order to check for printouts and log messages. Typically there is quite a lot of data in the NSO Java VM log. It can sometime be hard to find our own printouts and log messages. Therefore it can be convenient to use the command:

```
admin@iron% set java-vm exception-error-message verbosity trace
```

which will make the relevant exception stack traces visible in the CLI.

It's also possible to dynamically, from the CLI control the level of logging as well as which Java packages that shall log. Say that we're interested in Maapi calls, but don't want the log cluttered with what is really NSO Java library internal calls. We can then do:

```
admin@iron% set java-vm java-logging logger com.tailf.ncs level level-error
[ok][2012-07-12 11:10:50]
admin@iron% set java-vm java-logging logger com.tailf.conf level level-error
[ok][2012-07-12 11:11:15]
admin@iron% commit
Commit complete.
```

Now considerably less log data will come. If we want these settings to always be there, even if we restart NSO from scratch with an empty database (no .cdb file in ./ncs-cdb) we can save these settings as XML, and put that XML inside the ncs-cdb directory, that way ncs will use this data as initialization data on fresh restart. We do:

```
$ ncs_load -F p -p /ncs:java-vm/java-logging > ./ncs-cdb/loglevels.xml
$ ncs-setup --reset
$ ncs
```

The `ncs-setup --reset` command, stops the NSO daemon and resets NSO back into "factory defaults". A restart of NSO will reinitialize NSO from all XML files found in the CDB directory.

Running the NSO Java VM Standalone

It's possible to tell NSO to not start the NSO Java VM at all. This is interesting in two different scenarios. First is if want to run the NSO Java code embedded in a larger application, such as a Java Application Server (JBoss), the other is when debugging a package. First we configure NSO to not start the NSO Java VM at all by adding the following snippet to `ncs.conf`:

```
<java-vm>
  <auto-start>false</auto-start>
</java-vm>
```

Now, after a restart or a configuration reload, no Java code is running, if we do:

```
admin@iron> show status packages
```

we will see that the `oper-status` of the packages is `java-uninitialized`. We can also do

```
admin@iron> show status java-vm
start-status auto-start-not-enabled;
status      not-connected;
[ok][2012-07-12 11:27:28]
```

And this is expected, since we've told NSO to not start the NSO Java VM. Now, we can do that manually, at the UNIX shell prompt.

```
$ ncs-start-java-vm
.....
.. all stdout from NCS Java VM
```

So, now we're in position where we can manually stop the NSO Java VM, recompile the Java code, restart the NSO Java VM. This development cycle works fine. However, even though we're running the NSO Java VM standalone, we can still redeploy packages from the NSO CLI as

```
admin@iron% request packages package stats redeploy
```

```
result true
[ok][2012-07-12 10:59:01]
```

to reload and restart just our Java code, no need to restart the NSO Java VM.

Using Eclipse to Debug the Package Java Code

Since we can run the NSO Java VM standalone in a UNIX Shell, we can also run it inside eclipse. If we stand in a NSO project directory, like NCS generated earlier in this chapter, we can issue the command

```
$ ncs-setup --eclipse-setup
```

This will generate two files, `.classpath` and `.project`. If we add this directory to eclipse as a "File->New->Java Project", uncheck the "Use the default location" and enter the directory where the `.classpath` and `.project` have been generated. We're immediately ready to run this code in eclipse. All we need to do is to choose the `main()` routine in the `NcsJVMLauncher` class.

The eclipse debugger works now as usual, and we can at will start and stop the Java code. One caveat here which is worth mentioning is that there are a few timeouts between NSO and the Java code that will trigger when we sit in the debugger. While developing with the eclipse debugger and breakpoints we typically want to disable all these timeouts.

First we have 3 timeouts in `ncs.conf` that matter. Copy the system `ncs.conf` and set the three values of

```
/ncs-config/japi/new-session-timeout
/ncs-config/japi/query-timeout
/ncs-config/japi/connect-timeout
```

to a large value. See man page `ncs.conf(5)` for a detailed description on what those values are. If these timeouts are triggered, NSO will close all sockets to the Java VM and all bets are off.

```
$ cp $NCS_DIR/etc/ncs/ncs.conf .
```

Edit the file and enter the following XML entry just after the `Webui` entry.

```
<japi>
  <new-session-timeout>PT1000S</new-session-timeout>
  <query-timeout>PT1000S</query-timeout>
  <connect-timeout>PT1000S</connect-timeout>
</japi>
```

Now restart ncs.

We also have a few timeouts that are dynamically reconfigurable from the CLI. We do:

```
$ ncs_cli -u admin

admin connected from 127.0.0.1 using console on iron.local
admin@iron> configure
Entering configuration mode private
[ok][2012-07-12 12:54:13]
admin@iron% set devices global-settings connect-timeout 1000
[ok][2012-07-12 12:54:31]

[edit]
admin@iron% set devices global-settings read-timeout 1000
[ok][2012-07-12 12:54:39]

[edit]
admin@iron% set devices global-settings write-timeout 1000
[ok][2012-07-12 12:54:44]
```

```
[edit]
admin@iron% commit
Commit complete.
```

and then to save these settings so that ncs will have them again on a clean restart (no cdb files)

```
$ ncs_load -F p -p /ncs:devices/global-settings > ./ncs-cdb/global-settings.xml
```

Remote Connecting with Eclipse to the NSO Java VM

The eclipse Java debugger can connect remotely to a NSO Java VM and debug that NSO Java VM. This requires that the NSO Java VM has been started with some additional flags. By default the script in `$NCS_DIR/bin/ncs-start-java-vm` is used to start the NSO Java VM. If we provide the `-d` flag, we will launch the NSO Java VM with

```
"-Xdebug -Xrunjdwp:transport=dt_socket,address=9000,server=y,suspend=n"
```

This is what is needed to be able to remote connect to the NSO Java VM, in the `ncs.conf` file

```
<java-vm>
  <start-command>ncs-start-java-vm -d</start-command>
</java-vm>
```

Now if we in Eclipse, add a "Debug Configuration" and connect to port 9000 on localhost, we can attach the Eclipse debugger to an already running system and debug it remotely.

Working with ncs-project

An NSO project is a complete running NSO installation. It contain all the needed packages and the config data that is required to run the system.

By using the `ncs-project` commands, the project can be populated with the necessary packages and kept updated. This can be used for encapsulating NSO demos or even a full blown turn-key system.

For a developer, the typical workflow looks like this:

- Create a new project using the `ncs-project create` command
- Define what packages to use in the `project-meta-data.xml` file.
- Fetch any remote packages with the `ncs-project update` command.
- Prepare any initial data and/or config files.
- Run the application.
- Possibly export the project for somebody else to run.

Create a new project

Using the `ncs-project create` command, a new project is created. The file `project-meta-data.xml` should be updated with relevant information as will be described below. The project will also get a default `ncs.conf` configuration file that can be edited to better match different scenarios. All files and directories should be put into a version control system, such as `git`.

Example 123. Creating a new project

```
$ ncs-project create test_project
Creating directory: /home/developer/dev/test_project
Using NCS 5.7 found in /home/developer/ncs_dir
wrote project to /home/developer/dev/test_project
```

A directory called `test_project` is created containing the files and directories of a NSO project as shown below:

```
test_project/
|-- init_data
|-- logs
|-- Makefile
|-- ncs-cdb
|-- ncs.conf
|-- packages
|-- project-meta-data.xml
|-- README.ncs
|-- scripts
|-- |-- command
|-- |-- post-commit
|-- setup.mk
|-- state
|-- test
|-- |-- internal
|-- |-- |-- lux
|-- |-- |-- basic
|-- |-- |-- |-- Makefile
|-- |-- |-- |-- run.lux
|-- |-- |-- Makefile
|-- |-- Makefile
|-- Makefile
|-- pkgtest.env
```

The `Makefile` contains targets for building, starting, stopping and cleaning the system. It also contains targets for entering the CLI as well as some useful targets for dealing with any git packages. Study the `Makefile` to learn more.

Any initial CDB data can be put in the `init_data` directory. The `Makefile` will copy any files in this directory to the `ncs-cdb` before starting NSO.

There is also a `test` directory created with a directory structure used for automatic tests. These tests are dependent on the test tool *Lux* (<https://github.com/hawk/lux.git>).

Project setup

To fill this project with anything meaningful, the `project-meta-data.xml` file needs to be edited.

Project version number is configurable, the version we get from the *create* command is 1.0. The description should also be changed to a small text explaining what the project is intended for. Our initial content of the `project-meta-data.xml` may now look like this:

Example 124. Project meta data

```
<project-meta-data xmlns="http://tail-f.com/ns/ncs-project">
  <name>test_project</name>
  <project-version>1.0</project-version>
  <description>Skeleton for a NCS project</description>

  <!-- More things to be added here -->

</project-meta-data>
```

For this example, let say we have a released package: `ncs-4.1.2-cisco-ios-4.1.5.tar.gz`, a package located in a remote git repository `foo.git`, and a local package that we have developed ourself: `mypack`. The relevant part of our `project-meta-data.xml` file would then look like this:

Example 125. Package project meta data

```
<!-- we will add a package-store section here -->
<!-- we will add a netsim section here -->

<package>
  <name>cisco-ios</name>
  <url>file:///tmp/ncs-4.1.2-cisco-ios-4.1.5.tar.gz</url>
</package>

<package>
  <name>foo</name>
  <git>
    <repo>ssh://git@my-repo.com/foo.git</repo>
    <branch>stable</branch>
  </git>
</package>

<package>
  <name>mypack</name>
  <local/>
</package>
```

By specifying netsim devices in the `project-meta-data.xml` file, the necessary commands for creating the netsim configuration will be generated in the `setup.mk` file that `ncs-project update` creates. The `setup.mk` file is included in the top `Makefile`, and provides some useful make targets for creating and deleting our netsim setup.

Example 126. Netsim project meta data

```
<netsim>
  <device>
    <name>cisco-ios</name>
    <prefix>ce</prefix>
    <num-devices>2</num-devices>
  </device>
</netsim>
```

When done editing the `project-meta-data.xml`, run the command `ncs-project update`. Add the `-v` switch to see what the command does.

Example 127. NSO Project update

```
$ ncs-project update -v
ncs-project: installing packages...
ncs-project: found local installation of "mypad"
ncs-project: unpacked tar file: /tmp/ncs-4.1.2-cisco-ios-4.1.5.tar.gz
ncs-project: git clone "ssh://git@my-repo.com/foo.git" "/home/developer/dev/test_project/pack
ncs-project: git checkout -q "stable"
ncs-project: installing packages...ok
ncs-project: resolving package dependencies...
ncs-project: resolving package dependencies...ok
ncs-project: determining build order...
ncs-project: determining build order...ok
ncs-project: determining ncs-min-version...
ncs-project: determining ncs-min-version...ok
The file 'setup.mk' will be overwritten, Continue (y/n)?
```

Answer *yes* when asked to overwrite the `setup.mk`. After this a new runtime directory is created with `ncs` and simulated devices configured. You are now ready to compile your system with: `make all`.

If you have a lot of packages, all located at the same git repository, it is convenient to specify the repository just once. This can be done by adding a `packages-store` section as shown below:

Example 128. Project packages store

```
<packages-store>
  <git>
    <repo>ssh://git@my-repo.com</repo>
    <branch>stable</branch>
  </git>
</packages-store>

<!-- then it is enough to specify the package like this: -->
<package>
  <name>foo</name>
  <git/>
</package>
```

This means that if a package does not have a git repository defined, the repository and branch in the `packages-store` is used.



Note

If a package has specified that it is dependent on some other packages in its `package-meta-data.xml` file, `ncs-project update` will try to clone those packages from any of the specified `packages-store`. To override this behaviour, specify explicitly all packages in your `project-meta-data.xml` file.

Export



Note

When the development is done the project can be bundled together and distributed further. The `ncs-project` comes with a command, `export`, used for this purpose. The `export` command creates a tarball of the required files and any extra files as specified in the `project-meta-data.xml` file.

Developers are encouraged to distribute the project, either via some Source Code Managements system, like git, or by exporting bundles using the `export` command.

When using `export`, a subset of the packages should be configured for exporting. The reason for not exporting all packages in a project is if some of the packages is used solely for testing or similar. When configuring the bundle the packages included in the bundle are leafrefs to the packages defined at the root of the model, see [Example 132, “The NSO Project YANG model”](#). We can also define a specific tag, commit or branch, even a different location for the packages, different from the one used while developing. For example we might develop against an experimental branch of a repository, but bundle with a specific release of that same repository.



Note

Bundled packages specified as of type `file://` or `url://` will not be built, they will simply be included as is by the `export` command.

The bundle also have a name and a list of included files. Unless another name is specified from the command line the final compressed file will be named using the configured bundle name and project version.

We create the tar-ball by using the `export` command:

Example 129. NSO Project export

```
$ ncs-project export
```

There are two ways to make use of a bundle:

- Together with the `ncs-project create --from-bundle=<bundlefile>` command
- Extract the included packages using `tar` for manual installation in an NSO deployment.

In the first scenario, it is possible to create an NSO project, populated with the packages from the bundle, to create a ready to run NSO system. The optional `init_data` part makes it possible to prepare CDB with configuration, prior to starting the system the very first time. The `project-meta-data.xml` file will specify all the packages as *local* to avoid any dangling pointers to non-accessible git repositories.

The second scenario is intended for the case when you want to install the packages manually, or via a custom process, into your running NSO systems.

The switch `--snapshot` will add a timestamp in the name of the created bundle file in order to make it clear that it is not a proper version numbered release.

To import our exported project we would do an `ncs-project create` and point out where the bundle is located.

Example 130. NSO Project import

```
$ ncs-project create --from-bundle=test_project-1.0.tar.gz
```

NSO Project man pages

`ncs-project` has a full set of manpages that describes its usage and syntax. Below is an overview of the commands which will be explained in more detail further down below.

Example 131. NSO Project man page

```
$ ncs-project --help

Usage: ncs-project <command>

COMMANDS

create      Create a new ncs-project
update      Update the project with any changes in the
            project-meta-data.xml
git         For each git package repo: execute an arbitrary git
            command.
export      Export a project, including init-data and configuration.
help        Display the man page for <command>

OPTIONS

-h, --help          Show this help text.
-n, --ncs-min-version  Display the NCS version(s) needed
                        to run this project
```

```
--ncs-min-version-non-strict      As -n, but include the non-matching
                                  NCS version(s)
```

See manpage for ncs-project(1) for more info.

The project-meta-data.xml file

The project-meta-data.xml file defines the project meta data for a NSO project according to the \$NCS_DIR/src/ncs/ncs_config/tailf-ncs-project.yang YANG model. See the tailf-ncs-project.yang module where all options are described in more detail. To get an overview, use the IETF RFC 8340 based YANG tree diagram.

Example 132. The NSO Project YANG model

```
$ yanger -f tree tailf-ncs-project.yang

module: tailf-ncs-project
  +-rw project-meta-data
    +-+rw name                  string
    +-+rw project-version?     version
    +-+rw description?         string
    +-+rw packages-store
      |  +-+rw directory* [name]
      |  |  +-+rw name      string
      |  +-+rw git* [repo]
      |    +-+rw repo          string
      |    +-+rw (git-type)?
      |      +--+:(branch)
      |      |  +-+rw branch?   string
      |      +--+:(tag)
      |      |  +-+rw tag?      string
      |      +--+:(commit)
      |      |  +-+rw commit?   string
    +-+rw netsim
      |  +-+rw device* [name]
      |    +-+rw name           -> /project-meta-data/package/name
      |    +-+rw prefix          string
      |    +-+rw num-devices     int32
    +-+rw bundle!
      |  +-+rw name?            string
      |  +-+rw includes
      |    |  +-+rw file* [path]
      |    |  |  +-+rw path      string
      |  +-+rw package* [name]
        |  +-+rw name           -> ../../../../package/name
        |  +-+rw (package-location)?
        |    +--+:(local)
        |    |  +-+rw local?      empty
        |    +--+:(url)
        |    |  +-+rw url?       string
        |    +--+:(git)
        |      +-+rw git
          |      +-+rw repo?      string
          |      +-+rw (git-type)?
          |        +--+:(branch)
          |        |  +-+rw branch?   string
          |        +--+:(tag)
          |        |  +-+rw tag?      string
          |        +--+:(commit)
          |          +-+rw commit?   string
    +-+rw package* [name]
```

The project-meta-data.xml file

```

    +-rw name          string
    +-rw (package-location)?
      +--:(local)
      |  +-rw local?   empty
      +--:(url)
      |  +-rw url?    string
      +--:(git)
      |  +-rw git
      |    +-rw repo?   string
      |    +-rw (git-type)?
      |      +--:(branch)
      |      |  +-rw branch?  string
      |      +--:(tag)
      |      |  +-rw tag?    string
      |      +--:(commit)
      |        +-rw commit?  string

```

Example 133. Example bundle project-meta-data.xml file

```

<project-meta-data xmlns="http://tail-f.com/ns/ncs-project">
  <name>l3vpn-demo</name>
  <project-version>1.0</project-version>
  <description>l3vpn demo</description>
  <bundle>
    <!-- filename default -->
    <name>example_bundle</name>
    <package>
      <name>my-package-1</name>
      <local/>
    </package>
    <!-- The same package as used by the project, but with a specific URL -->
    <package>
      <name>my-package-2</name>
      <url>http://localhost:9999/my-local.tar.gz</url>
    </package>
    <package>
      <name>my-package-3</name>
      <git>
        <repo>ssh://git@example.com/pkg/resource-manager.git</repo>
        <tag>1.2</tag>
      </git>
    </package>
  </bundle>
  <package>
    <name>my-package-1</name>
    <local/>
  </package>
  <package>
    <name>my-package-2</name>
    <local/>
  </package>
  <package>
    <name>my-package-3</name>
    <git>
      <repo>ssh://git@example.com/pkg/resource-manager.git</repo>
      <tag>1.2</tag>
    </git>
  </package>
</project-meta-data>

```

Below is a list of the settings in the `tailf-ncs-project.yang` that is configured thorough the meta data file. A detailed description can be found in the YANG model.

**Note**

The order of the XML entries in a `project-meta-data.xml` must be in the same order as the model.

- `name` - A unique name of the project.
- `project-version` - the version of the project. This is for administrative purposes only.
- `packages-store`
 - `directory` - paths for package dependencies.
 - `git`
 - `repo` - default git package repositories.
 - `branch`, `tag`, or `commit id`
- `netsim` - list netsim devices used by the project to generate a proper Makefile running the 'ncs-project setup' script.
 - `device`
 - `prefix`
 - `num-devices`
- `bundle` - Information to collect files and packages to pack them in a tarball bundle.
 - `name` - tarball filename.
 - `includes` - files to include.
 - `package` - packages to include (leafref to the package list below)
 - `name` - name of the package.
 - `local`, `url`, or `git` - where to get the package. GIT option need a branch, tag, or commit id.
- `package` - packages used by the project.
 - `name` - name of the package.
 - `local`, `url`, or `git` - where to get the package. GIT option need a branch, tag, or commit id.



CHAPTER 14

Developing NSO Services

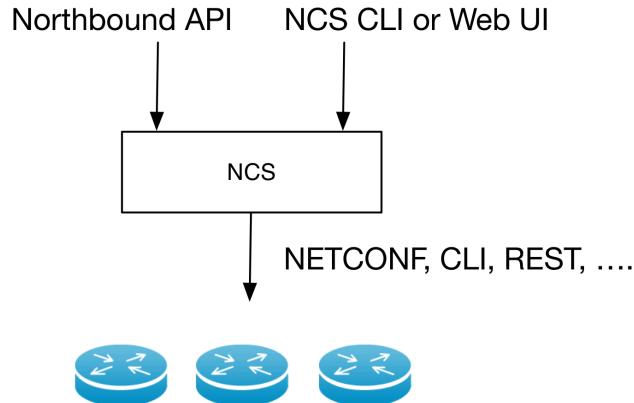
- [Introduction, page 229](#)
- [Definitions, page 230](#)
- [The Fundamentals, page 231](#)
- [Writing the Service Model, page 232](#)
- [Finding the Mapping, page 232](#)
- [Strategies to Implement the Mapping, page 234](#)
- [Creating an NSO Service Application, page 234](#)
- [Mapping using Service Templates, page 235](#)
- [Mapping using Java, page 245](#)
- [Mapping using Java combined with Templates, page 260](#)
- [Service Mapping: Putting Things Together, page 263](#)
- [FASTMAP Description, page 273](#)
- [Reactive FASTMAP , page 275](#)
- [Services that involve virtual devices, NFV, page 328](#)
- [Advanced Mapping Techniques , page 330](#)
- [Service Discovery, page 339](#)
- [Partial sync, page 345](#)

Introduction

This section describes how to develop a service application. A service application maps input parameters to create, modify, and delete a service instance into the resulting native commands to devices in the network. The input parameters are given from a northbound system such as a self-service portal via API calls to NSO or a network engineer using any of the NSO User Interfaces such as the NSO CLI.

Figure 134. 10 000 Feet View of Service Applications

Create VPN for Customer ACME!



The service application has a single task: from a given set of input parameters for a service instance modification, calculate the minimal set of device interface operations to achieve the desired service change.

It is very important that the service application supports *any* change, i.e., full create, delete, and update of any service parameter.

Definitions

Below follows a set of definitions that is used throughout this section:

<i>Service type</i>	A specific type of service like "L2 VPN", "L3 VPN", "VLAN", "Firewall Rule set".
<i>Service instance</i>	A specific instance of a service type, such as "ACME L3 VPN"
<i>Service model</i>	The schema definition for a service type. In NSO YANG is used as the schema language to define service types. Service models are used in different contexts/systems and therefore have slightly different meanings. In the context of NSO, a service model is a black-box specification of the attributes required to instantiate the service.
	This is different from service models in ITIL-based CMDBs or OSS inventory systems, where a service model is more of a white-box model that describes the complete structure.
<i>Service application</i>	The code that implements a service, i.e., maps the parameters for a service instance to device configuration.
<i>Device configuration</i>	Network devices are configured to perform network functions. Every service instance results in corresponding device configuration changes. The dominating way to represent and change device configurations in current networks are CLI representations and sequences. NETCONF represents the configuration as XML instance documents corresponding to the YANG schema.

The Fundamentals

Mapping

Developing a service application that transforms a service request to corresponding device configurations is done differently in NSO than in other tools on the market. It is therefore important to understand the underlying fundamental concepts and how they differ from what you might assume.

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

Note well that irrespective of the underlying device type and corresponding native device interface, the mapping is towards a YANG device model, not the native CLI for example. This means that as you write the service mapping, you do not have to worry about the syntax of different devices' CLI commands or in which order these commands are sent to the devices. This is all taken care of by the NSO device manager.

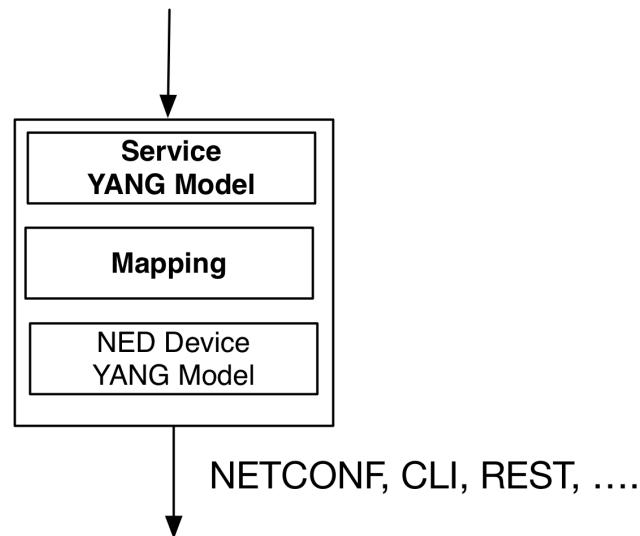
The above means that implementing a service in NSO is reduced to transforming the input data structure (described in YANG) to device data structures (also described in YANG).

Who writes the models?

- Developing the service model is part of developing the service application and is covered later in this chapter.
- Every device NED comes with a corresponding device YANG model. This model has been designed by the NED developer to capture the configuration data that is supported by the device.

This means that a service application has two primary artifacts: a YANG service model and a mapping definition to the device YANG as illustrated below.

Figure 135. Service Application Artifacts



At this point you should realize the following:

- The mapping is not defined using workflows, or sequences of device commands.
- The mapping is not defined in the native device interface language.

FASTMAP and Transactions

A common problem for systems that tries to automate service activation is that a "back-end" needs to be defined for every possible service instance change. Take for example a L3 VPN, a northbound system or a network engineer may during a service life-cycle want to:

- Create the VPN
- Add a leg to the VPN
- Remove a leg from the VPN
- Modify the bandwidth of a VPN leg
- Change the interface of a VPN leg
- ...
- Delete the VPN

The possible run-time changes for an existing service instance are numerous. If a developer has to define a back-end for every possible change, like a script or a workflow, the task is daunting, error-prone, and never-ending.

NSO reduces this problem to a single data-mapping definition for the "create" scenario. At run-time NSO will render the minimum change for any possible change like all the ones mentioned below. This is managed by the FASTMAP algorithm explained later in this section.

Another challenge in traditional systems is that a lot of code goes into managing error scenarios. The NSO built-in transaction manager takes that away from the developer of the Service Application.

Auto-rendering from the Service Model

Since NSO automatically renders the northbound APIs and database schema from the YANG models, NSO enables a DevOps way of working with service models. A new service model can be defined as part of a package and loaded into NSO. An existing service model can be modified and the package upgraded. All northbound APIs and User Interfaces are automatically re-rendered to cater for the new models or updated models.

Writing the Service Model

The YANG Service Model specifies the input parameters to NSO. For a specific service model think of the parameters that a northbound system sends to NSO or the parameters that a network engineer needs to enter in the NSO CLI.

This model can be iterated without having any mapping defined. Write the YANG model, reload the service package in NSO and try the model with network engineers or northbound systems.

The result of this exercises for a L3 VPN service might be:

- VPN name
- AS Number
- End-point CE device and interface
- End-point PE device and interface

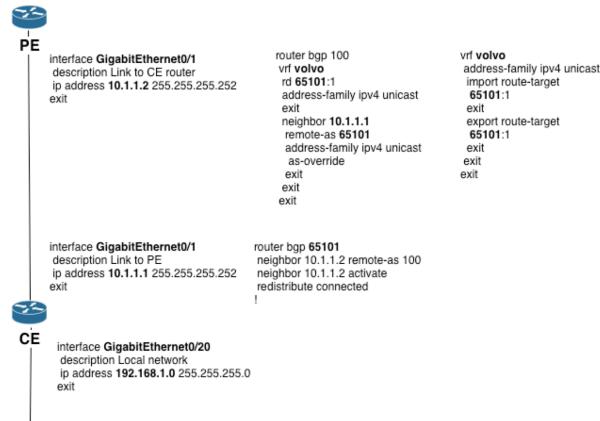
Finding the Mapping

The most straight-forward way of finding the mapping is to create one example of the service instance manually on the devices. Either create it using the native device interface and then synchronize the configuration into NSO, or use the NSO CLI to create the device configuration.

Based on this example device configuration for a service instance, note which part of the device configuration are variables resulting from the service configuration.

The figure below illustrates an example VPN configuration. Configuration items in bold are variables that are mapped from the service input.

Figure 136. Example L3 VPN Device Configuration



Now look at the attributes of the service model and make sure you have a clear picture how the values are mapped into the corresponding device configuration.

Mapping Iterations

During the above exercises you might come into a situation where the input parameters for a service are not sufficient to render the device configuration.

Examples:

- Assume the northbound system only provides the CE device and wants NSO to pick the right PE.
- Assume the northbound system wants NSO to pick an IP address and does not pass that as an input parameter.

This is part of the service design iteration. If the input parameters are not sufficient to define the corresponding device configuration you either add more attributes to the service model so that the device configuration data can be defined as a pure data model mapping or you assume the mapping can fetch the missing pieces.

In the latter case there are several alternatives. All of these will be explained in detail later. Typical patterns are listed below:

- If the mapping needs pre-configured data, you can define a YANG data model for this data. For example, in the VPN case NSO could have a list of CE-PE links loaded into NSO and the mapping then uses this list to find the PE for a CE and the PE therefore does not need to be part of the service model.
- If the mapping needs to request data from an external system, for example query an IP address manager for the IP addresses, you can use the [Reactive FASTMAP pattern](#).
- Use NSO to handle allocation of resources like VLAN IDs etc. A package can be defined to manage VLAN pools within NSO and the mapping then requests a new VLAN from the VLAN pool and therefore it needs not to be passed as input. The [Reactive FASTMAP pattern](#) is used in this case as well.

Strategies to Implement the Mapping

This section gives an overview of the different design patterns to define the mapping. NSO provides three different ways to express the YANG service model to YANG device model mapping:

Service templates

If the mapping is a pure data model mapping without any complex calculations, algorithms, external call-outs, resource management or Reactive FASTMAP patterns, the mapping can be defined as service templates. Service templates requires no programming skills and are derived from example device configurations. They are therefore well suited for network engineers. See `examples.ncs/service-provider/simple-mpls-vpn` for an example.

Java and configuration templates

This is the most common technique for real-life deployments. A thin layer of Java implements the device-type independent algorithms and passes variables to templates that maps this into device specific configurations across vendors. The templates are often defined "per feature". This means that the Java code calculates a number of variables that are device independent. The Java code then applies templates with these variables as inputs, and the templates maps this to the various device types. All device-specifics are done in the templates, thus keeping the Java code clean. See `examples.ncs/service-provider/mpls-vpn` for an example.

Java only

There are no real benefits of this approach compared to the above combination of Java and templates. This more depends on the skills of the developer, programmers with less networking skills might prefer this approach. Abstracting away different device vendors are often more cumbersome than in the Java and templates approach. See `examples.ncs/datacenter/datacenter` for an example.

Creating an NSO Service Application

The purpose of this section is to outline the overall steps in NSO to create a service application. The following sections exemplifies these steps for the different mapping strategies. The command **ncs-make-package** in *NSO 5.7 Manual Pages* is used in these examples to create a skeleton service package.

All of the below assume you have a NSO local installation (see Chapter 2, *NSO Local Install* in *NSO Installation Guide*, and have created an NSO instance with **ncs-setup** in *NSO 5.7 Manual Pages*. This command creates the NSO instance in a directory, called the NSO runtime directory, which is specified on the command line:

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

In this example the NSO runtime directory is `./ncs-run`.

Step 1

Generate a service package in the packages directory in the runtime directory. In this example, the package name is `vlan`, and it is a service package with java code and templates:

```
$ cd ncs-run/packages
$ ncs-make-package --service-skeleton TYPE PACKAGE-NAME
```

Step 2

Edit the skeleton YANG service model in the generated package. The YANG file resides in `PACKAGE-NAME/src/yang`

Step 3 Build the service model:

```
$ cd PACKAGE-NAME/src
$ make
```

Step 4 Try the service model in the NSO CLI. In order to have NSO to load the new package including the service model do:

```
admin@ncs# packages reload
```

Step 5 Iterate the above steps from [Step 2](#) until you have a service model you are happy with.

Step 6 If the service does not have any templates, continue with [Step 11](#)

Create an example device configuration either directly on the devices or by using the NSO CLI. This can be done either using netsim or real devices. In case the configuration was created directly on the devices, synchronize the configuration back into NSO:

```
admin@ncs# devices sync-from
```

Step 7 Save the example device configuration as an XML file which is the format of templates.

```
admin@ncs# show full-configuration devices devices config ... | display xml | file save mytemplate.xml
```

Step 8 Move the XML file to the template folder of the package.

Step 9 Replace hard-coded values of the XML template with variables referring to the service model or variables passed from the Java code. This is explained in detail later in this section.

Step 10 If this template is a template without any Java code make sure the service-point name in the YANG service model has a corresponding service-point in the XML file. Again this is explained in detail later.

Step 11 If a Java mapping layer is included, modify the Java in the src/java directory. Build the Java code:

```
$ cd PACKAGE-NAME/src
$ make
```

Step 12 Reload the packages; this reloads both the data models and the Java code:

```
admin@ncs# packages reload
```

Step 13 Try the mapping by creating and modifying service instances in the CLI. Validate the changes by:

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

Mapping using Service Templates

In this example, you will create a simple VLAN service using a mapping with service templates only (i.e., no Java code). To keep the example simple, it will use only one single device type (IOS).

Preparation

In order to reuse an existing environment for NSO and netsim, the examples.ncs/getting-started/using-ncs/1-simulated-cisco-ios/ example is used. Make sure you have stopped any running NSO and netsim.

Step 1 Navigate to the example directory:

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

Step 2 Now you need to create a environment for the simulated IOS devices. This is done using the command **ncs-netsim** in [NSO 5.7 Manual Pages](#).

```
$ ncs-netsim create-network $NCS_DIR/packages/neds/cisco-ios 3 c
DEVICE c0 CREATED
DEVICE c1 CREATED
DEVICE c2 CREATED
```

This command creates the simulated network in ./netsim.

Step 3 Next, you need an NSO instance with the simulated network:

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

Defining the Service Model

Step 1 The first step is to generate a skeleton package for a service. (For details on packages see the section called “*Packages*” in *NSO 5.7 Getting Started Guide*). The package is called `vlan`:

```
$ cd packages
$ ncs-make-package --service-skeleton template vlan
```

This results in a directory structure:

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

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

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

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

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

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

            leaf name {
                type string;
            }

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

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

```

    }
}
```

If this is your first exposure to YANG you can see that the modeling language is very straightforward and easy to understand. See [RFC 6020](#) for more details and examples for YANG.

The concepts you should understand in the above generated skeleton are:

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

Step 2

The next step is to modify the skeleton service YANG model and add the real parameters.

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

```

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

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

        leaf name {
            type string;
        }

        leaf vlan-id {
            type uint32 {
                range "1..4096";
            }
        }

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

This simple VLAN service model says:

- 1 Each VLAN must have a unique name, for example "net-1".
- 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.

Step 3 The next step is to build the data model:

```
$ cd $NCS_DIR/examples.ncs/getting-started/using-ncs/1-simulated-cisco-ios/packages/vlan/src
$ make
.../ncsc `ls vlan-ann.yang > /dev/null 2>&1 && echo "-a vlan-ann.yang" ` \
-c -o ../load-dir/vlan.fxs yang/vlan.yang
```

A nice property of NSO is that already at this point you can load the service model into NSO and try if it works well in the CLI etc. Nothing will happen to the devices since the mapping is not defined yet. This is normally the way to iterate a model; load it into NSO, test the CLI towards the network engineers, make changes, reload it into NSO etc.

Step 4 Go to the root directory of the simulated-ios example:

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

Step 5 Start netsim and NSO:

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

When NSO was started above, you gave NSO a parameter to reload all packages so that the newly added vlan package is included. Without this parameter, NSO starts with the same packages as last time. Packages can also be reloaded without starting and stopping NSO.

Step 6 Start the NSO CLI:

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

Step 7 Since this is the first time NSO is started with some devices, you need to make sure NSO synchronizes its database with the devices:

```
admin@ncs# devices sync-from
sync-result {
    device c0
    result true
}
sync-result {
    device c1
    result true
}
sync-result {
    device c2
    result true
}
```

Step 8 At this point we have a service model for VLANs, but no mapping of VLAN to device configurations. This is fine; you can try the service model and see if it makes sense. Create a VLAN service:

```
admin@ncs# config
Entering configuration mode terminal
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 native
admin@ncs(config)# commit
Commit complete.

```

Committing service changes at this point has no effect on the devices since there is no mapping defined. This is why the output to the command **commit dry-run outformat native** doesn't show any output. The service instance data will just be stored in the data base in NSO.

Note that you get tab completion on the devices since they are references to device names in CDB. You also get tab completion for interface types 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, a reference to the ios interface name according to the IOS model could be used. 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 a string, but this is omitted here for simplicity.

Make sure you delete the vlan service instance before moving on with the example:

```

admin@ncs(config)# no services vlan
admin@ncs(config)# commit
Commit complete.

```

Defining the Template

Step 1

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. In this example, this is done by manually configuring one vlan on a device. This concrete device configuration is a starting point 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

```

Defining the Template

```

switchport trunk allowed vlan 1234
exit
!
!
```

Step 2

The concrete configuration above has the interface and VLAN hard-wired. This is what we now will make into a template. It is always recommended to start like this 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. Display the device configuration 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>
```

Step 3

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?>
        </interface>
      </config>
    </device>
  </devices>
</config-template>

```

Defining the Template

```

        </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 instance 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 to replace).
- 2 For 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.



Tip While experimenting with the template it can be helpful to remove the nocreate tag. In that way you will always create configuration from the template even if the interface 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`.

For details on the template syntax, see [the section called “Service Templates”](#)

Step 4

Throw away the uncommitted changes to the device, and request NSO to reload the packages:

```

admin@ncs(config)# exit no-confirm
admin@ncs# packages reload
reload-result {
    package cisco-ios
    result true
}
reload-result {
    package vlan
    result true
}

```

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

Step 5

We can now create services that will make things happen in the network. Create a VLAN service:

```

admin@ncs# config
Entering configuration mode terminal
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 vlan 1234
    !
    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 | details
...
Commit complete.

```

Note that the **commit** command stored the service data in NSO, and at the same time pushed the changes to the two devices affected by the service.

Step 6

The VLAN service instance can now be changed:

```

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 {

```

Defining the Template

```

name c1
data no vlan 1234
    vlan 1222
!
interface FastEthernet1/0
    switchport trunk allowed vlan 1222
exit
}
}
admin@ncs(config)# commit
Commit complete.

```

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 any explicit mappings to for 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.

```

Step 7

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.
```

- Step 8** To make the VLAN service package complete edit the `vlan/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.

Mapping using Java

Overview

This section will illustrate how to implement a simple VLAN service in Java. The end-result will be the same as shown previously using templates but this time implemented in Java instead.

Note well that the examples in this section are extremely simplified from a networking perspective in order to illustrate the concepts.

We will first look at the following preparatory steps:

-
- Step 1** Prepare a simulated environment of Cisco IOS devices: in this example we start from scratch in order to illustrate the complete development process. We will not reuse any existing NSO examples.
- Step 2** Generate a template service skeleton package: use NSO tools to generate a Java based service skeleton package.
- Step 3** Write and test the VLAN Service Model.
- Step 4** Analyze the VLAN service mapping to IOS configuration.
-

The above steps are no different from defining services using templates. Next is to start playing with the Java Environment:

- 1 Configuring start and stop of the Java VM.
- 2 First look at the Service Java Code: introduction to service mapping in Java.
- 3 Developing by tailing log files.
- 4 Developing using Eclipse.

Setting up the environment

We will start by setting up a run-time environment that includes simulated Cisco IOS devices and configuration data for NSO. Make sure you have sourced the `ncsrc` file. Create a directory somewhere like:

```
$ mkdir ~/vlan-service  
$ cd ~/vlan-service
```

Now lets create a simulated environment with 3 IOS devices and a NSO that is ready to run with this simulated network:

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

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

Start the simulator and NSO:

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

Use the Cisco CLI towards one of the devices:

```
$ ncs-netsim cli-i c0
admin connected from 127.0.0.1 using console on ncs
c0> enable
c0# configure
Enter configuration commands, one per line. End with CNTL/Z.
c0(config)# show full-configuration
no service pad
no ip domain-lookup
no ip http server
no ip http secure-server
ip routing
ip source-route
ip vrf my-forward
bgp next-hop Loopback 1
!
...
```

Use the NSO CLI to get the configuration:

```
$ ncs_cli -C -u admin

admin connected from 127.0.0.1 using console on ncs
admin@ncs# devices sync-from
sync-result {
    device c0
    result true
}
sync-result {
    device c1
    result true
}
sync-result {
    device c2
    result true
}
admin@ncs# config
Entering configuration mode terminal

admin@ncs(config)# show full-configuration devices device c0 config
devices device c0
config
no ios:service pad
ios:ip vrf my-forward
bgp next-hop Loopback 1
!
ios:ip community-list 1 permit
ios:ip community-list 2 deny
ios:ip community-list standard s permit
no ios:ip domain-lookup
no ios:ip http server
no ios:ip http secure-server
```

```
ios:ip routing
...
```

Finally, set VLAN information manually on a device to prepare for the mapping later.

```
admin@ncs(config)# devices device c0 config ios:vlan 1234
admin@ncs(config)# devices device c0 config ios:interface
    FastEthernet 1/0 switchport mode trunk
admin@ncs(config-if)# switchport trunk allowed vlan 1234
admin@ncs(config-if)# top

admin@ncs(config)# show configuration
devices device c0
config
  ios:vlan 1234
  !
  ios:interface FastEthernet1/0
    switchport mode trunk
    switchport trunk allowed vlan 1234
  exit
  !
  !

admin@ncs(config)# commit
```

Creating a service package

In the run-time directory you created:

```
$ ls -F1
README.ncs
README.netsim
logs/
ncs-cdb/
ncs.conf
netsim/
packages/
scripts/
state/
```

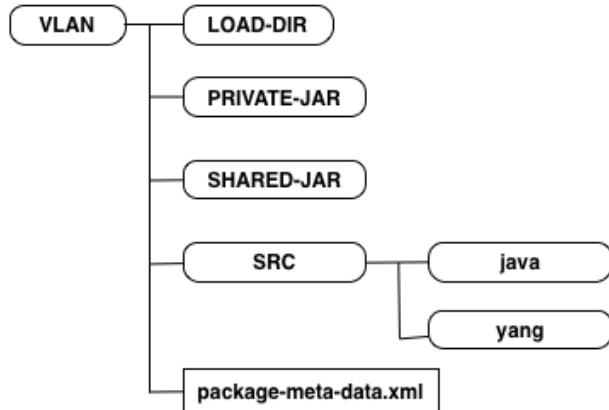
Note the packages directory, **cd** to it:

```
$ cd packages
$ ls -l
total 8
cisco-ios -> ../../packages/neds/cisco-ios
```

Currently there is only one package, the Cisco IOS NED. We will now create a new package that will contain the VLAN service.

```
$ ncs-make-package --service-skeleton java vlan
$ ls
cisco-ios vlan
```

This creates a package with the following structure:

Figure 137. Package Structure

During the rest of this section we will work with the `vlan/src/yang/vlan.yang` and `vlan/src/java/src/com/example/vlan/vlanRFS.java` files.

The Service Model

Edit the `vlan/src/yang/vlan.yang` according to below:

```

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

        uses ncs:service-data;
        ncs:servicepoint "vlan-servicepoint";
        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 string;
            }
        }
    }
}
  
```

This simple VLAN service model says:

- 1 We give a VLAN a name, for example `net-1`

- 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 name is just a string. A more correct and useful example would specify this is a reference to an interface to the device, but for now it is better to keep the example simple.

Make sure you do keep the lines generated by the **ncs-make-package**:

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

The first line expands to a YANG structure that is shared amongst all services. The second line connects the service to the Java callback.

To build this service model **cd** to `packages/vlan/src` and type **make** (assuming you have the make build system installed).

```
$ cd packages/vlan/src/
$ make
```

We can now test the service model by requesting NSO to reload all packages:

```
$ ncs_cli -C -U admin
admin@ncs# packages reload
>>> System upgrade is starting.
>>> Sessions in configure mode must exit to operational mode.
>>> No configuration changes can be performed until upgrade has completed.
>>> System upgrade has completed successfully.
result Done
```

You can also stop and start NSO, but then you have to pass the option **--with-package-reload** when starting NSO. This is important, NSO does not by default take any changes in packages into account when restarting. When packages are reloaded the `state/packages-in-use` is updated.

Now, create a VLAN service, (nothing will happen since we have not defined any mapping).

```
admin@ncs(config)# services vlan net-0 vlan-id 1234 device-if c0 interface 1/0
admin@ncs(config-device-if-c0)# top
admin@ncs(config)# commit
```

Ok, that worked let us move on and connect that to some device configuration using Java mapping. Note well that Java mapping is not needed, templates are more straight-forward and recommended but we use this as an "Hello World" introduction to Java Service Programming in NSO. Also at the end we will show how to combine Java and templates. Templates are used to define a vendor independent way of mapping service attributes to device configuration and Java is used as a thin layer before the templates to do logic, call-outs to external systems etc.

Managing the NSO Java VM

The default configuration of the Java VM is:

```
admin@ncs(config)# show full-configuration java-vm | details
java-vm stdout-capture enabled
java-vm stdout-capture file ./logs/ncs-java-vm.log
java-vm connect-time      60
java-vm initialization-time 60
java-vm synchronization-timeout-action log-stop
java-vm jmx jndi-address 127.0.0.1
java-vm jmx jndi-port 9902
java-vm jmx jmx-address 127.0.0.1
```

```
java-vm jmx jmx-port 9901
```

By default, ncs will start the Java VM invoking the command `$NCS_DIR/bin/ncs-start-java-vm`. That script will invoke

```
$ java com.tailf.ncs.NcsJVMLauncher
```

The class `NcsJVMLauncher` contains the `main()` method. The started java vm will automatically retrieve and deploy all java code for the packages defined in the load-path of the `ncs.conf` file. No other specification than the `package-meta-data.xml` for each package is needed.

The verbosity of Java error messages can be controlled by:

```
admin@ncs(config)# java-vm exception-error-message verbosity
Possible completions:
    standard  trace  verbose
```

For more detail on the Java VM settings see [Chapter 5, The NSO Java VM](#).

A first look at Java Development

The service model and the corresponding Java callback is bound by the service point name. Look at the service model in `packages/vlan/src/yang`:

Figure 138. VLAN Service model service-point

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

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

The corresponding generated Java skeleton, (one print hello world statement added):

Figure 139. Java Service Create Callback

```
@ServiceCallback(servicePoint="vlan-servicepoint",
    callType=ServiceCType.CREATE)
public Properties create(ServiceContext context,
    NavuNode service,
    NavuNode ncsRoot,
    Properties opaque)
    throws DpCallbackException {
    System.out.println("Hello World!");

    try {
        // check if it is reasonable to assume that devices
        // initially has been sync-from:ed
        NavuList managedDevices = ncsRoot.
            container("devices").list("device");
```

Modify the generated code to include the print "Hello World!" statement in the same way. Re-build the package:

```
$ cd packages/vlan/src/
$ make
```

Whenever a package has changed we need to tell NSO to reload the package. There are three ways:

- 1 Just reload the implementation of a specific package, will not load any model changes: **admin@ncs# packages package vlan redeploy**
- 2 Reload all packages including any model changes: **admin@ncs# packages reload**
- 3 Restart NSO with reload option: **\$ncs --with-package-reload**

When that is done we can create a service (or modify an existing) and the callback will be triggered:

```
admin@ncs(config)# vlan net-0 vlan-id 888
admin@ncs(config-vlan-net-0)# commit
```

Now, have a look in the logs/ncs-java-vm.log:

```
$ tail ncs-java-vm.log
...
<INFO> 03-Mar-2014::16:55:23.705 NcsMain JVM-Launcher: \
    - REDEPLOY PACKAGE COLLECTION --> OK
<INFO> 03-Mar-2014::16:55:23.705 NcsMain JVM-Launcher: \
    - REDEPLOY ["vlan"] --> DONE
<INFO> 03-Mar-2014::16:55:23.706 NcsMain JVM-Launcher: \
    - DONE COMMAND --> REDEPLOY_PACKAGE
<INFO> 03-Mar-2014::16:55:23.706 NcsMain JVM-Launcher: \
    - READ SOCKET =>
Hello World!
```

Tailing the ncs-java-vm.log is one way of developing. You can also start and stop the Java VM explicitly and see the trace in the shell. First of all tell NSO not to start the VM by adding the following snippet to ncs.conf:

```
<java-vm>
    <auto-start>false</auto-start>
</java-vm>
```

Then, after restarting NSO or reloading the configuration, from the shell prompt:

```
$ ncs-start-java-vm
...
.. all stdout from JVM
```

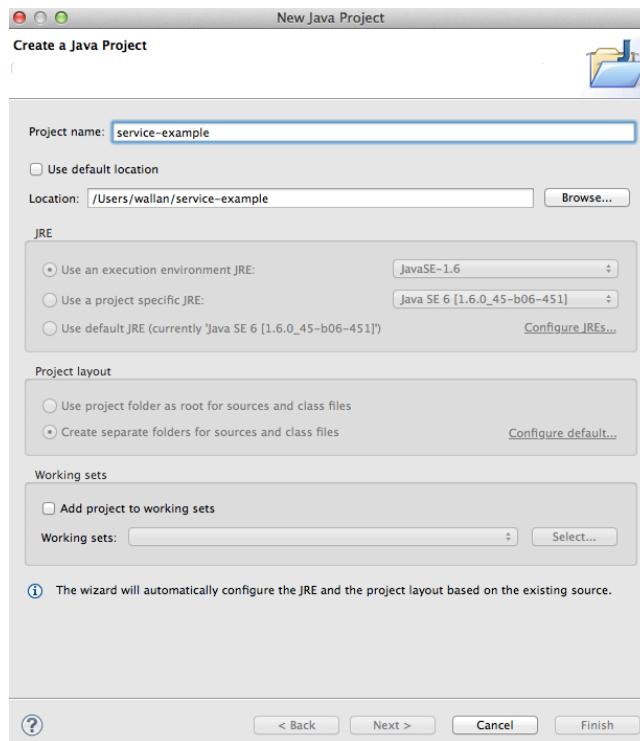
So modifying or creating a VLAN service will now have the "Hello World!" string show up in the shell. You can modify the package and reload/redeploy and see the output.

Using Eclipse

First of all generate environment for Eclipse:

```
$ ncs-setup --eclipse-setup
```

This will generate two files, .classpath and .project. If we add this directory to eclipse as a "File->New->Java Project", uncheck the "Use the default location" and enter the directory where the .classpath and .project have been generated. We're immediately ready to run this code in eclipse.

Figure 140. Creating the project in Eclipse

All we need to do is to choose the `main()` routine in the `NcsJVMLauncher` class. The eclipse debugger works now as usual, and we can at will start and stop the Java code.

One caveat here which is worth mentioning is that there are a few timeouts between NSO and the Java code that will trigger when we sit in the debugger. While developing with the eclipse debugger and breakpoints we typically want to disable all these timeouts. First we have 3 timeouts in `ncs.conf` that matter. Set the three values of `/ncs-config/japi/new-session-timeout` /`ncs-config/japi/query-timeout` /`ncs-config/japi/connect-timeout` to a large value. See man page `ncs.conf(5)` for a detailed description on what those values are. If these timeouts are triggered, NSO will close all sockets to the Java VM and all bets are off.

```
$ cp $NCS_DIR/etc/ncs/ncs.conf .
```

Edit the file and enter the following XML entry just after the Webui entry.

```
<japi>
    <new-session-timeout>PT1000S</new-session-timeout>
    <query-timeout>PT1000S</query-timeout>
    <connect-timeout>PT1000S</connect-timeout>
</japi>
```

Now restart ncs, and from now on start it as

```
$ ncs -c ./ncs.conf
```

You can verify that the Java VM is not running by checking the package status:

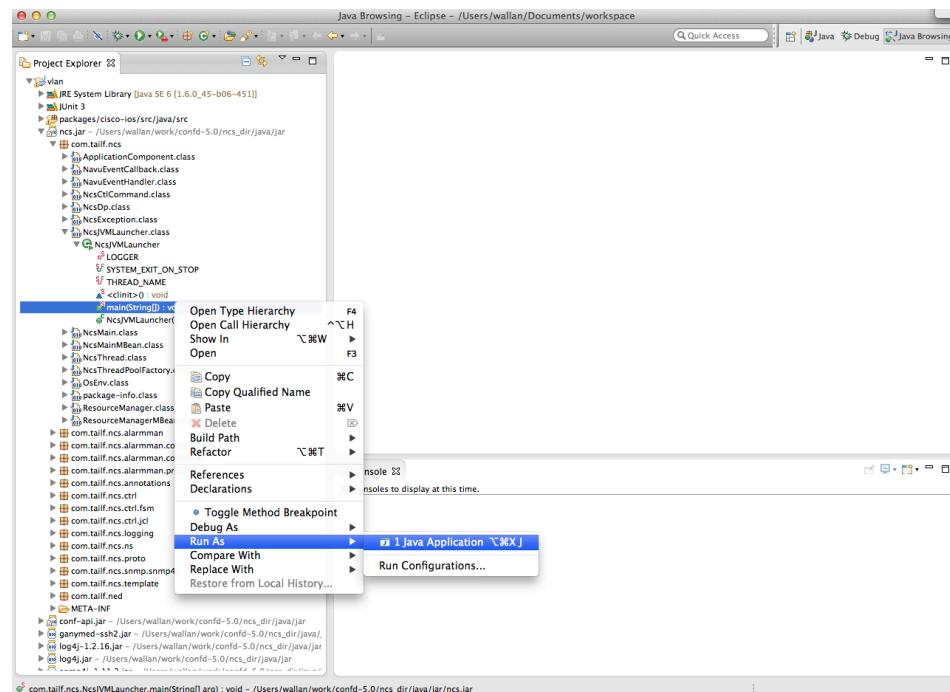
```
admin@ncs# show packages package vlan
packages package vlan
```

```

package-version 1.0
description      "Skeleton for a resource facing service - RFS"
ncs-min-version 3.0
directory        ./state/packages-in-use/1/vlan
component RFSSkeleton
callback java-class-name [ com.example.vlan.vlanRFS ]
oper-status java-uninitialized
  
```

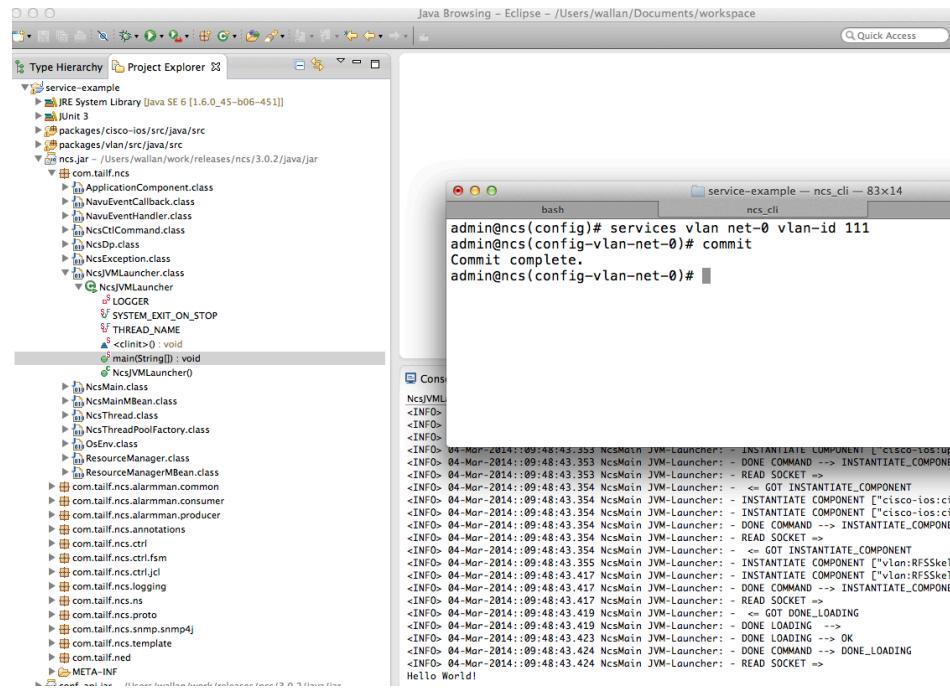
Create a new project and start the launcher main in Eclipse:

Figure 141. Starting the NSO JVM from Eclipse



You can start and stop the Java VM from Eclipse. Note well that this is not needed since the change cycle is: modify the Java code, make in the src directory and then reload the package. All while NSO and the JVM is running. Change the VLAN service and see the console output in Eclipse:

Figure 142. Console output in Eclipse



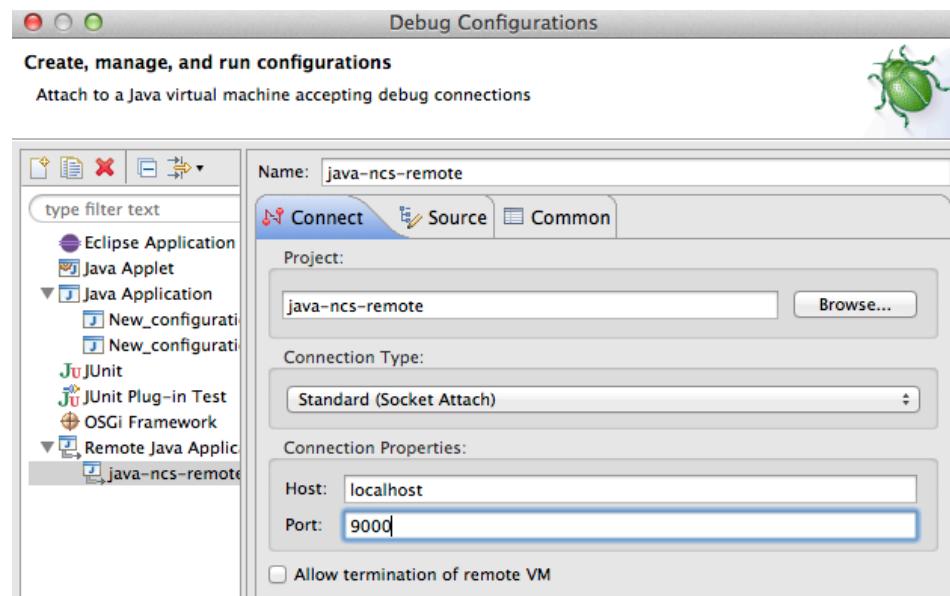
Another option is to have Eclipse connect to the running VM. Start the VM manually with the **-d** option.

```
$ ncs-start-java-vm -d
Listening for transport dt_socket at address: 9000
NCS JVM STARTING
...

```

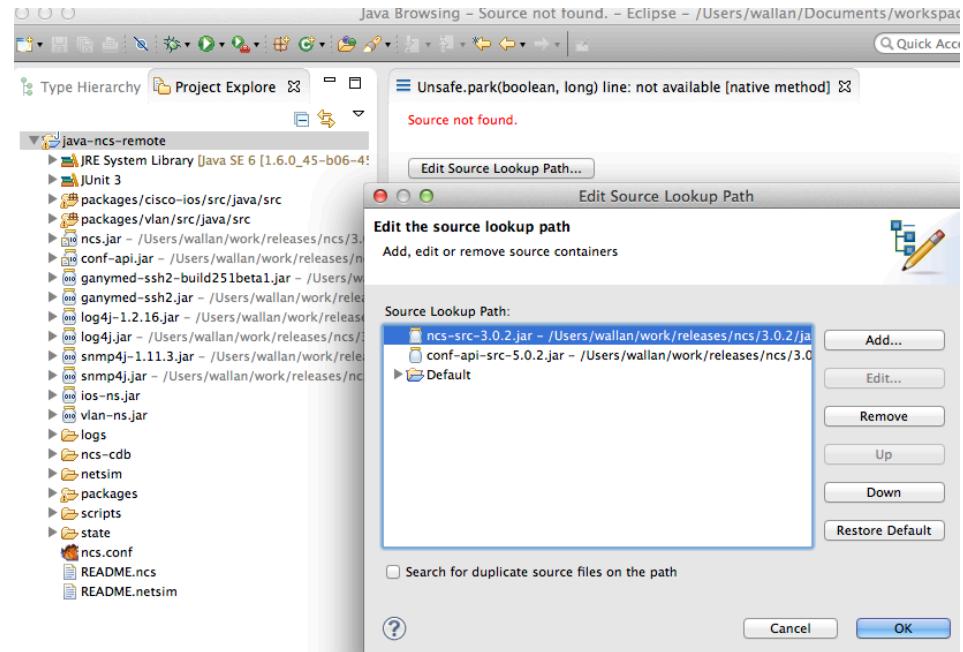
Then you can setup Eclipse to connect to the NSO Java VM:

Figure 143. Connecting to NSO Java VM Remote with Eclipse



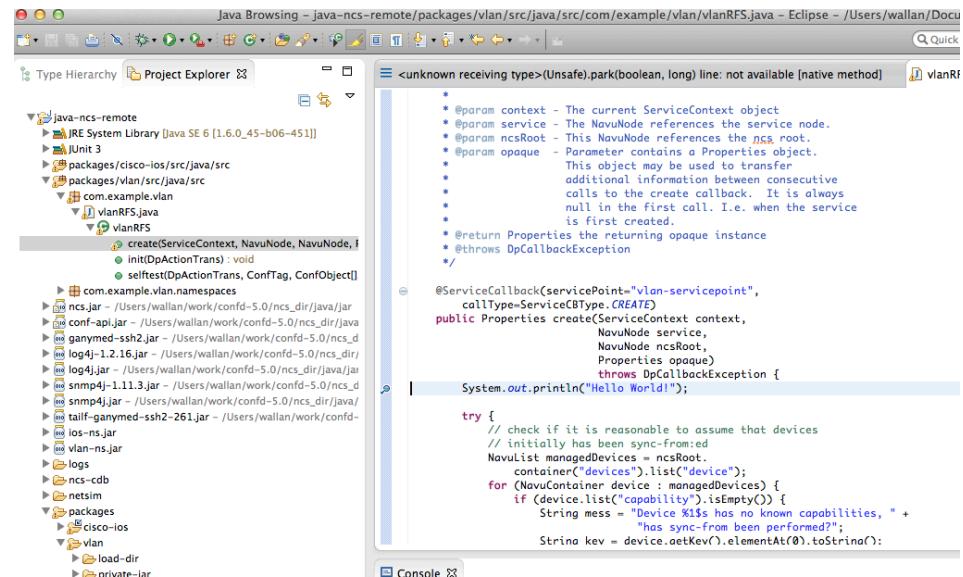
In order for Eclipse to show the NSO code when debugging add the NSO Source Jars, (add external Jar in Eclipse):

Figure 144. Adding the NSO source Jars



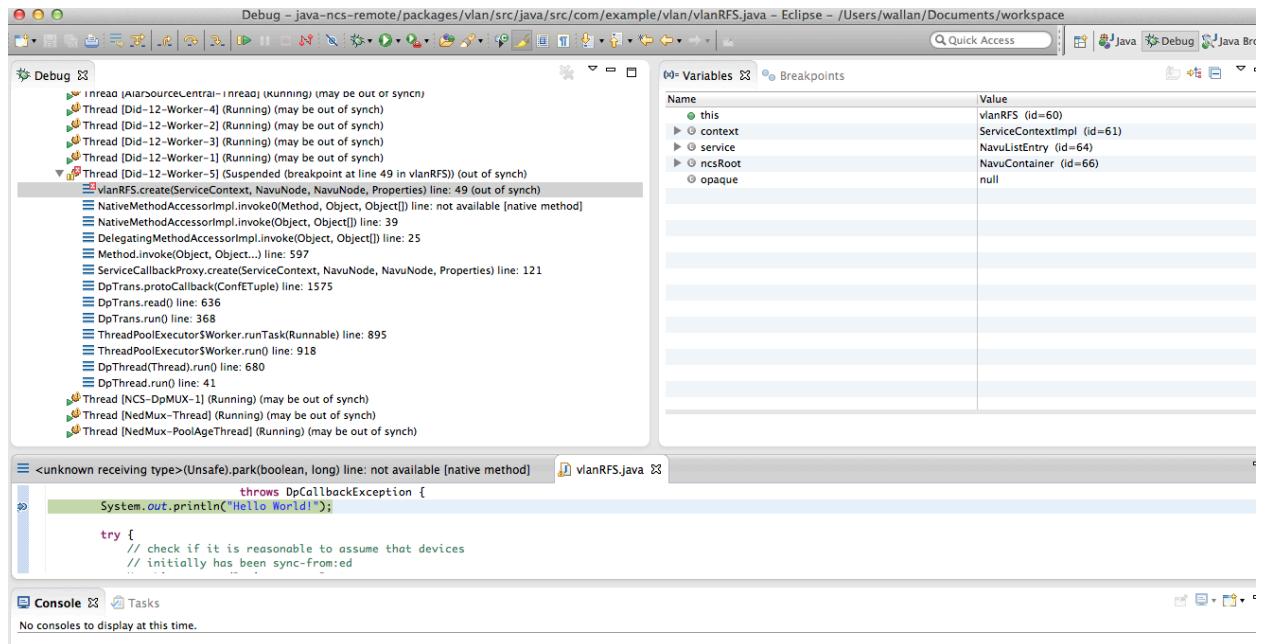
Navigate to the service create for the VLAN service and add a breakpoint:

Figure 145. Setting a break-point in Eclipse



Commit a change of a VLAN service instance and Eclipse will stop at the breakpoint:

Figure 146. Service Create breakpoint



Writing the service code

Fetching the service attributes

So the problem at hand is that we have service parameters and a resulting device configuration. Previously in this user guide we showed how to do that with templates. The same principles apply in Java. The service model and the device models are YANG models in NSO irrespective of the underlying protocol. The Java mapping code transforms the service attributes to the corresponding configuration leafs in the device model.

The NAVU API lets the Java programmer navigate the service model and the device models as a DOM tree. Have a look at the create signature:

```
@ServiceCallback(servicePoint="vlan-servicepoint",
    callType=ServiceCBType.CREATE)
public Properties create(ServiceContext context,
    NavuNode service,
    NavuNode ncsRoot,
    Properties opaque)
throws DpCallbackException {
```

Two NAVU nodes are passed: the actual service serviceinstance and the NSO root ncsRoot.

We can have a first look at NAVU by analyzing the first try statement:

```
try {
    // check if it is reasonable to assume that devices
    // initially has been sync-ed
    NavuList managedDevices =
        ncsRoot.container("devices").list("device");
    for (NavuContainer device : managedDevices) {
        if (device.list("capability").isEmpty()) {
            String mess = "Device %1$s has no known capabilities, " +
```

```
        "has sync-from been performed?" ;  
    String key = device.getKey().elementAt(0).toString();  
    throw new DpCallbackException(String.format(mess, key));  
}  
}
```

NAVU is a lazy evaluated DOM tree that represents the instantiated YANG model. So knowing the NSO model: `devices/device`, (`container/list`) corresponds to the list of capabilities for a device, this can be retrieved by `ncsRoot.container("devices").list("device")`.

The service node can be used to fetch the values of the VLAN service instance:

- vlan/name
 - vlan/vlan-id
 - vlan/device-if/device and vlan/device-if/interface

A first snippet that iterates the service model and prints to the console looks like below:

Figure 147. The first example

```
String vlanName = service.leaf("name").valueAsString();
NavuLeaf vlanIDLeaf = service.leaf("vlan-id");
ConfUInt32 vlanID = (ConfUInt32)vlanIDLeaf.value();

System.out.println("VLAN name: " + vlanName);
System.out.println("VLAN ID: " + vlanID);

NavuList interfaces = service.list("device-if");
for(NavuContainer intf: interfaces.elements()){
    System.out.println("Device: " + intf.leaf("device-name").valueAsString());
    System.out.println("Fast Ethernet if: " + intf.leaf("interface").valueAsString());
}

}
```

The `com.tailf.conf` package contains Java Classes representing the YANG types like `ConfUInt32`.

Try it out by the following sequence:

- 1 Rebuild the Java Code : in `packages/vlan/src` type `make`.
 - 2 Reload the package : in the NSO Cisco CLI do `admin@ncs# packages package vlan redeploy`.
 - 3 Create or modify a vlan service: in NSO CLI `admin@ncs(config)# services vlan net-0 vlan-id 844 device-if c0 interface 1/0`, and `commit`.

Mapping service attributes to device configuration

Figure 148. Fetching values from the service instance

```
// Fetch the VLAN ID
String vlanString = service.leaf("vlan-id").valueAsString();
ConfUInt32 vlanID = (ConfUInt32)service.leaf(vlan._vlan_id).value();
ConfUInt16 vlanID16 = new ConfUInt16(vlanID.longValue());
```

Remember the `service` attribute is passed as a parameter to the `create` method. As a starting point, look at the first three lines:

- 1 To reach a specific leaf in the model use the NAVU leaf method with the name of the leaf as parameter. This leaf then has various methods like getting the value as a string.

- 2 `service.leaf("vlan-id")` and `service.leaf(vlan._vlan_id_)` are two ways of referring to the `vlan-id` leaf of the service. The latter alternative uses symbols generated by the compilation steps. If this alternative is used, you get the benefit of compilation time checking. From this leaf you can get the value according to the type in the YANG model `ConfUINt32` in this case.
- 3 Line 3 shows an example of casting between types. In this case we prepare the VLAN ID as a 16 unsigned int for later use.

Next step is to iterate over the devices and interfaces. The `NAVU elements()` returns the elements of a NAVU list.

Figure 149. Iterating a list in the service model

```
// Get the device and interface list
// device-if
//   ---device-name
//   ---interface
NavuList interfaces = service.list("device-if");
for(NAVUContainer intf: interfaces.elements()){
    System.out.println("Device: " + intf.leaf("device-name").valueAsString());
    System.out.println("Fast Ethernet if: " + intf.leaf("interface").valueAsString());
}
```

In order to write the mapping code, make sure you have an understanding of the device model. One good way of doing that is to create a corresponding configuration on one device and then display that with pipe target "display xpath". Below is a CLI output that shows the model paths for "FastEthernet 1/0":

```
admin@ncs% show devices device c0 config ios:interface
                    FastEthernet 1/0 | display xpath

/devices/device[name='c0']/config/ios:interface/
                    FastEthernet[name='1/0']/switchport	mode/trunk

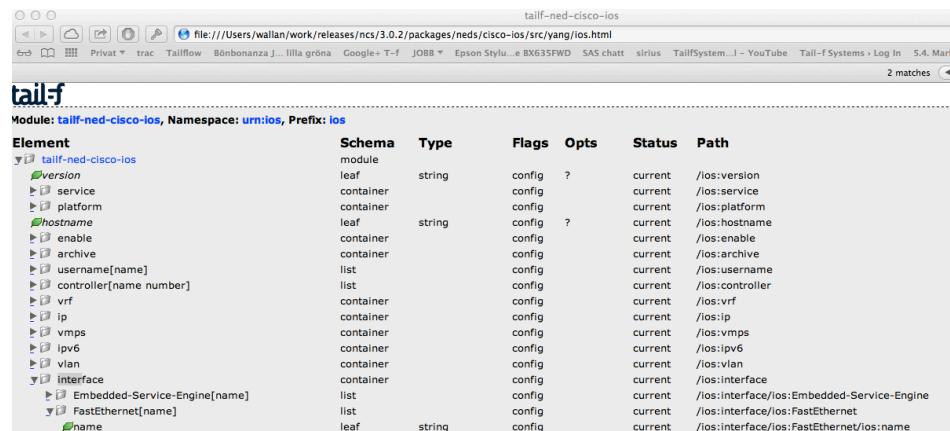
/devices/device[name='c0']/config/ios:interface/
                    FastEthernet[name='1/0']/switchport/trunk/allowed/vlan/vlans [ 111 ]
```

Another useful tool is to render a tree view of the model:

```
$ pyang -f jstree tailf-ned-cisco-ios.yang -o ios.html
```

This can then be opened in a Web browser and model paths are shown to the right:

Figure 150. The Cisco IOS Model



Now, we replace the print statements with setting real configuration on the devices.

Figure 151. Setting the VLAN list

```
// Get the device and interface list
// device-if
//   !---device-name
//   !---interface
NavuList interfaces = service.list("device-if");
for(NavuContainer intf: interfaces.elements()){
    NavuLeaf deviceLeaf = intf.leaf("device-name");
    String feIntfName = intf.leaf("interface").valueAsString();

    // The device
    NavuContainer device = (NavuContainer)deviceLeaf.deref().get(0).getParent();
    // Set the VLAN list
    device.container("config").container("ios", "vlan").list("vlan-list").sharedCreate(vlanString);
    // Get the interfaces for the device
    NavuList feIntfList = device.container("config").container("ios", "interface").list("FastEthernet");
    // Do we have the interface given by the service?
    try {
        if (!feIntfList.containsNode(feIntfName) ) {
            throw new DpCallbackException("Can not find FastEthernet interface: " + feIntfName);
        }
    }
}
```

Let us walk through the above code line by line. The `device-name` is a `leafref`. The `deref` method returns the object that the `leafref` refers to. The `getParent()` might surprise the reader. Look at the path for a `leafref`: `/device/name/config/ios:interface/name`. The name `leafref` is the key that identifies a specific interface. The `deref` returns that key, while we want to have a reference to the interface, `(/device/name/config/ios:interface)`, that is the reason for the `getParent()`.

The next line sets the `vlan-list` on the device. Note well that this follows the paths displayed earlier using the NSO CLI. The `sharedCreate()` is important, it creates device configuration based on this service, and it says that other services might also create the same value, "shared". Shared create maintains reference counters for the created configuration in order for the service deletion to delete the configuration only when the last service is deleted. Finally the interface name is used as a key to see if the interface exists, "`containsNode()`".

The last step is to update the VLAN list for each interface. The code below adds an element to the VLAN `leaf-list`.

```
// The interface
NavuNode theIf = feIntfList.elem(feIntfName);
theIf.container("switchport").
    sharedCreate().
    container("mode").
    container("trunk").
    sharedCreate();
// Create the VLAN leaf-list element
theIf.container("switchport").
    container("trunk").
    container("allowed").
    container("vlan").
    leafList("vlans").
    sharedCreate(vlanID16);
```

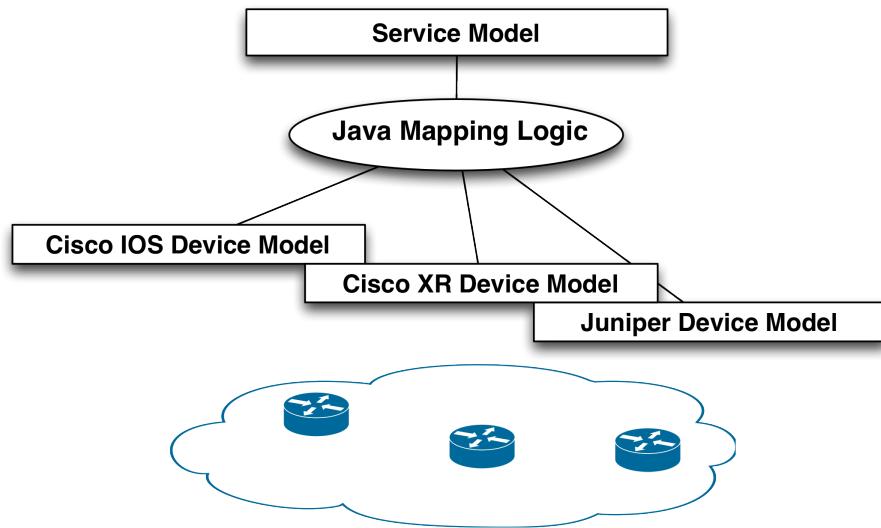
The above create method is all that is needed for create, read, update and delete. NSO will automatically handle any changes, like changing the VLAN ID, adding an interface to the VLAN service and deleting the service. Play with the CLI and modify and delete VLAN services and make sure you realize this. This is handled by the FASTMAP engine, it renders any change based on the single definition of the create method.

Mapping using Java combined with Templates

Overview

We have shown two ways of mapping a service model to device configurations, service templates and Java. The mapping strategy using only Java is illustrated in the Figure below.

Figure 152. Flat mapping with Java

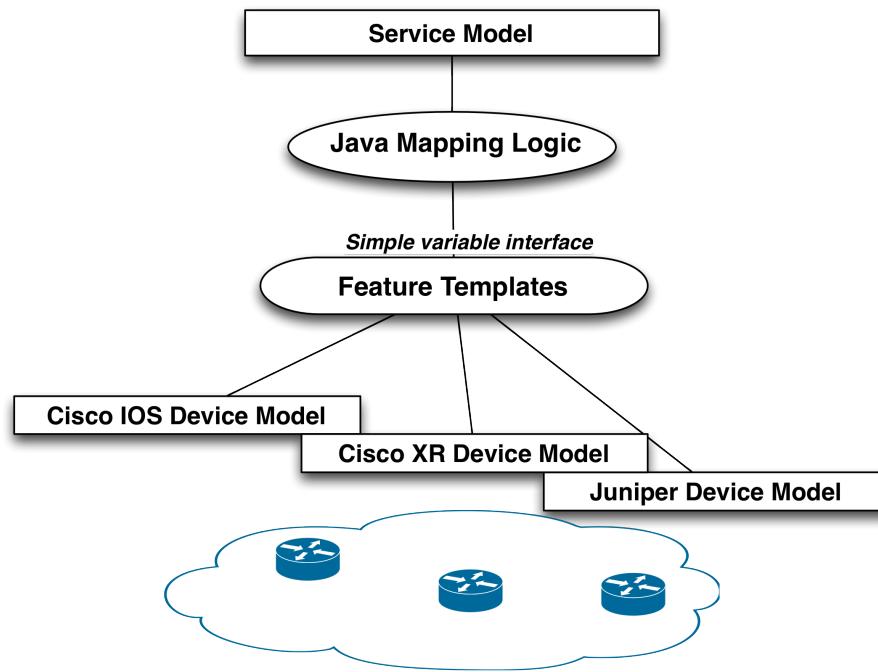


This strategy has some drawbacks:

- Managing different device vendors. If we would introduce more vendors in the network this would need to be handled by the Java code. Of course this can be factored into separate classes in order to keep the general logic clean and just passing the device details to specific vendor classes, but this gets complex and will always require Java programmers for introducing new device types.
- No clear separation of concerns, domain expertise. The general business logic for a service is one thing, detailed configuration knowledge of device types something else. The latter requires network engineers and the first category is normally separated into a separate team that deals with OSS integration.

Java and templates can be combined according to below:

Figure 153. Two layered mapping using feature templates



In this model the Java layer focus on required logic, but it never touches concrete device models from various vendors. The vendor specific details are abstracted away using feature templates. The templates takes variables as input from the service logic, and the templates in turn transforms these into concrete device configuration. Introducing of a new device type does not affect the Java mapping.

This approach has several benefits:

- The service logic can be developed independently of device types.
- New device types can be introduced at runtime without affecting service logic.
- Separation of concerns: network engineers are comfortable with templates, they look like a configuration snippet. They have the expertise how configuration is applied to real devices. People defining the service logic often are more programmers, they need to interface with other systems etc, this suites a Java layer.

Note that the logic layer does not understand the device types, the templates will dynamically apply the correct leg of the template depending on which device is touched.

The VLAN Feature Template

From an abstraction point of view we want a template that takes the following variables:

- VLAN id
- Device and interface

So the mapping logic can just pass these variables to the feature template and it will apply it to a multi-vendor network.

Create a template as described before.

- Create a concrete configuration on a device, or several devices of different type
- Request NSO to display that as XML
- Replace values with variables

This results in a feature template like below:

```
<!-- Feature Parameters -->
<!-- $DEVICE -->
<!-- $VLAN_ID -->
<!-- $INTF_NAME -->

<config-template xmlns="http://tail-f.com/ns/config/1.0"
                  servicepoint="vlan">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>{$DEVICE}</name>
      <config>
        <vlan xmlns="urn:ios" tags="merge">
          <vlan-list>
            <id>{$VLAN_ID}</id>
          </vlan-list>
        </vlan>
        <interface xmlns="urn:ios" tags="merge">
          <FastEthernet tags="nocreate">
            <name>{$INTF_NAME}</name>
            <switchport>
              <trunk>
                <allowed>
                  <vlan tags="merge">
                    <vlans>{$VLAN_ID}</vlans>
                  </vlan>
                </allowed>
              </trunk>
            </switchport>
          </FastEthernet>
        </interface>
      </config>
    </device>
  </devices>
</config-template>
```

This template only maps to Cisco IOS devices (the xmlns="urn:ios" namespace), but you can add "legs" for other device types at any point in time and reload the package.



Note

Nodes set with a template variable evaluating to the empty string are ignored, e.g., the setting <some-tag>{\$VAR}</some-tag> is ignored if the template variable \$VAR evaluates to the empty string. However, this does not apply to XPath expressions evaluating to the empty string. A template variable can be surrounded by the XPath function string() if it is desirable to set a node to the empty string.

The VLAN Java Logic

The Java mapping logic for applying the template is shown below:

Figure 154. Mapping logic using template

```

Template vlanTemplate = new Template(context, "vlan-template");
String vlanString = service.leaf("vlan-id").valueAsString();
NavuList interfaces = service.list("device-if");
for(NavuContainer intf: interfaces.elements()){
    String deviceNameString = intf.leaf("device-name").valueAsString();
    String ifNameString = intf.leaf("interface").valueAsString();

    TemplateVariables vlanVar = new TemplateVariables();
    vlanVar.putQuoted("VLAN_ID",vlanString);
    vlanVar.putQuoted("DEVICE",deviceNameString);
    vlanVar.putQuoted("INTF_NAME",ifNameString);
    vlanTemplate.apply(service, vlanVar);
}

```

Note that the Java code has no clue about the underlying device type, it just passes the feature variables to the template. At run-time you can update the template with mapping to other device types. The Java-code stays untouched, if you modify an existing VLAN service instance to refer to the new device type the commit will generate the corresponding configuration for that device.

The smart reader will complain, "why do we have the Java layer at all?", this could have been done as a pure template solution. That is true, but now this simple Java layer gives room for arbitrary complex service logic before applying the template.

Steps to Build a Java and Template Solution

The steps to build the solution described in this section are:

- Step 1** Create a run-time directory: `$ mkdir ~/service-template; cd ~/service-template`
- Step 2** Generate a netsim environment: `$ ncs-netsim create-network $NCS_DIR/packages/neds/cisco-ios 3 c`
- Step 3** Generate the NSO runtime environment: `$ ncs-setup --netsim-dir ./netsim --dest ./`
- Step 4** Create the VLAN package in the packages directory: `$ cd packages; ncs-make-package --service-skeleton java vlan`
- Step 5** Create a template directory in the VLAN package: `$ cd vlan; mkdir templates`
- Step 6** Save the above described template in `packages/vlan/templates`
- Step 7** Create the YANG service model according to above: `packages/vlan/src/yang/vlan.yang`
- Step 8** Update the Java code according to above: `packages/vlan/src/java/src/com/example/vlan/vlanRFS.java`
- Step 9** Build the package: in `packages/vlan/src` do `make`
- Step 10** Start NSO

Service Mapping: Putting Things Together

The purpose of this section is to show a more complete example of a service mapping. It is based on the example `examples.ncs/service-provider/mpls-vpn`.

Auxiliary Service Data

In the previous sections we have looked at service mapping when the input parameters are enough to generate the corresponding device configurations. In many cases this is not the case. The service mapping

logic may need to reach out to other data in order to generate the device configuration. This is common in the following scenarios:

- Policies: it might make sense to define policies that can be shared between service instances. The policies, for example QoS, have data models of their own (not service models) and the mapping code reads from that.
- Topology information: the service mapping might need to know connected devices, like which PE the CE is connected to.
- Resources like VLAN IDs, IP addresses: these might not be given as input parameters. This can be modeled separately in NSO or fetched from an external system.

It is important to design the service model to consider the above examples: what is input? what is available from other sources? This example illustrates how to define QoS policies "on the side". A reference to an existing QoS policy is passed as input. This is a much better principle than giving all QoS parameters to every service instance. Note well that if you modify the QoS definitions that services are referring to, this will not change the existing services. In order to have the service to read the changed policies you need to perform a **re-deploy** on the service.

This example also uses a list that maps every CE to a PE. This list needs to be populated before any service is created. The service model only has the CE as input parameter, and the service mapping code performs a lookup in this list to get the PE. If the underlying topology changes a service re-deploy will adopt the service to the changed CE-PE links. See more on topology below.

NSO has a package to manage resources like VLAN and IP addresses as a pool within NSO. In this way the resources are managed within the transaction. The mapping code could also reach out externally to get resources. The Reactive FASTMAP pattern is recommended for this.

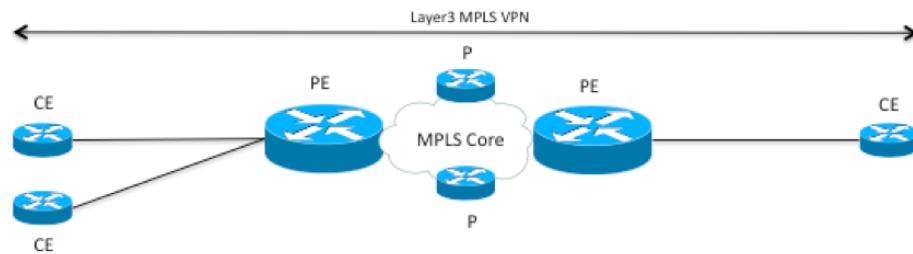
Topology

Using topology information in the instantiation of a NSO service is a common approach, but also an area with many misconceptions. Just like a service in NSO takes a black-box view of the configuration needed for that service in the network NSO treats topologies in the same way. It is of course common that you need to reference topology information in the service but it is highly desirable to have a decoupled and self-sufficient service that only uses the part of the topology that is interesting/needed for the specific service should be used.

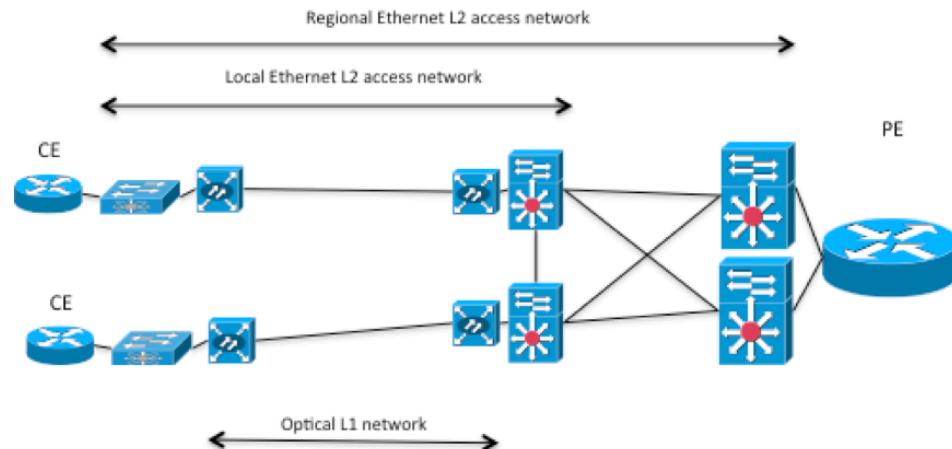
Other parts of the topology could either be handled by other services or just let the network state sort it out, it does not necessarily relate to configuration the network. A routing protocol will for example handle the IP path through the network.

It is highly desirable to not introduce unneeded dependencies towards network topologies in your service.

To illustrate this, lets look at a Layer3 MPLS VPN service. A logical overview of an MPLS VPN with three endpoints could look something like this. CE routers connecting to PE routers, that are connected to an MPLS core network. In the MPLS core network there are a number of P routers.

Figure 155. Simple MPLS VPN Topology

In the service model you only want to configure the CE devices to use as endpoints. In this case topology information could be used to sort out what PE router each CE router is connected to. However what type of topology do you need. Lets look at a more detailed picture of what the L1 and L2 topology could look like for one side of the picture above.

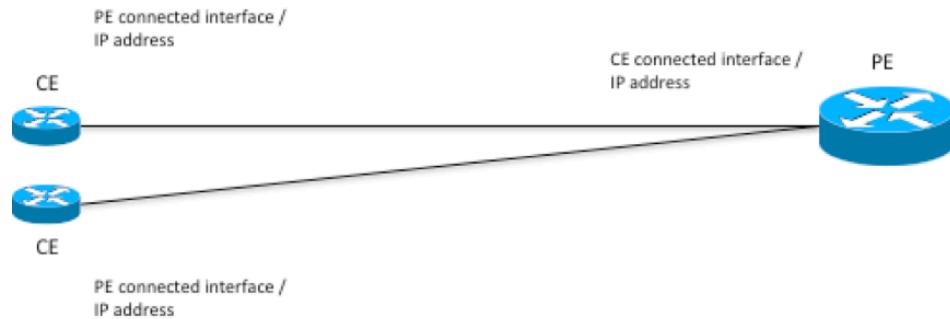
Figure 156. L1-L2 Topology

In pretty much all networks there is an access network between the CE and PE router. In the picture above the CE routers are connected to local Ethernet switches connected to a local Ethernet access network, connected through optical equipment. The local Ethernet access network is connected to a regional Ethernet access network, connected to the PE router. Most likely the physical connections between the devices in this picture has been simplified, in the real world redundant cabling would be used. The example above is of course only one example of how an access network could look like and it is very likely that a service provider have different access technologies. For example Ethernet, ATM, or a DSL based access network.

Depending on how you design the L3VPN service, the physical cabling or the exact traffic path taken in the layer 2 Ethernet access network might not be that interesting, just like we don't make any assumptions or care about how traffic is transported over the MPLS core network. In both these cases we trust the underlying protocols handling state in the network, spanning tree in the Ethernet access network, and routing protocols like BGP in the MPLS cloud. Instead in this case it could make more sense to have a separate NSO service for the access network, both so it can be reused for both for example L3VPN's and L2VPN's but also to not tightly couple to the access network with the L3VPN service since it can be different (Ethernet or ATM etc.).

Looking at the topology again from the L3VPN service perspective, if services assume that the access network is already provisioned or taken care of by another service, it could look like this.

Figure 157. Black-box topology



The information needed to sort out what PE router a CE router is connected to as well as configuring both CE and PE routers is:

- Interface on the CE router that is connected to the PE router, and IP address of that interface.
- Interface on the PE router that is connected to the CE router, and IP address to the interface.

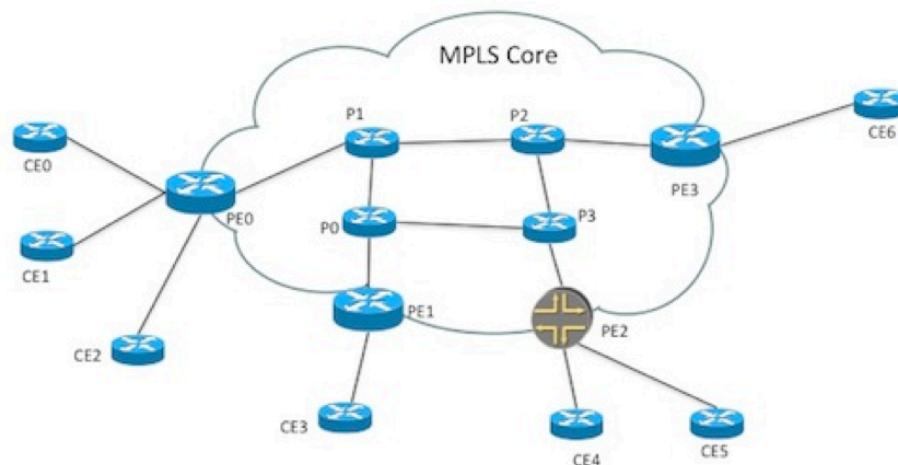
Creating a Multi-Vendor Service

This section describes the creation of a MPLS L3VPN service in a multi vendor environment applying the concepts described above. The example discussed can be found in `examples.ncs/service-provider/mpls-vpn`. The example network consists of Cisco ASR 9k and Juniper core routers (P and PE) and Cisco IOS based CE routers.

The goal with the NSO service is to setup a MPLS Layer3 VPN on a number of CE router endpoints using BGP as the CE-PE routing protocol. Connectivity between the CE and PE routers is done through a Layer2 Ethernet access network, which is out of scope for this service. In a real world scenario the access network could for example be handled by another service.

In the example network we can also assume that the MPLS core network already exists and is configured.

Figure 158. The MPLS VPN Example



YANG Service Model Design

When designing service YANG models there are a number of things to take into consideration. The process usually involves the following steps:

-
- Step 1** Identify the resulting device configurations for a deployed service instance.
 - Step 2** Identify what parameters from the device configurations that are common and should be put in the service model.
 - Step 3** Ensure that the scope of the service and the structure of the model works with the NSO architecture and service mapping concepts. For example, avoid unnecessary complexities in the code to work with the service parameters.
 - Step 4** Ensure that the model is structured in a way so that integration with other systems north of NSO works well. For example, ensure that the parameters in the service model map to the needed parameters from an ordering system.
-

Step 1 and 2: Device Configurations and Identifying parameters

Deploying a MPLS VPN in the network results in the following basic CE and PE configurations. The snippets below only include the Cisco IOS and Cisco IOS-XR configurations. In a real process all applicable device vendor configurations should be analyzed.

Example 159. CE Router Config

```
interface GigabitEthernet0/1.77
description Link to PE / pe0 - GigabitEthernet0/0/0/3
encapsulation dot1Q 77
ip address 192.168.1.5 255.255.255.252
service-policy output volvo
!
policy-map volvo
  class class-default
    shape average 6000000
  !
!
interface GigabitEthernet0/11
description volvo local network
ip address 10.7.7.1 255.255.255.0
exit
router bgp 65101
neighbor 192.168.1.6 remote-as 100
neighbor 192.168.1.6 activate
network 10.7.7.0
!
```

Example 160. PE Router Config

```
vrf volvo
address-family ipv4 unicast
import route-target
  65101:1
exit
export route-target
  65101:1
exit
exit
policy-map volvo-ce1
```

```

class class-default
    shape average 6000000 bps
!
end-policy-map
!
interface GigabitEthernet 0/0/0/3.77
    description Link to CE / cel - GigabitEthernet0/1
    ipv4 address 192.168.1.6 255.255.255.252
    service-policy output volvo-ce1
    vrf          volvo
    encapsulation dot1q 77
exit
router bgp 100
vrf volvo
rd 65101:1
address-family ipv4 unicast
exit
neighbor 192.168.1.5
remote-as 65101
address-family ipv4 unicast
as-override
exit
exit
exit
exit

```

The device configuration parameters that need to be uniquely configured for each VPN have been marked in bold.

Step 3 and 4: Model Structure and Integration with other Systems

When configuring a new MPLS l3vpn in the network we will have to configure all CE routers that should be interconnected by the VPN, as well as the PE routers they connect to.

However when creating a new l3vpn service instance in NSO it would be ideal if only the endpoints (CE routers) are needed as parameters to avoid having knowledge about PE routers in a northbound order management system. This means a way to use topology information is needed to derive or compute what PE router a CE router is connected to. This makes the input parameters for a new service instance very simple. It also makes the entire service very flexible, since we can move CE and PE routers around, without modifying the service configuration.

Resulting YANG Service Model:

```

container vpn {
    list l3vpn {
        tailf:info "Layer3 VPN";
        uses ncs:service-data;
        ncs:servicepoint l3vpn-servicepoint;

        key name;
        leaf name {
            tailf:info "Unique service id";
            type string;
        }
        leaf as-number {
            tailf:info "MPLS VPN AS number.";
            mandatory true;
            type uint32;
        }
    }
}

```

```
list endpoint {
    key id;
    leaf id {
        tailf:info "Endpoint identifier";
        type string;
    }
    leaf ce-device {
        mandatory true;
        type leafref {
            path "/ncs:devices/ncs:device/ncs:name";
        }
    }
    leaf ce-interface {
        mandatory true;
        type string;
    }
    leaf ip-network {
        tailf:info "private IP network";
        mandatory true;
        type inet:ip-prefix;
    }
    leaf bandwidth {
        tailf:info "Bandwidth in bps";
        mandatory true;
        type uint32;
    }
}
```

The snippet above contains the l3vpn service model. The structure of the model is very simple. Every VPN has a name, an as-number and a list of all the endpoints in the VPN. Each endpoint has:

- A unique id
 - A reference to a device (a CE router in our case)
 - A pointer to the LAN local interface on the CE router. This is kept as a string since we want this to work in a multi-vendor environment.
 - LAN private IP network
 - Bandwidth on the VPN connection.

To be able to derive the CE to PE connections we use a very simple topology model. Notice that this YANG snippet does not contain any servicepoint, which means that this is not a service model but rather just a YANG schema letting us store information in CDB.

```
container topology {
    list connection {
        key name;
        leaf name {
            type string;
        }
    }
    container endpoint-1 {
        tailf:cli-compact-syntax;
        uses connection-grouping;
    }
    container endpoint-2 {
        tailf:cli-compact-syntax;
        uses connection-grouping;
    }
}
```

```

        leaf link-vlan {
            type uint32;
        }
    }

grouping connection-grouping {
    leaf device {
        type leafref {
            path "/ncs:devices/ncs:device/ncs:name";
        }
    }
    leaf interface {
        type string;
    }
    leaf ip-address {
        type tailf:ipv4-address-and-prefix-length;
    }
}

```

The model basically contains a list of connections, where each connection points out the device, interface and ip-address in each of the connection.

Defining the Mapping

Since we need to lookup which PE routers to configure using the topology model in the mapping logic it is not possible to use a declarative configuration template based mapping. Using Java and configuration templates together is the right approach.

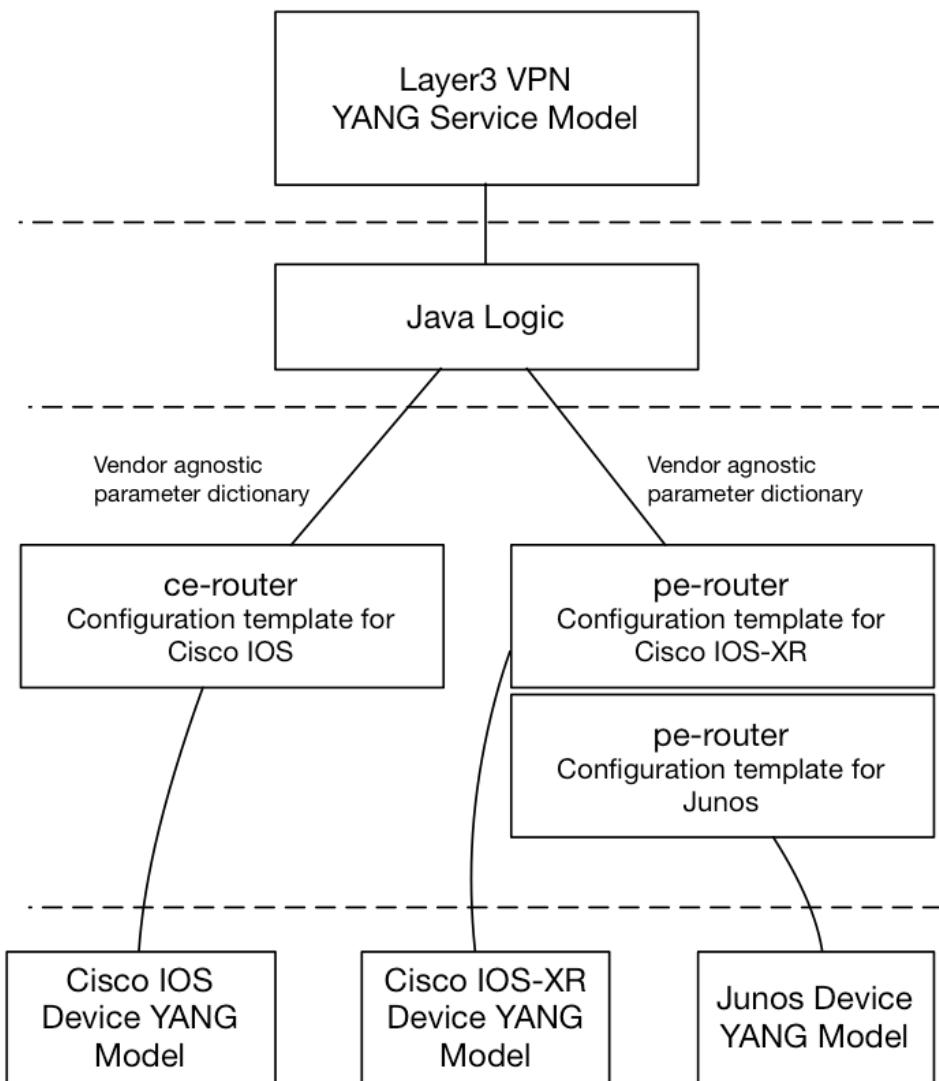
The Java logic lets you set a list of parameters that can be consumed by the configuration templates. One huge benefit of this approach is that all the parameters set in the Java code is completely vendor agnostic. When writing the code there is no need for knowledge of what kind of devices or vendors that exists in the network, thus creating an abstraction of vendor specific configuration. This also means that in to create the configuration template there is no need to have knowledge of the service logic in the Java code. The configuration template can instead be created and maintained by subject matter experts, the network engineers.

With this service mapping approach it makes sense to modularize the service mapping by creating configuration templates on a per feature level, creating an abstraction for a feature in the network. In this example means we will create the following templates:

- CE router
- PE router

This is both to make services easier to maintain and create but also to create components that are reusable from different services. This can of course be even more detailed with templates with for example BGP or interface configuration if needed.

Since the configuration templates are decoupled from the service logic it is also possible to create and add additional templates in a running NSO system. You can for example add a CE router from a new vendor to the layer3 VPN service by only creating a new configuration template, using the set of parameters from the service logic, to a running NSO system without changing anything in the other logical layers.

Figure 161. The MPLS VPN Example

The Java Code

The Java code part for the service mapping is very simple and follows the following pseudo code steps:

```

READ topology
FOR EACH endpoint
    USING topology
DERIVE connected-pe-router
    READ ce-pe-connection
    SET pe-parameters
    SET ce-parameters
    APPLY TEMPLATE l3vpn-ce
    APPLY TEMPLATE l3vpn-pe
  
```

This section will go through relevant parts of the Java outlined by the pseudo code above. The code starts with defining the configuration templates and reading the list of endpoints configured and the topology. The Navu API is used for navigating the data models.

```
Template peTemplate = new Template(context, "l3vpn-pe");
```

```
Template ceTemplate = new Template(context,"13vpn-ce");
NavuList endpoints = service.list("endpoint");
NavuContainer topology = ncsRoot.getParent().
    container("http://com/example/13vpn").
    container("topology");
```

The next step is iterating over the VPN endpoints configured in the service, find out connected PE router using small helper methods navigating the configured topology.

```
for(NavuContainer endpoint : endpoints.elements()) {
    try {
        String ceName = endpoint.leaf("ce-device").valueAsString();
        // Get the PE connection for this endpoint router
        NavuContainer conn =
            getConnection(topology,
                endpoint.leaf("ce-device").valueAsString());
        NavuContainer peEndpoint = getConnectedEndpoint(
            conn,ceName);
        NavuContainer ceEndpoint = getMyEndpoint(
            conn,ceName);
```

The parameter dictionary is created from the `TemplateVariables` class and is populated with appropriate parameters.

```
TemplateVariables vpnVar = new TemplateVariables();
vpnVar.putQuoted("PE",peEndpoint.leaf("device").valueAsString());
vpnVar.putQuoted("CE",endpoint.leaf("ce-device").valueAsString());
vpnVar.putQuoted("VLAN_ID", vlan.valueAsString());
vpnVar.putQuoted("LINK_PE_ADR",
    getIPAddress(peEndpoint.leaf("ip-address").valueAsString()));
vpnVar.putQuoted("LINK_CE_ADR",
    getIPAddress(ceEndpoint. leaf("ip-address").valueAsString()));
vpnVar.putQuoted("LINK_MASK",
    getNetMask(ceEndpoint. leaf("ip-address").valueAsString()));
vpnVar.putQuoted("LINK_PREFIX",
    getIPPrefix(ceEndpoint.leaf("ip-address").valueAsString()));
```

The last step after all parameters have been set is applying the templates for the CE and PE routers for this VPN endpoint.

```
peTemplate.apply(service, vpnVar);
ceTemplate.apply(service, vpnVar);
```

Configuration Templates

The configuration templates are XML templates based on the structure of device YANG models. There is a very easy way to create the configuration templates for the service mapping if NSO is connected to a device with the appropriate configuration on it, using the following steps.

-
- | | |
|---------------|--|
| Step 1 | Configure the device with the appropriate configuration. |
| Step 2 | Add the device to NSO |
| Step 3 | Sync the configuration to NSO. |
| Step 4 | Display the device configuration in XML format. |
| Step 5 | Save the XML output to a configuration template file and replace configured values with parameters |
-

The commands in NSO give the following output. To make the example simpler only the BGP part of the configuration is used

```

admin@ncs# devices device cel sync-from
admin@ncs# show running-config devices device cel config \
    ios:router bgp | display xml

<config xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>cel</name>
      <config>
        <router xmlns="urn:ios">
          <bgp>
            <as-no>65101</as-no>
            <neighbor>
              <id>192.168.1.6</id>
              <remote-as>100</remote-as>
              <activate/>
            </neighbor>
            <network>
              <number>10.7.7.0</number>
            </network>
          </bgp>
        </router>
      </config>
    </device>
  </devices>
</config>

```

The final configuration template with the replaced parameters marked in bold is shown below. If the parameter starts with a \$-sign is taken from the Java parameter dictionary, otherwise it is a direct xpath reference to the value from the service instance.

```

<config-template xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device tags="nocreate">
      <name>{$CE}</name>
      <config>
        <router xmlns="urn:ios" tags="merge">
          <bgp>
            <as-no>{/as-number}</as-no>
            <neighbor>
              <id>{$LINK_PE_ADR}</id>
              <remote-as>100</remote-as>
              <activate/>
            </neighbor>
            <network>
              <number>{$LOCAL_CE_NET}</number>
            </network>
          </bgp>
        </router>
      </config>
    </device>
  </devices>
</config-template>

```

FASTMAP Description

FASTMAP covers the complete service life-cycle: creating, changing and deleting the service. The solution requires a minimum amount of code for mapping from a service model to a device model.

FASTMAP is based on generating changes from an initial create. When the service instance is created the reverse of the resulting device configuration is stored together with the service instance. If an NSO

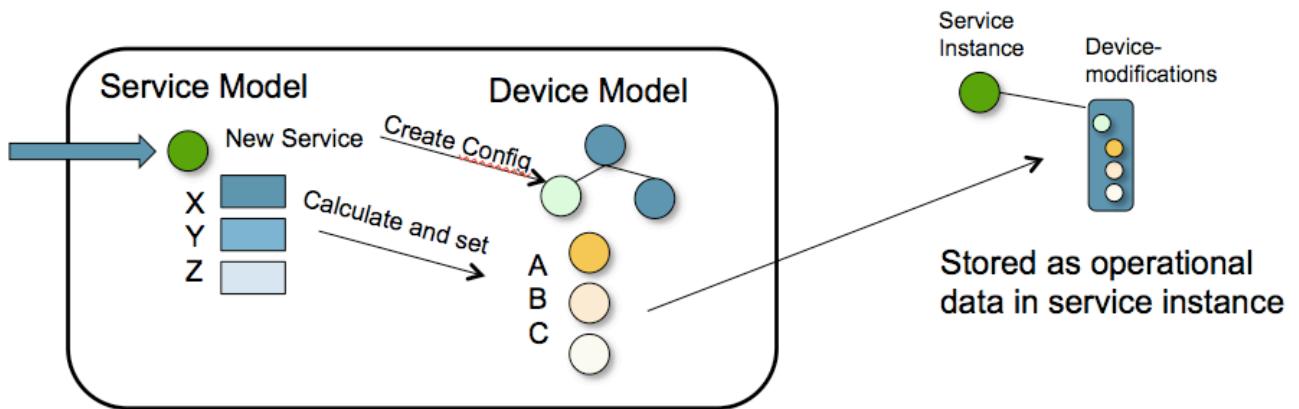
user later changes the service instance, NSO first applies (in a transaction) the reverse diff of the service, effectively undoing the previous results of the service creation code. Then it runs the logic to create the service again, and finally executes a diff to current configuration. This diff is then sent to the devices.

**Note**

This means that it is very important that the service create code produces the same device changes for a given set of input parameters every time it is executed. See [the section called “ Persistent FASTMAP Properties ”](#) for techniques to achieve this.

If the service instance is deleted, NSO applies the reverse diff of the service, effectively removing all configuration changes the service did from the devices.

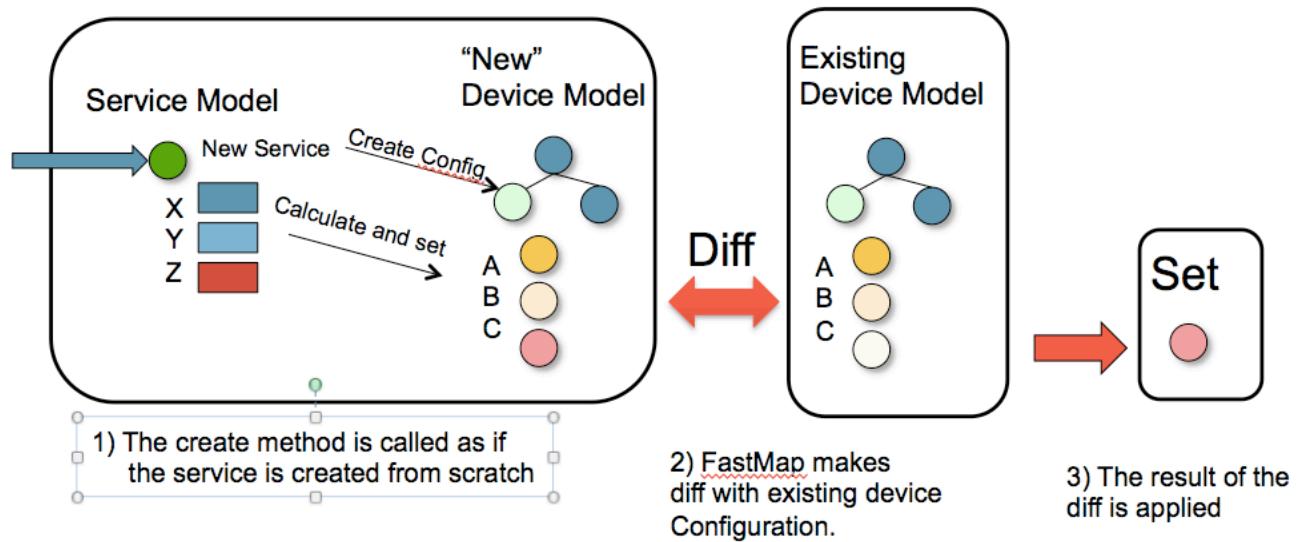
Figure 162. FASTMAP Create a Service



Assume we have a service model that defines a service with attributes X, Y, and Z. The mapping logic calculates that attributes A, B, and C shall be created on the devices. When the service is instantiated, the inverse of the corresponding device attributes A, B, and C are stored with the service instance in the NSO data-store CDB. This inverse answers the question: what should be done to the network to bring it back to the state before the service was instantiated.

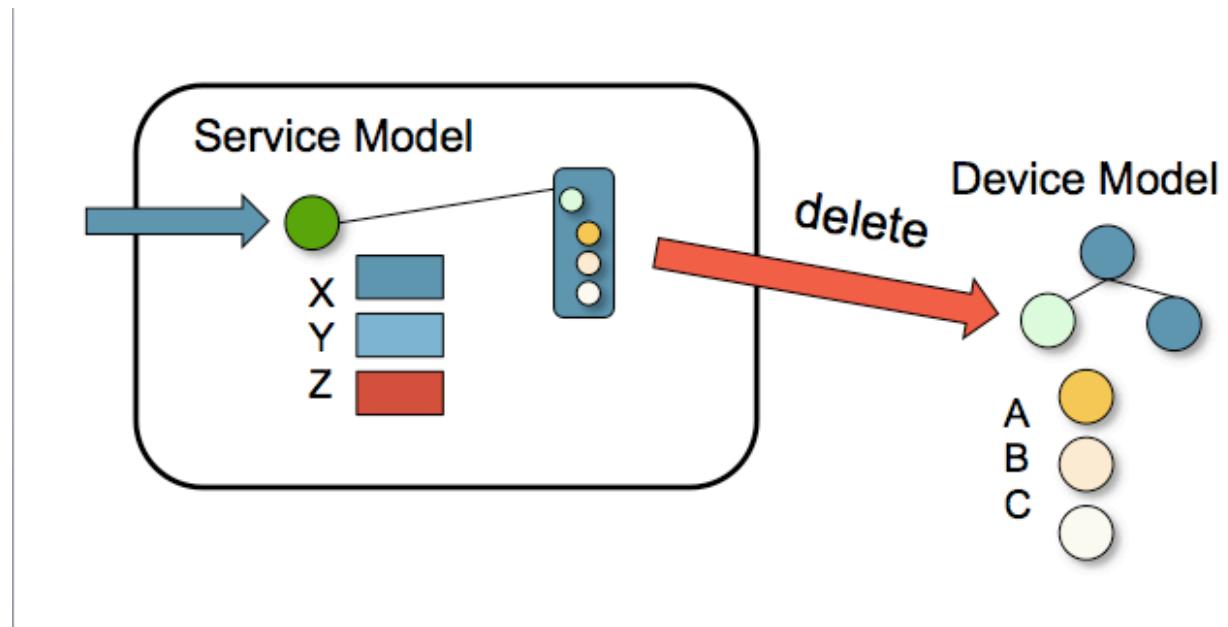
Now let us see what happens if one service attribute is changed. In the scenario below the service attribute Z is changed. NSO will execute this as if the service was created from scratch. The resulting device configurations are then compared with the actual configuration and the minimum diff is sent to the devices. Note that this is managed automatically, there is no code to handle "change Z".

Figure 163. FASTMAP Change a Service



When a user deletes a service instance NSO can pick up the stored device configuration and delete that:

Figure 164. FASTMAP Delete a Service



Reactive FASTMAP

A FASTMAP service is not allowed to perform explicit function calls that have side effects. The only action a service is allowed to take is to modify the configuration of the current transaction. For example, a service may not invoke a RPC to allocate a resource or start a virtual machine. All such actions must take place before the service is created and provided as input parameters to the service. The reason for this

restriction is that the FASTMAP code may be executed as part of a `commit dry-run`, or the commit may fail, in which case the side effects would have to be undone.

Reactive FASTMAP is a design pattern that provides a side-effect free solution to invoking RPCs from a service. In the services discussed previously in this chapter, the service was modeled in such a way that all required parameters were given to the service instance. The mapping logic code could immediately do its work.

Sometimes this is not possible. Two examples where Reactive FASTMAP is the solution are:

- 1 A resource is allocated from an external system, such as an IP address or vlan id. It's not possible to do this allocation from within the normal FASTMAP `create()` code since there is no ways to deallocate the resource on commit abort or failure, and when the service is deleted. Furthermore, the `create()` code runs within the transaction lock. The time spent in the `create()` should be as short as possible.
- 2 The service requires the start of one or more Virtual Machines, Virtual Network Functions. The VMs don't yet exist, and the `create()` code needs to trigger something that starts the VMs, and then later, when the VMs are operational, configure them.

The basic idea is to let the `create()` code not just write data in the `/ncs:devices` tree, but also write data in some auxiliary data structure. A CDB subscriber subscribes to that auxiliary data structure and perform the actual side effect, for example a resource allocation. The response is written to CDB as operational data where the service can read it during subsequent invocations.

The pseudo code for a Reactive FASTMAP service that allocates an id from an id pool may look like this:

```
create(serv) {
    /* request resource allocation */
    ResourceAllocator.requestId(serv, idPool, allocId);

    /* check for allocation response */
    if (!ResourceAllocator.idReady(idPool, allocId))
        return;

    /* read allocation id */
    id = ResourceAllocator.idRead(idPool, allocId);

    /* use id in device config */
    configure(id)
}
```

The actual deployment of a Reactive FASTMAP service will involve multiple executions of the `create()` code.

- 1 In the first run the code will request an id by writing an allocation request to the resource manager tree. It will then check if the response is ready, which it will not be, and return.
- 2 The resource manager subscribes to changes to the resource manager tree, looking for allocation request to be created and deleted. In this case a new allocation request is created. The resource manager allocates the resource and write the response in a CDB oper leaf. Finally the resource manager trigger a service `reactive-re-deploy` action.
- 3 The `create()` is run for a second time. The code will create the allocation request, just as it did the first time, then check if the response is ready. This time it will be ready and the code can proceed to read the allocated id, and use it in its configuration.

Let us make a small digress on the `reactive-re-deploy` action mentioned above. Any service will expose both a `re-deploy` and a `reactive-re-deploy` action. Both actions are similar in that they activate the FASTMAP algorithm and invokes the service `create()` logic. However, while `re-`

`deploy` is user facing and has e.g. dry-run functionality, the `reactive-re-deploy` is specifically tailored for the Reactive FASTMAP pattern. Hence the `reactive-re-deploy` takes no arguments and has no extra functionality, instead it performs the re-deploy as the same user and with the same commit parameters as the original service commit. Also the `reactive-re-deploy` will make a "shallow" re-deploy in the sense that underlying stacked services will not be re-deployed. This "shallow" feature is important when stacked services are used for performance optimization reasons. In the rest of this chapter when service re-deploy is mentioned we will imply that this is performed using the `reactive-re-deploy` action.

In the above `ResourceAllocator` example, when the service is deleted we want the allocated id to be returned to the resource manager and become available for others to allocate. This is achieved as follows.

- 1 The service is deleted with the consequence that all configuration that the service created during its deployment will be removed, in particular the id allocation request will be removed.
- 2 Since the resource manager subscribes to changes in the resource manager tree it will be notified that an allocation request has been deleted. It can then release the resource allocated for this specific request.

Other side effects can be handled in similar ways, for example, starting virtual machines, updating external servers etc. The `resource-manager-example` and `id-allocator-example` packages can be found in `examples.ncs//service-provider/virtual-mpls-vpn`



Note

All packages and NEDs used in the examples are just example packages/NEDs, and are in no way production ready packages nor are they supported. There are official Function Packs (a collection of packages) and NEDs which resembles the packages used in the examples, but they are *not* the same. Never consider packages and NEDs found in the example collection to be official supported packages.

The example in `examples.ncs/getting-started/developing-with-ncs/4-rfs-service` has a package called `vlan-reactive-fastmap` that implements external allocation of a `unit` and a `vlan-id` for the service. The code consists of three parts:

- 1 The YANG model, which is very similar to the `vlan` package previously described in this chapter. The difference is that two parameters are missing, the `unit` and the `vlan-id`. Another difference is that a parallel list structure to the services is maintained. The list entries contain help data and eventually the operational data holding the missing parameters will end up there.
- 2 The `create()` method. This code drives the Reactive FASTMAP loop forward. The YANG model for the service has this structure

```
module: alloc-vlan-service
  +-rw alloc-vlan* [name]
    +-rw name                      string
    +-rw iface                     string
    +-rw description                string
    +-rw arp*                      enumeration
```

The parallel auxiliary model is:

```
module: alloc-vlan-service
  +-rw alloc-vlan-data* [name]
    |  +-rw name                  string
    |  +-rw request-allocate-unit!
    |  |  +-ro unit?      string
    |  +-rw request-allocate-vid!
    |  |  +-ro vlan-id?    uint16
```

When the `create()` method gets called the code creates an allocation request by writing config data into the buddy list entry. It then checks its "buddy" list entry to see if the unit and the vlan-id are there. If they are, the FASTMAP code starts to write into the `/ncs:devices` tree. If they are not it returns.

- 3 A CDB subscriber that subscribes to the `/alloc-vlan-data` tree where the normal FASTMAP `create()` code writes. The CDB subscriber picks up, in this case for example the "CREATE" of `/alloc-vlan-data[name="KEY"]/request-allocate-unit` and allocates a unit number, writes that number as operational data in the `/alloc-vlan-data` tree, and finally redeloys the service, thus triggering the call of `create()` again. This loop of `create()`, CDB subscriber, redeploy continues until the `create()` decides that it has all required data to enter the normal FASTMAP phase, where the code writes to the `/ncs:devices` tree.

There are many variations on this same pattern that can be applied. The common theme is that the `create()` code relies on auxiliary operational data to be filled in. This data contains the missing parameters.

Progress reporting using plan-data

Since the life-cycle of a Reactive FASTMAP service is more complex, there is also a need to report the progress of the service in addition to the success or failure of individual transactions. A Reactive FASTMAP service is expected to be designed with a self-sustained chain of `reactive-re-deploy` until the service has completed. If this chain of `reactive-re-deploy` is broken the service will fail to complete even though no transaction failure has occurred.

To support Reactive FASTMAP service progress reporting there is a YANG grouping, `ncs:plan-data`, and Java API utility `PlanComponent`. It is a recommendation to implement this into the service to promote a "standardized" view of progress reporting.

The YANG submodule defining the `ncs:plan-data` grouping is named `tailf-ncs-plan.yang` and contains the following:

```

submodule tailf-ncs-plan {
    yang-version 1.1;
    belongs-to tailf-ncs {
        prefix ncs;
    }

    import ietf-yang-types {
        prefix yang;
    }

    import tailf-common {
        prefix tailf;
    }

    include tailf-ncs-common;
    include tailf-ncs-services;
    include tailf-ncs-devices;
    include tailf-ncs-log;

    organization "Tail-f Systems";

    description
        This submodule contains a collection of YANG definitions for
configuring plans in NCS.

    Copyright 2016-2021 Cisco Systems, Inc.
    All rights reserved.
    Permission is hereby granted to redistribute this file without

```

```

modification.";

revision 2021-12-17 {
    description
        "Released as part of NCS-5.7.

        Non-backwards-compatible changes have been introduced.

        Obsoleted the usage of the 'service-commit-queue' grouping.

        Added YANG extensions 'all' and 'any'.

        Added YANG extensions 'all' and 'any' as valid substatements
        to the 'pre-condition' YANG extension.

        Added grouping 'pre-condition-grouping' that contains
        data to express nano-service pre-conditions.

        Added status obsolete to leafs 'create-monitor', 'create-trigger-expr',
        'delete-monitor' and 'delete-trigger-expr' in the 'pre-conditions'
        container in the 'nano-plan-components' grouping.

        Added containers 'create' and 'delete' to the 'pre-conditions' 
        containers in the 'nano-plan-components' grouping. Both of the
        new containers contain the grouping 'pre-condition-grouping'.

        Add the converge-on-re-deploy extension.

        Updated plan-location type to yang:xpath1.0.

        Update the description of the self-as-service-status extension.";
}

revision 2021-09-02 {
    description
        "Released as part of NCS-5.6.

        Updated the description and added the 'sync' parameter to the
        /zombies/service/reactive-re-deploy action.

        Remove mandatory statement from status leaf in plan-state-change
        notification.

        Added commit-queue container and trace-id leaf to plan-state-change
        notification.

        Remove unique statement from /services/plan-notifications/subscription.";
}

revision 2021-02-09 {
    description
        "Released as part of NCS-5.5.1.

        Added the 'ncs-commit-params' grouping to the
        /zombies/service/re-deploy action input parameters.

        Added /zombies/service/latest-commit-parameters leaf.";
}

revision 2020-11-26 {
    description
        "Released as part of NCS-5.5.

```

```

    Add when statement to variables.

    Add self-as-service-status to plan-outline.

    Added the ncs-commit-params grouping to the force-back-track input
    parameters.

    Add new extension deprecates-component and a version leaf to
    nano components.";
}

revision 2020-06-25 {
    description
        "Released as part of NCS-5.4.

        Add trigger-on-delete trigger type to precondition monitors.

        Add sync option to post actions.

        Remove the experimental tag from the plan-location statement.

        Add purge action to side-effect queue and make the automatic
        cleanup of the side-effect queue configurable.

        Add force-commit to create and delete.

        Add zombies/service/pending-delete leaf.

        Add zombies/service/plan/commit-queue container.

        Use service-commit-queue grouping under zombies/service.

        Add load-device-config action under
        commit-queue/queue-item/failed-device for zombies.

        Add canceled as a valid value of side-effect-queue/status.
        The status of a side-effect is canceled if a related
        commit-queue item has failed. Canceled side-effects are
        cleaned up as part of the side-effect queue's automatic cleanup.";
}

revision 2019-11-28 {
    description
        "Released as part of NCS-5.3.

        Added dry-run option to the zombie resurrect action.

        Added service log and error-info to zombies.

        Added reactive-re-deploy action to revive and
        reactive-re-deploy a zombie.";

}

revision 2019-04-09 {
    description
        "Released as part of NCS-5.1.

        Added operation leaf to plan-state-change notification and to
        subscription list.

```

```

        Added ned-id-list in the service's private data.";  

    }  
  

revision 2018-11-12 {  

    description  

    "Released as part of NCS-4.7.2.  

    Major changes to nano services.";  

}  
  

revision 2018-06-21 {  

    description  

    "Released as part of NCS-4.7.  

    Added commit-queue container in plan.";  

}  
  

revision 2017-03-16 {  

    description  

    "Released as part of NCS-4.4.  

    Added error-info container in plan for additional error information.";  

}  
  

revision 2016-11-24 {  

    description  

    "Released as part of NCS-4.3.  

    Major additions to this submodule to incorporate Nano Services.";  

}  
  

revision 2016-05-26 {  

    description  

    "Initial revision";  

}  
  

typedef plan-xpath {  

    type yang:xpath1.0;  

    description  

    "This type represents an XPath 1.0 expression that is evaluated  

    in the following context:  


- The set of namespace declarations are the prefixes defined  

        in all YANG modules implemented, mapped to the namespace  

        defined in the corresponding module.
- The set of variable bindings contains all variables  

        declared with 'ncs:variable' that are in scope, and all  

        variables defined in the service code's 'opaque' key-value  

list (if any), and the following variables:  

  - 'SERVICE': a nodeset with the service instance node as the  

            only member, or no nodes if the service  

            instances is being deleted.
  - 'ZOMBIE': a nodeset with the service instance node as the  

            only member when it is being deleted, or no  

            nodes if the service instance exists.
  - 'PLAN': a nodeset with the 'plan' container for the service  

            instance as the only member.

```

```

    o The function library is the core function library.

    o If this expression is in a descendant to a 'ncs:foreach' statement, the context node is the node in the node set in the 'ncs:foreach' result. Otherwise, the context node is initially the service instance node.

    ";
}

/*
 * Plan Component Types
 */

typedef plan-component-type-t {
    description
        "This is a base type from which all service specific plan components can be derived.";  

    type identityref {
        base plan-component-type;
    }
}

identity plan-component-type {
    description
        "A service plan consists of several different plan components. Each plan component moves forward in the plan as the service comes closer to fulfillment.";  

}
identity self {
    description
        "A service should when it constructs its plan, include a column of type 'self', this column can be used by upper layer software to determine which state the service is in as a whole.";  

    base plan-component-type;
}

/*
 * Plan States
 */

typedef plan-state-name-t {
    description
        "This is a base type from which all plan component specific states can be derived.";  

    type identityref {
        base plan-state;
    }
}

typedef plan-state-operation-t {
    type enumeration {
        enum created {
            tailf:code-name "plan_state_created";
        }
        enum modified {
            tailf:code-name "plan_state_modified";
        }
        enum deleted {
    }
}

```

```

        tailf:code-name "plan_state_deleted";
    }
}
}

typedef plan-state-status-t {
    type enumeration {
        enum not-reached;
        enum reached;
        enum failed {
            tailf:code-name "plan_failed";
        }
    }
}

typedef side-effect-q-status-t {
    type enumeration {
        enum not-reached;
        enum reached;
        enum failed {
            tailf:code-name "plan_failed";
        }
        enum canceled {
            tailf:code-name "effect_canceled";
        }
    }
}

typedef plan-state-action-status-t {
    type enumeration {
        enum not-reached;
        enum create-reached;
        enum delete-reached;
        enum failed {
            tailf:code-name "plan_action_failed";
        }
        enum create-init;
        enum delete-init;
    }
}

identity plan-state {
    description
        "This is the base identity for plan states. A plan component in a plan goes through certain states, some, such as 'init' and 'ready', are specified here, and the application augments these with app specific states.";
}

identity init {
    description
        "The init state in all plan state lists, primarily used as a place holder with a time stamp.";
    base plan-state;
}

identity ready {
    description
        "The final state in a 'state list' in the plan";
    base plan-state;
}

```

```

/*
 * Plan Notifications
 */

augment "/ncs:services" {
    container plan-notifications {
        description
            "Configuration to send plan-state-change notifications for
            plan state transitions. A notification can be configured to
            be sent when a specified service's plan component enters a
            given state.

            The built in stream 'service-state-changes' is used to send
            these notifications.";
        list subscription {
            key name;
            description
                "A list of our plan notification subscriptions.";

            leaf name {
                type string;
                description
                    "A unique identifier for this subscription.";
            }
            leaf service-type {
                type tailf:node-instance-identifier;
                tailf:cli-completion-actionpoint "servicepoints-with-plan";
                description
                    "The type of service. If not set, all service types are
                    subscribed.";
            }
            leaf component-type {
                type plan-component-type-t;
                description
                    "The type of component in the service's plan. If not set,
                    all component types of the specified service types are
                    subscribed.";
            }
            leaf state {
                type plan-state-name-t;
                description
                    "The name of the state for the component in the service's plan.
                    If not set, all states of the specified service types and
                    plan components are subscribed.";
            }
            leaf operation {
                type plan-state-operation-t;
                description
                    "The type of operation performed on the state(s) in the
                    component(s). If not set, all operations are subscribed.";
            }
        }
    }
}

notification plan-state-change {
    description
        "This notification indicates that the specified service's
        plan component has entered the given state.

        This notification is not sent unless the system has been
        configured to send the notification for the service type.";
```

```

leaf service {
    type instance-identifier;
    mandatory true;
    description
        "A reference to the service whose plan has been changed.";
}
leaf component {
    type string;
    description
        "Refers to the name of a component in the service's plan;
        plan/component/name." ;
}
leaf state {
    type plan-state-name-t;
    mandatory true;
    description
        "Refers to the name of the new state for the component in
        the service's plan;
        plan/component/state"; ;
}
leaf operation {
    type plan-state-operation-t;
    description
        "The type of operation performed on the given state." ;
}
leaf status {
    type plan-state-status-t;
    description
        "Refers to the status of the new state for the component in
        the service's plan;
        plan/component/state/status"; ;
}
container commit-queue {
    presence "The service is being committed through the commit queue." ;
    list queue-item {
        key id;
        max-elements 1;
        leaf id {
            type uint64;
            description
                "If the queue item in the commit queue refers to this service
                this is the queue number." ;
        }
        leaf tag {
            type string;
            description
                "Opaque tag set in the commit." ;
        }
    }
}
leaf trace-id {
    type string;
    description
        "The trace id assigned to the commit that last changed
        the service instance." ;
}
}

/*
 * Groupings
 */

```

```

grouping plan-data {
    description
        "This grouping contains the plan data that can show the
         progress of a Reactive FASTMAP service. This grouping is optional
         and should only be used by services i.e lists or presence containers
         that uses the ncs:servicepoint callback";
    container plan {
        config false;
        tailf:cdb-oper {
            tailf:persistent true;
        }
        uses plan-components;
        container commit-queue {
            presence "The service is being committed through the commit queue.";
            list queue-item {
                key id;
                leaf id {
                    type uint64;
                    description
                        "If the queue item in the commit queue refers to this service
                         this is the queue number.";
                }
            }
        }
        leaf failed {
            type empty;
            description
                "This leaf is present if any plan component in the plan is in
                 a failed state; i.e., a state with status 'failed', or
                 if the service failed to push its changes to the network.";
        }
        container error-info {
            presence "Additional info if plan has failed";
            leaf message {
                type string;
                description
                    "An explanatory message for the failing plan.";
            }
            leaf log-entry {
                type instance-identifier {
                    require-instance false;
                }
                description
                    "Reference to a service log entry with additional information.";
            }
        }
    }
    container plan-history {
        config false;
        tailf:cdb-oper {
            tailf:persistent true;
        }
        list plan {
            key time;
            description
                "Every time the plan changes its structure, i.e., a
                 plan component is added or deleted, or a state is added or
                 deleted in a plan component, a copy of the old plan is stored
                 in the plan history list.";
            leaf time {
                type yang:date-and-time;
            }
        }
    }
}

```

```

        tailf:cli-value-display-template "${./datetime}";
    }
    uses plan-components;
}
}

grouping plan-components {
    description
    "This grouping contains a list of components that reflects the
     different steps or stages that a Reactive FASTMAP service comprises.";
list component {
    ordered-by user;
    key name;
    description
    "A component has a type and a list of states.
     It is required that the first plan component is of type ncs:self.
     It is also required that the first state of a component is ncs:init
     and the last state is ncs:ready.
     A service can in addition to the 'self' component have any number of
     components. These additional components will have types that are
     defined by user specified YANG identities.";

    uses plan-component-body {
        refine "state/status" {
            mandatory true;
        }
    }
}

grouping plan-component-body {
    leaf name {
        type string;
    }
    leaf type {
        description
        "The plan component type is defined by an YANG identity.
         It is used to identify the characteristics of a certain component.
         Therefore, if two components in the same service are of the same
         type they should be identical with respect to number, type and order
         of their contained states.";

        type plan-component-type-t;
        mandatory true;
    }
    list state {
        description
        "A plan state represents a certain step or stage that a service needs
         to execute and/or reach. It is identified as an YANG identity.
         There are two predefined states ncs:init and ncs:ready which is the
         first respectively last state of a plan component.";

        ordered-by user;
        key name;
        leaf name {
            tailf:alt-name state;
            type plan-state-name-t;
        }
        leaf status {
            description
            "A plan state is always in one of three states 'not-reached' when

```

the state has not been executed, '**reached**' when the state has been executed and '**failed**' if the state execution failed.";

```

type plan-state-status-t;
}
leaf when {
    type yang:date-and-time;
    tailf:cli-value-display-template "$(./datetime)";
    when '../status != "not-reached"';
    description
        "The time this state was successfully reached or failed.";
}
leaf service-reference {
    description
        "If this component reflects the state of some other data, e.g
        an instantiated RFS, an instantiated CFS or something else, this
        optional field can be set to point to that instance";
    type instance-identifier {
        require-instance false;
    }
    tailf:display-column-name "ref";
}
}
/*
 * Nano-service related definitions
 */
grouping force-back-track-action {
    tailf:action force-back-track {
        tailf:info "Force a component to back-track";
        description
            "Forces an existing component to start back-tracking";
        tailf:actionpoint ncsinternal {
            tailf:internal;
        }
        input {
            leaf back-tracking-goal {
                type leafref {
                    path "../state/name";
                }
                description
                    "Target state for back-track.";
            }
            uses ncs-commit-params;
        }
        output {
            leaf result {
                type boolean;
                description
                    "Set to true if the forced back tracking was successful,
                    otherwise false.";
            }
            leaf info {
                type string;
                description
                    "A message explaining why the forced back tracking wasn't
                    successful.";
            }
        }
    }
}

```

```

}

grouping post-action-input-params {
    description
        "A Nano service post-action can choose to implement this grouping
        as its input parameters. If so the action will be invoked with:
            * opaque-props      - The list of name, value pairs in the service opaque
            * component-props   - The list of component properties for
                                the invoking plan component state.

        post-actions that does not implement this grouping as its input
        parameters will be invoked with an empty parameter list.";
    list opaque-props {
        key name;
        leaf name {
            type string;
        }
        leaf value {
            type string;
        }
    }
    list component-props {
        key name;
        leaf name {
            type string;
        }
        leaf value {
            type string;
        }
    }
}

grouping nano-plan-data {
    description
        "This grouping is required for nano services. It replaces the
        plan-data grouping. This grouping contains an executable plan
        that has additional state data which is internally used to
        control service execution.";
    uses nano-plan;
}

grouping nano-plan {
    container plan {
        config false;
        tailf:cdb-oper {
            tailf:persistent true;
        }
        uses nano-plan-components {
            augment "component" {
                uses force-back-track-action;
            }
        }
        container commit-queue {
            presence "The service is being committed through the commit queue.";
            list queue-item {
                key id;
                leaf id {
                    type uint64;
                    description
                        "If the queue item in the commit queue refers to this service
                        this is the queue number.";
                }
            }
        }
    }
}

```

```

    }
}

leaf failed {
    type empty;
    description
        "This leaf is present if any plan component in the plan is in
         a failed state; i.e., a state with status 'failed', or
         if the service failed to push its changes to the network.";
}

container error-info {
    presence "Additional info if plan has failed";
    leaf message {
        type string;
        description
            "An explanatory message for the failing plan.";
    }
    leaf log-entry {
        type instance-identifier {
            require-instance false;
        }
        description
            "Reference to a service log entry with additional information.";
    }
}

leaf deleting {
    tailf:hidden fastmap-private;
    type empty;
}

leaf service-location {
    tailf:hidden fastmap-private;
    type instance-identifier {
        require-instance false;
    }
}

grouping nano-plan-components {
    description
        "This grouping contains a list of components that reflects the
         different steps or stages that a nano service comprises.";
    list component {
        ordered-by user;
        key "type name";
        description
            "A component has a type and a list of states. It is required
             that the first plan component is of type ncs:self. It is
             also required that the first state of a component is ncs:init
             and the last state is ncs:ready. A service can in addition
             to the 'self' component have any number of components. These
             additional components will have types that are defined by
             user specified YANG identities.";
    }

    uses plan-component-body {
        augment "state" {
            leaf create-cb {
                tailf:hidden full;
                description
                    "indicate if a create callback should be registered
                     for this state";
                type boolean;
            }
        }
    }
}

```

```

leaf create-force-commit {
    tailf:hidden full;
    description
        "Indicate if the current transaction should be committed before
         running any later states.";
    type boolean;
    default false;
}

leaf delete-cb {
    tailf:hidden full;
    description
        "indicate if a delete callback should be registered
         for this state";
    type boolean;
}

leaf delete-force-commit {
    tailf:hidden full;
    description
        "Indicate if the current transaction should be committed before
         running any later states.";
    type boolean;
    default false;
}

container pre-conditions {
    tailf:display-groups "summary";
    description
        "Pre-conditions for a state controls whether or not a
         state should be executed. There are separate conditions
         for the 'create' and 'delete' case. At create the
         create conditions checked and if possible executed with
         the ultimate goal for the state of having status
         'reached'. At the 'delete' case the delete conditions
         control whether the state changes should be deleted
         with the ultimate goal of the state having status
         'not-reached'";

    presence "Preconditions for executing the plan state";

    // Kept for backwards compatibility
    leaf create-trigger-expr {
        status obsolete;
        type yang>xpath1.0;
    }

    leaf create-monitor {
        status obsolete;
        type yang>xpath1.0;
    }

    leaf delete-trigger-expr {
        status obsolete;
        type yang>xpath1.0;
    }

    leaf delete-monitor {
        status obsolete;
        type yang>xpath1.0;
    }
}

```

```

grouping pre-condition-grouping {
    leaf fun {
        type enumeration {
            enum all {
                tailf:code-name fun-all;
            }
            enum any {
                tailf:code-name fun-any;
            }
        }
    }
    list pre-condition {
        key id;
        leaf id {
            type string;
        }
        leaf monitor {
            type yang>xpath1.0;
        }
        leaf trigger-expr {
            type yang>xpath1.0;
        }
    }
}
container create {
    presence "Create precondition exists";
    uses pre-condition-grouping;
}
container delete {
    presence "Delete precondition exists";
    uses pre-condition-grouping;
}
container post-actions {
    tailf:display-groups "summary";

    description
        "post-actions are called after successful execution of a
         state. These are optional and there are separate
         action that can be set for the 'create' and 'delete'
         case respectively.

        These actions are put as requests in the
        side-effect-queue and are executed asynchronously with
        respect to the original service transaction.";

    presence "Asynchronous side-effects after successful execution";
    leaf create-action-node {
        description
            "This leaf identifies the node on which a specified
             action resides. This action is called after this state
             as got a 'reached' status.";
        type yang>xpath1.0;
    }
    leaf create-action-name {
        description
            "The name of the action.";
        type string;
    }
    leaf create-action-result-expr {

```

description
"An action responds with a structured result. A certain value could indicate an error or a successful result, e.g. 'result true'.

This statement describes an XPath expression to evaluate the result of the action so that the side-effect-queue can indicate action errors.

The result of the expression is converted to a boolean using the standard XPath rules. If the result is '**true**' the action is reported as successful, otherwise as failed.

The context for evaluating this expression is the resulting xml tree of the action.

The set of **namespace** declarations are all available namespaces, with the prefixes defined in the modules.";

```

type yang>xpath1.0;
}
choice create-action-operation-mode {
    description
        "Specifies if the create post action should be run synchronously or not.";
    leaf create-action-async {
        type empty;
    }
    leaf create-action-sync {
        type empty;
    }
    default create-action-async;
}
leaf delete-action-node {
    description
        "This leaf identifies the node on which a specified action resides. This action is called after this state as got a 'not-reached' status.";
    type yang>xpath1.0;
}
leaf delete-action-name {
    description
        "The name of the action.";
    type string;
}
leaf delete-action-result-expr {
    description
        "An action responds with a structured result. A certain value could indicate an error or a successful result, e.g. 'result true'.

```

This statement describes an XPath expression to evaluate the result of the action so that the side-effect-queue can indicate action errors.

The result of the expression is converted to a boolean using the standard XPath rules. If the result is '**true**' the action is reported as successful, otherwise as failed.

The context for evaluating this expression is the resulting xml tree of the action.

The set of **namespace** declarations are all available namespaces,

```

        with the prefixes defined in the modules.";
    type yang:xpath1.0;
}
choice delete-action-operation-mode {
    description
        "Specifies if the delete post action should be run synchronously
         or not.";
    leaf delete-action-async {
        type empty;
    }
    leaf delete-action-sync {
        type empty;
    }
    default delete-action-async;
}
leaf post-action-status {
when '../post-actions';
type plan-state-action-status-t;
description
    "This leaf is initially set to 'not-reached'.

    If a post-action was specified, and returned
    successfully, this leaf will be set to 'create-reached'
    if the component is not back-tracking, and
    'delete-reached' if it is back-tracking.

    If the post-action did not return successfully, this
    leaf is set to 'failed'.";
}

container modified {
tailf:display-groups "summary";
config false;
tailf:callpoint ncs {
    tailf:internal;
}
description
    "Devices and other services this service has modified directly or
     indirectly (through another service).";
tailf:info
    "Devices and other services this service modified directly or
     indirectly.";
leaf-list devices {
tailf:info
    "Devices this service modified directly or indirectly";
type leafref {
    path "/ncs:devices/ncs:device/ncs:name";
}
}
leaf-list services {
tailf:info
    "Services this service modified directly or indirectly";
type instance-identifier {
    require-instance false;
}
}
leaf-list lsa-services {
tailf:info
    "Services residing on remote LSA nodes this service
     has modified directly or indirectly.";
}

```

```

        type instance-identifier {
            require-instance false;
        }
    }

container directly-modified {
    tailf:display-groups "summary";
    config false;
    tailf:callpoint ncs {
        tailf:internal;
    }
    description
        "Devices and other services this service has explicitly
         modified.";
    tailf:info
        "Devices and other services this service has explicitly
         modified.";
    leaf-list devices {
        tailf:info
            "Devices this service has explicitly modified.";
        type leafref {
            path "/ncs:devices/ncs:device/ncs:name";
        }
    }
    leaf-list services {
        tailf:info
            "Services this service has explicitly modified.";
        type instance-identifier {
            require-instance false;
        }
    }
    leaf-list lsa-services {
        tailf:info
            "Services residing on remote LSA nodes this service
             has explicitly modified.";
        type instance-identifier {
            require-instance false;
        }
    }
}

uses service-get-modifications;

container private {
    description
        "NCS service related internal data stored here.";
    tailf:hidden fastmap-private;
    ncs:ncs-service-private;
    leaf diff-set {
        description
            "Internal node use by NCS service manager to remember
             the reverse diff for a service instance. This is the
             data that is used by FASTMAP";
        tailf:hidden full;
        type binary;
    }
    leaf forward-diff-set {
        description
            "Internal node use by NCS service manager to remember
             the forwards diff for a service instance. This data is
             is used to produce the proper 'get-modifications' output";
    }
}

```

```

        tailf:hidden full;
        type binary;
    }
leaf-list device-list {
    description
        "A list of managed devices this state has manipulated.";
    tailf:hidden full;
    type string;
}
leaf-list ned-id-list {
    description
        "A list of NED identities this service instance has
        manipulated.";
    tailf:hidden full;
    type string;
}
leaf-list service-list {
    description
        "A list of services this state has manipulated.";
    tailf:hidden full;
    type instance-identifier {
        require-instance false;
    }
}
leaf-list lsa-service-list {
    description
        "A list of LSA services this service instance has manipulated.";
    tailf:hidden full;
    type instance-identifier {
        require-instance false;
    }
}
}
container private {
    description
        "NCS service related internal data stored here.";
    tailf:hidden fastmap-private;
}

container property-list {
    description
        "FASTMAP service component instance data used by the
        service implementation.";
list property {
    key name;
    leaf name {
        type string;
    }
    leaf value {
        type string;
    }
}
leaf back-track {
    type boolean;
    default false;
}
leaf back-track-goal {
    tailf:alt-name goal;
    type plan-state-name-t;
}

```

```

        }
    leaf version {
        tailf:hidden full;
        type uint32;
    }
}

grouping nano-plan-history {
    container plan-history {
        config false;
        tailf:cdb-oper {
            tailf:persistent true;
        }
        list plan {
            key time;
            description
                "Every time the plan changes its structure, i.e., a
                plan component is added or deleted, or a state is added or
                deleted in a plan component, a copy of the old plan is stored
                in the plan history list.";
```

- leaf time {
 type yang:date-and-time;
 tailf:cli-value-display-template "\${./datetime}";
 }
 uses nano-plan-components;
 }
}
}

/*
 * Internal structures
 */

container side-effect-queue {

- list side-effect {**
- config false;
 tailf:cdb-oper {
 tailf:persistent true;
 }

key id;
leaf id {
 description
 "Unique identification of the side-effect action";
 type string;
}
leaf created {
 type yang:date-and-time;
}
leaf invoked {
 type yang:date-and-time;
}
leaf service {
 description
 "The service that added the side effect.";
 type string;
}
leaf requestor {
 description
 "Path to the requester of side-effect.

```

    Typically a plan state for a service.";
  type string;
}
leaf requestor-op {
  description
    "The base operation for the request-or when issuing the side-effect.";
  type enumeration {
    enum create {
      tailf:code-name op_create;
    }
    enum delete {
      tailf:code-name op_delete;
    }
  }
leaf action-node {
  description
    "This leaf identifies the node on which a specified
     action resides.";
  type yang>xpath1.0;
}
leaf action-name {
  description
    "The name of the action.";
  type yang:yang-identifier;
}
list variable {
  key name;
  description
    "A list of variable bindings that will be part of the
     context when the action-node path expression is evaluated.";
leaf name {
  type string;
  description
    "The name of the variable";
}
leaf value {
  type yang>xpath1.0;
  mandatory true;
  description
    "An XPath expression that will be the value of the variable
     'name'. Note that both expressions and path expressions are
     allowed, which implies that literals must be quoted.";
}
leaf result-expr {
  description
    "An action responds with a structured result. A certain
     value could indicate an error or a successful result, e.g.
     'result true'.
```

This statement describes an XPath expression to evaluate the result of the action so that the side-effect-queue can indicate action errors.

The result of the expression is converted to a boolean using the standard XPath rules. If the result is '**true**' the action is reported as successful, otherwise as failed.

The context for evaluating this expression is the resulting xml tree of the action.

```
There are no variable bindings in this evaluation.  
The set of namespace declarations are all available namespaces,  
with the prefixes defined in the modules.";  
    type yang:xpath1.0;  
}  
leaf status {  
    description  
        "Resulting status to be set as the request's post-action-status.";  
    type side-effect-q-status-t;  
}  
leaf error-message {  
    description  
        "An additional error message for the action if this is applicable.  
        I.e. an error is thrown.";  
    type string;  
}  
leaf u-info {  
    tailf:hidden full;  
    type binary;  
}  
leaf sync {  
    type boolean;  
}  
}  
  
container settings {  
    description  
        "Settings related to the side effect queue.";  
    container automatic-purge {  
        description  
            "Settings for the automatic purging of side effects.";  
        container failed-queue-time {  
            description  
                "The time failed side effects should be kept in the queue.";  
            choice failed-queue-time-choice {  
                leaf forever {  
                    description  
                        "Failed side effects should be kept forever.";  
                    type empty;  
                }  
                leaf seconds {  
                    type uint16;  
                }  
                leaf minutes {  
                    type uint16;  
                }  
                leaf hours {  
                    type uint16;  
                }  
                leaf days {  
                    type uint16;  
                    default 7;  
                }  
                default days;  
            }  
        }  
    }  
}  
  
tailf:action invoke {  
    tailf:info "Invoke queued side-effects asynchronously";  
    description
```

```

    "Invokes all not already executing/executed side-effects in the
     side effect queue.";
tailf:actionpoint ncsinternal {
    tailf:internal;
}
input {
}
output {
    leaf num-invoked {
        type uint32;
    }
}
}

tailf:action purge {
    tailf:info "Purge all failed side effects";
    description
        "Purge all failed side effects.";
    tailf:actionpoint ncsinternal {
        tailf:internal;
    }
    input {
    }
    output {
        leaf purged-side-effects {
            type uint16;
        }
    }
}
}

container zombies {
    config false;
    tailf:cdb-oper {
        tailf:persistent true;
    }
    description
        "Container for deleted Nano Services that still perform staged deletes.';

list service {
    key service-path;
    leaf service-path {
        description
            "The path to where the service resided that has been deleted
             and become a zombie.";
        type string;
    }
    leaf delete-path {
        description
            "The path to the node nearest to the top that was deleted and resulted
             in this service becoming a zombie.";
        type string;
    }
    leaf pending-delete {
        type empty;
    }

    leaf diffset {
        tailf:hidden full;
        type binary;
    }
}

```

```

leaf latest-commit-params {
    tailf:hidden full;
    type binary;
    description
        "Latest transactions commit parameters are stored there, these are
         used in reactive-re-deploy actions that must have the same
         parameters as the original service commit.";
}
leaf latest-u-info {
    tailf:hidden full;
    type binary;
    description
        "Latest transactions user info is stored there, these are
         used in reactive-re-deploy actions that must be performed by
         a user with the same user info.";
}

container private {
    leaf-list device-list {
        description
            "A list of managed devices this state has manipulated.";
        tailf:hidden full;
        type string;
    }
}

container plan {
    uses nano-plan-components {
        augment "component" {
            uses force-back-track-action;
        }
    }
    container commit-queue {
        presence "The service is being committed through the commit queue.";
        list queue-item {
            key id;
            leaf id {
                type uint64;
                description
                    "If the queue item in the commit queue refers to this service
                     this is the queue number.";
            }
        }
    }
}

leaf failed {
    tailf:code-name "failedx";
    type empty;
}
container error-info {
    presence "Additional info if plan has failed";
    leaf message {
        type string;
        description
            "An explanatory message for the failing plan.";
    }
    leaf log-entry {
        type instance-identifier {
            require-instance false;
        }
        description
            "Reference to a service log entry with additional information.";
    }
}

```

```

        }
    }
leaf deleting {
    tailf:hidden fastmap-private;
    type empty;
}
}

tailf:action re-deploy {
    tailf:info "revive the zombie and re-deploy it.";
    description
        "The nano service became a zombie since it was deleted but not
         all delete pre-conditions was fulfilled. This action revives the
         zombie service and re-deploys and stores it back as a zombie if
         necessary. This will be performed with the user who requested the
         action.";
    tailf:actionpoint ncsinternal {
        tailf:internal;
    }
    input {
        uses ncs-commit-params;
    }
    output {
        uses ncs-commit-result;
    }
}
tailf:action reactive-re-deploy {
    tailf:info "revive the zombie and reactive re-deploy it.";
    description
        "The nano service became a zombie since it was deleted but not
         all delete pre-conditions was fulfilled. This action revives the
         zombie service and re-deploys and stores it back as a zombie if
         necessary. This will be performed with the same user as the original
         commit.

By default this action is asynchronous and returns nothing.";

tailf:actionpoint ncsinternal {
    tailf:internal;
}
input {
    leaf sync {
        description
            "By default the action is asynchronous, i.e. it does not wait for
             the service to be re-deployed. Use this leaf to get synchronous
             behaviour and block until the service re-deploy transaction is
             committed. It 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.";
        type empty;
    }
}
output {
    uses ncs-commit-result;
}
}
tailf:action resurrect {
    tailf:info "Load the zombie back as service in current state.";
    description
        "The zombie resurrection is used to stop the progress of a staged
         nano service delete and restore current state as is.";
    tailf:actionpoint ncsinternal {

```

```

tailf:internal;
}

input {
    container dry-run {
        presence "";
        leaf outformat {
            type outformat3;
        }
    }
}

output {
    leaf result {
        type string;
    }
}

choice outformat {
    case case-xml {
        uses dry-run-xml;
    }
    case case-cli {
        uses dry-run-cli;
    }
    case case-native {
        uses dry-run-native;
    }
}
}

uses log-data;

uses service-commit-queue {
    status obsolete;
    augment "commit-queue/queue-item/failed-device" {
        tailf:action load-device-config {
            tailf:display-when ".../config-data != ''";
            tailf:info "Load device configuration into an open transaction.";
            description
                "Load device configuration into an open transaction.";
            tailf:actionpoint ncsinternal {
                tailf:internal;
            }
        }
        input {
        }
        output {
            leaf result {
                type string;
            }
        }
    }
}
}

/* 
 * Plan Extension Statements
 */

extension plan-outline {
    argument id {
        tailf:arg-type {
            type tailf:identifier;
        }
}

```

```

}
tailf:occurrence "*";
tailf:use-in "module";
tailf:use-in "submodule";
tailf:substatement "description";
tailf:substatement "ncs:self-as-service-status" {
    tailf:occurrence "?";
}
tailf:substatement "ncs:component-type" {
    tailf:occurrence "+";
}
description
"This statement is optionally used in a node that defines a service to document its plan. It is required for a nano service."

A plan is outlined by listing all component-types that the service can instantiate, and their related states. Note that a specific service instance may instantiate zero, one, or more components of a certain type.

It is required that a plan has one component of type ncs:self.";
```

}

```

extension self-as-service-status {
description
"If this statement has been set on a plan outline the self components init and ready states status will reflect the overall status of the service."

The self components ready state will not be set to reached until all other components ready states have been set to reached and all post actions have been run successfully. Likewise when deleting a service the init state will not be set to not-reached (and the service deleted) until all other components init states have had their status set to not-reached and any post actions have been run successfully.

If any state in a component, other than the self component, or post action have failed the ready/init state of the self component will also be set to failed to reflect that the service has failed.";
```

}

```

extension component-type {
argument name {
    tailf:arg-type {
        type tailf:identifier-ref;
    }
}
tailf:substatement "description";
tailf:substatement "ncs:state" {
    tailf:occurrence "*";
}
description
"This statement identifies the component type, which is a reference to a YANG identity."

A component-type contains an ordered list of states which in also are references to YANG identities. It is required that the first state in a component-type is ncs:init and the last state is ncs:ready.

Each state represents a unit of work performed by the
```

```

        service when a certain pre condition is satisfied.";
    }

extension state {
    argument name {
        tailf:arg-type {
            type tailf:identifier-ref;
        }
    }
    tailf:substatement "description";
    tailf:substatement "ncs:create" {
        tailf:occurence "?";
    }
    tailf:substatement "ncs:delete" {
        tailf:occurence "?";
    }
}

description
"This statement identifies the state, which is a reference to a YANG identity.

It represents a unit of work performed by the service when a certain pre condition is satisfied.";
}

extension create {
    tailf:substatement "description";
    tailf:substatement "ncs:nano-callback" {
        tailf:occurence "?";
    }
    tailf:substatement "ncs:pre-condition" {
        tailf:occurence "?";
    }
    tailf:substatement "ncs:post-action-node" {
        tailf:occurence "?";
    }
    tailf:substatement "ncs:force-commit" {
        tailf:occurence "?";
    }
}

description
"This statement defines nano service state characteristics for entering this state.

The component will advance to this state when it is not back tracking, it has reached its previous state, and the 'pre-condition' is met.

If the 'nano-callback' statement is defined, it means that there is a callback function (or template) that will be invoked before this state is entered.

The 'post-action-node' optionally defines an action to be invoked when this state has been entered.";
}

extension delete {
    tailf:substatement "description";
    tailf:substatement "ncs:nano-callback" {
        tailf:occurence "?";
    }
    tailf:substatement "ncs:pre-condition" {

```

```

        tailf:occurrence "?";
    }
    tailf:substatement "ncs:post-action-node" {
        tailf:occurrence "?";
    }
    tailf:substatement "ncs:force-commit" {
        tailf:occurrence "?";
    }

description
    "This statement defines nano service state characteristics for
     leaving this state.

    The component will advance to this state when it is back
    tracking, it has reached its following state, and the
    'pre-condition' is met.

    If the 'nano-callback' statement is defined, it means that
    there is a callback function (or template) that will be invoked
    before this state is left.

    The 'post-action-node' optionally defines an action to be
    invoked when this state has been left.";

}

extension nano-callback {
    description
        "This statement indicates that a callback function (or a
         template) is defined for this state and operation.";}

extension post-action-node {
    argument xpath {
        tailf:arg-type {
            type plan-xpath;
        }
    }
    tailf:substatement "description";
    tailf:substatement "ncs:action-name" {
        tailf:occurrence "1";
    }
    tailf:substatement "ncs:result-expr" {
        tailf:occurrence "?";
    }
    tailf:substatement "ncs:sync" {
        tailf:occurrence "?";
    }

description
    "This statement defined a action side-effect to be executed
     after the state has been successfully been executed.

    This statement argument is the node where the action resides.

    This action is executed asynchronously with respect to initial
    service transaction. The result is manifested as a value in
    the requesting plan states post-action-status leaf.

    The XPath expression is evaluated in the context described for
    'plan-xpath'.";
}

```

```

extension action-name {
    argument name {
        tailf:arg-type {
            type string;
        }
    }
    tailf:substatement "description";
    description
        "The name of the action.";
}

extension result-expr {
    argument xpath {
        tailf:arg-type {
            type yang>xpath1.0;
        }
    }
    tailf:substatement "description";
    description
        "An action responds with a structured result. A certain value
        can indicate an error or a successful result, e.g.,
        'result true'.

The result of the expression is converted to a boolean using
the standard XPath rules. If the result is 'true' the action
is reported as successful, otherwise as failed.

The context for evaluating this expression is the
resulting xml tree of the action.

There are no variable bindings in this evaluation.
The set of namespace declarations are all available namespaces,
with the prefixes defined in the modules.";
}

extension sync {
    description
        "Run the action synchronously so that later states cannot proceed before
        the post action has finished running successfully.";
}

extension force-commit {
    description
        "Force a commit before any later states can proceed.";
}

/*
 * Behavior tree extensions for nano services
 */

extension service-behavior-tree {
    argument servicepoint {
        tailf:arg-type {
            type tailf:identifier;
        }
    }
    tailf:occurrence "**";
    tailf:use-in "module";
    tailf:use-in "submodule";
    tailf:substatement "description";
    tailf:substatement "ncs:plan-outline-ref" {
        tailf:occurrence "1";
}

```

```

}
tailf:substatement "ncs:plan-location" {
    tailf:occurrence "?";
}
tailf:substatement "ncs:selector" {
    tailf:occurrence "*";
}
tailf:substatement "ncs:multiplier" {
    tailf:occurrence "*";
}
tailf:substatement "ncs:converge-on-re-deploy" {
    tailf:occurrence "?";
}
description
"This statement is used to define the behavior tree for a nano
service.

The argument to this statement is the name of the service point
for the nano service.

The behavior tree consists of control flow nodes and execution
nodes.

There are two types of control flow nodes, defined with the
'ncs:selector' and 'ncs:multiplier' statements.

There is one type of execution nodes, defined with the
'ncs:create-component' statement.

A behavior tree is evaluated by evaluating all top control flow
nodes, in order. When a control flow node is evaluated, it
checks if it should evaluate its children. How this is done
depend on the type of control flow node. When an execution
node is reached, the resulting component-type is added as a
component to the plan and given a component-name.

This process of dynamically instantiating a plan with its
components by evaluation of the behavior tree is called
synthesizing the plan.";
}

extension plan-outline-ref {
    argument id {
        tailf:arg-type {
            type tailf:identifier-ref;
        }
    }
    description
    "The name of the plan outline that the behavior tree will use
     to synthesize a service instance's plan.";
}

extension plan-location {
    argument path {
        tailf:arg-type {
            type yang>xpath1.0;
        }
    }
    description
    "XPath starting with absolute or relative path to a list or container
     where the plan is stored. Use this only if the plan is stored outside

```

the service.

The XPath expression is evaluated using the nano service as the context node, and the expression **must** return a node set.

If the target lies within lists, all keys **must** be specified. A **key** either has a **value**, or a **reference** to a **key** of the service using the function `current()` as starting point for an XPath location **path**. For example:

```

/a/b[k1='paul'][k2=current()/k]/c";
}

/* Control flow nodes */

extension selector {
    tailf:substatement "description";
    tailf:substatement "ncs:pre-condition" {
        tailf:occurrence "?";
    }
    tailf:substatement "ncs:observe" {
        tailf:occurrence "?";
    }
    tailf:substatement "ncs:variable" {
        tailf:occurrence "*";
    }
    tailf:substatement "ncs:selector" {
        tailf:occurrence "*";
    }
    tailf:substatement "ncs:multiplier" {
        tailf:occurrence "*";
    }
    tailf:substatement "ncs:create-component" {
        tailf:occurrence "*";
    }
}
description
"This control flow node synthesizes its children
that have their pre-conditions met.

All 'ncs:variable' statements in this statement will have their
XPath context node set to each node in the resulting node set.";
}

extension multiplier {
    tailf:substatement "description";
    tailf:substatement "ncs:pre-condition" {
        tailf:occurrence "?";
    }
    tailf:substatement "ncs:observe" {
        tailf:occurrence "?";
    }
    tailf:substatement "ncs:foreach" {
        tailf:occurrence "1";
    }
}
description
"This control flow node synthesizes zero or more copies of
its children.

When this node is evaluated, it evaluates the 'foreach'
expression. For each node in the resulting node set, it
synthesizes all children that have their pre-conditions
met.";
```

```

}

extension foreach {
    argument xpath {
        tailf:arg-type {
            type plan-xpath;
        }
    }
    tailf:substatement "description";
    tailf:substatement "ncs:when" {
        tailf:occurrence "?";
    }
    tailf:substatement "ncs:variable" {
        tailf:occurrence "*";
    }
    tailf:substatement "ncs:selector" {
        tailf:occurrence "*";
    }
    tailf:substatement "ncs:multiplier" {
        tailf:occurrence "*";
    }
    tailf:substatement "ncs:create-component" {
        tailf:occurrence "*";
    }
}

description
    This statement's argument is an XPath expression for the node set
    that is the basis for a multiplier selection. For each node in
    the resulting node set the children will be evaluated.

    The XPath expression is evaluated in the context described for
    'plan-xpath' .";
}

extension when {
    argument xpath {
        tailf:arg-type {
            type plan-xpath;
        }
    }
    tailf:substatement "description";

description
    This optional statement describes an XPath expression that is
    used to further filter the selection of nodes from the
    node set in a multiplier component or the variables that should
    be created for a component.

    The result of the expression is converted to a boolean using
    the standard XPath rules. If the result is 'true' the node is
    added to the node set or the variable is added to the variable
    list.

    The XPath expression is evaluated in the context described for
    'plan-xpath' .";
}

/* Execution nodes */

extension create-component {
    argument name {
        tailf:arg-type {

```

```

        type plan-xpath;
    }
}
tailf:substatement "description";
tailf:substatement "ncs:component-type-ref" {
    tailf:occurrence "1";
}
tailf:substatement "ncs:pre-condition" {
    tailf:occurrence "?";
}
tailf:substatement "ncs:observe" {
    tailf:occurrence "?";
}
tailf:substatement "ncs:deprecates-component" {
    tailf:occurrence "*";
}

description
"When this execution node is evaluated, it instantiates a component in the service's plan.

The name of the component is the result of evaluating the XPath expression and convert the result to a string.

The XPath expression is evaluated in the context described for 'plan-xpath' .";
}

extension component-type-ref {
    argument name {
        tailf:arg-type {
            type tailf:identifier-ref;
        }
    }
description
"This statement identifies the component type for the component. It must refer to a component-type defined in the plan-outline for the service.";
}

/* Common substatements */

extension pre-condition {
    tailf:substatement "description";
    tailf:substatement "ncs:monitor" {
        tailf:occurrence "?";
    }
    tailf:substatement "ncs:all" {
        tailf:occurrence "?";
    }
    tailf:substatement "ncs:any" {
        tailf:occurrence "?";
    }
description
"This statement defines a pre-condition that must hold for further evaluation/execution to proceed.

If the pre-condition is not satisfied a kicker will be created with the same monitor to observe the changes and then re-deploy the service.";
}

```

```

extension observe {
    tailf:substatement "description";
    tailf:substatement "ncs:monitor" {
        tailf:occurrence "1";
    }
    description
        "If a control flow node has been successfully evaluated, this
         statement's 'monitor' will be installed as a kicker, which will
         re-deploy the service if the monitor's trigger conditions are met.";
}

extension monitor {
    argument node {
        tailf:arg-type {
            type plan-xpath;
        }
    }
    tailf:substatement "description";
    tailf:substatement "ncs:trigger-on-delete" {
        tailf:occurrence "?";
    }
    tailf:substatement "ncs:trigger-expr" {
        tailf:occurrence "?";
    }
    description
        "If a node that matches the value of this statement and the
         'trigger' expression evaluates to true, this condition is
         satisfied. If the child statement 'trigger-on-delete' is used
         this condition will be satisfied when no nodes matches the
         value of this statement. Note that only one of 'trigger' and
         'trigger-on-delete' can be used as child statements to this
         statement.

The argument to this statement is like an instance-identifier,
but a list may be specified without any keys. This is treated
like a wildcard that matches all entries in the list.

The XPath expression is evaluated in the context described for
'plan-xpath' .";
}

extension trigger-on-delete {
    description
        "Specify if the monitored node should be checked if it has been deleted.";
}

extension trigger-expr {
    argument xpath {
        tailf:arg-type {
            type plan-xpath;
        }
    }
    tailf:substatement "description";

    description
        "This optional statement is used to further filter nodes
         in a given nodeset.

The result of the expression is converted to a boolean using
the standard XPath rules. If the result is 'true' the condition
is satisfied, otherwise it is not satisfied.

```

```

The XPath expression is evaluated in the context described for
'plan-xpath' .";
}

extension variable {
    argument name {
        tailf:arg-type {
            type string;
        }
    }

    tailf:substatement "description";
    tailf:substatement "ncs:when" {
        tailf:occurrence "?";
    }
    tailf:substatement "ncs:value-expr" {
        tailf:occurrence "?";
    }
}

description
"This statement defines an XPath variable with a name and a value. The value is evaluated as an XPath expression.

A variable called FOO can thus be retrieved as '{$FOO}'.

These variables can for example be used in a 'multiplier' control flow node to create unique names of duplicated components. The child components can be given names like 'comp_{$FOO}', and when that expression is evaluated, the resulting component will have a name with {$FOO} substituted with the value of the variable 'FOO' .";
}

extension value-expr {
    argument xpath {
        tailf:arg-type {
            type plan-xpath;
        }
    }

    tailf:substatement "description";

    description
"This statement defines an XPath expression that when evaluated constitutes a value for a variable.

The XPath expression is evaluated in the context described for
'plan-xpath' .";
}

extension deprecates-component {
    argument name {
        tailf:arg-type {
            type plan-xpath;
        }
    }

    tailf:substatement "ncs:component-type-ref" {
        tailf:occurrence "?";
    }
}

description

```

```

"Indicate that the component deprecates another deleted component and
that it will produce the same configuration as the old component. Before
running this component the reverse diffsets for all the old components
states will be applied and the component will be deleted.

If the old component is still present after synthesizing the behaviour
tree this statement will be ignored. ";
}

extension any {
    tailf:substatement "ncs:monitor" {
        tailf:occurrence "*";
    }
}

description
"This extension is used inside of pre-condition extensions
to allow multiple monitors inside of a single pre-condition.
A pre-condition using this extension is satisfied if at least
one of the monitors given as argument evaluates to true.

This extension uses short-circuit evaluation, i.e., if one of the
monitors given as argument evaluates to true the evaluation
will stop.";
}

extension all {
    tailf:substatement "ncs:monitor" {
        tailf:occurrence "*";
    }
}

description
"This extension is used inside of pre-condition extensions
to allow multiple monitors inside of a single pre-condition.
A pre-condition using this extension is satisfied if all
of the monitors given as argument evaluates to true.

This extension uses short-circuit evaluation, i.e., if one of
the monitors given as argument evaluates to false the evaluation
will stop.";
}

extension converge-on-re-deploy {
    status deprecated;
    description
"Do not converge a service in the transaction in which it is created.
On service creation the service will only synthesize the plan and
schedule a reactive-re-deploy of itself.

By default a service starts converging in the transaction in which
it is created, but in certain scenarios this might not be the desired
behaviour. E.g. when executing a service through the commit queue with
error recovery set to rollback on error, this will ensure that the
service intent is still present even when there are errors in the
commit queue.

Note: In a future release this behaviour will be the default and
this setting will be removed, hence the deprecated status.";
}
}

```

The ncs:plan-data grouping is defined as operational data that is supposed to be added to the Reactive FASTMAP service yang with a uses ncs:plan-data YANG directive.

A *plan* consists of one or many *component* entries. Each *component* has a name and a type. The type is an identityref and the service must therefore define identities for the types of components it uses. There is one predefined component type named *self* and a service with a plan is expected to have at least the *self* component defined.

Each component consists of two or more *state* entries where the state name is an identityref. The service must define identities for the states it wants to use. There are two predefined states *init* and *ready* and each plan component is expected to have *init* as its first state and *ready* as its last.

A state has a status leaf which can take one of the values *not-reached*, *reached* or *failed*.

The purpose of the *self* component is to show the overall progress of the Reactive FASTMAP service and the *self* component *ready* state should have status *reached* if and only if the service has completed successfully. All other components and states are optional and should be used to show the progress in more detail if necessary.

The plan should be defined and the statuses written inside the service `create()` method. Hence the same FASTMAP logic applies to the plan as for any other configuration data. This implies that the plan has to be defined completely at `create()` as if this was the first definition. If a service modification or `reactive-re-deploy` leave out a state or component, that has been earlier defined, this state or component will be removed.

When the status leaf in a component state changes value NSO will log the time of the status change in the *when* leaf. Furthermore when there is a structural changes of the plan, i.e added/removed components or states, NSO will log this in the *plan-history* list. The Reactive FASTMAP service need not and should not attempt doing this logging inside the `create` method.

A plan also defines an empty leaf *failed*. NSO will set this leaf when there exists states in the plan with status *failed*. As such this is an aggregation to make it easy to verify if a RFM service is progressing without problems.

In the Java API there exist a utility class to help writing plan data in the service `create` method. This class is called `PlanComponent` and has the following methods:

```
public class PlanComponent {

    /**
     * Creation of a plan component.
     * It uses a NavuNode pointing to the service. This is normally the same
     * NavuNode as supplied as an argument to the service create() method.
     *
     * @param service
     * @param name
     * @param componentType
     * @throws NavuException
     */
    public PlanComponent(NavuNode service,
                         String name,
                         String componentType) throws NavuException;

    /**
     * This method supplies a state to the specific component.
     * The initial status for this state can be ncs:reached or ncs:not-reached
     * and is indicated by setting the reached boolean to true or false
     * respectively
     *
     * @param stateName
     * @param reached
     * @return
     * @throws NavuException
     */
}
```

```

        */
    public PlanComponent append(String stateName) throws NavuException;

    /**
     * Setting status to ncs:not-reached for a specific state in the
     * plan component
     *
     * @param stateName
     * @return
     * @throws NavuException
     */
    public PlanComponent setNotReached(String stateName) throws NavuException;

    /**
     * Setting status to ncs:reached for a specific state in the plan component
     *
     * @param stateName
     * @return
     * @throws NavuException
     */
    public PlanComponent setReached(String stateName) throws NavuException;

    /**
     * Setting status to ncs:failed for a specific state in the plan component
     *
     * @param stateName
     * @return
     * @throws NavuException
     */
    public PlanComponent setFailed(String stateName) throws NavuException;
}

}

```

The constructor for the PlanComponent takes the service NavuNode from the `create()` method together with the component name and type. The type is either `ncs:self` or any other type defined as an identity in the service YANG module. The PlanComponent instance has an `append()` method to add new states which is either `ncs:init`, `ncs:ready` or any other state defined as an identity in the service YANG module. The `setNotReached()`, `setReached()` or `setFailed()` methods are used to set the current status of a given state.

Example: use of plan-data in the virtual-mpls-vpn

The following shows the use of plan-data in the `examples.ncs/service-provider/virtual-mpls-vpn` example. The objective in this example is to create and maintain a plan that has one main `self` component together with one component for each endpoint in the service. The endpoints can make use of either physical or virtual devices. If the endpoint uses a virtual device the corresponding plan component will contain additional states to reflect the staged setup of the virtual device.

In the service YANG file named `l3vpn.yang` we define the identity for the endpoint component type and the service specific states for the different components:

```

.....
identity l3vpn {
    base ncs:component-type;
}

identity pe-created {
    base ncs:plan-state;
}
identity ce-vpe-topo-added {

```

```

        base ncs:plan-state;
    }
    identity vpe-p0-topo-added {
        base ncs:plan-state;
    }
    identity qos-configured {
        base ncs:plan-state;
    }

    container vpn {
        list l3vpn {
            description "Layer3 VPN";

            key name;
            leaf name {
                tailf:info "Unique service id";
                tailf:cli-allow-range;
                type string;
            }

            uses ncs:plan-data;
            uses ncs:service-data;
            ncs:servicepoint l3vpn-servicepoint;

            ....
        }
    }
}

```

In the service list definition the plan data is introduced using the `uses ncs:plan-data` directive.

In the service `create()` method we introduce a Java Properties instance where we temporarily store data for the relevant Reactive FASTMAP steps that currently are completed. We create a private method `writePlanData()` that can write the plan with this Properties instance as input. Before we return from the `create()` method we call the `writePlanData()` method. The following code snippets from the class `l3vpnRFS.java` illustrates this design:

Initially we create a Properties instance called `rfmProgress`:

```

@ServiceCallback(servicePoint = "l3vpn-servicepoint",
                 callType = ServiceCBType.CREATE)
public Properties create(ServiceContext context,
                        NavuNode service,
                        NavuNode ncsRoot, Properties opaque)
throws ConfException
{
    WebUILogger.log(LOGGER, "***** Create/Reactive-re-deploy *****");
    Properties rfmProgress = new Properties();
}

```

For each Reactive FastMap step that we reach we store some relevant data in the `rfmProgress` Properties instance:

```

StringBuffer stb = new StringBuffer();
for (NavuContainer endpoint : endpoints.elements()) {
    if (stb.length() > 0) {
        stb.append(",");
    }
    stb.append(endpoint.leaf(l3vpn._id).valueAsString());
}
rfmProgress.setProperty("endpoints", stb.toString());

String tenant = service.leaf(l3vpn._name).valueAsString();
String deploymentName = "vpn";

```

```

String virtualPENName =
    Helper.makeDevName(tenant, deploymentName, "CSR", "esco");

for (NavuContainer endpoint : endpoints.elements()) {
    try {
        String endpointId = endpoint.leaf(l3vpn._id).valueAsString();

        String ceName = endpoint.leaf(l3vpn._ce_device).valueAsString();

        if (CEonVPE.contains(ceName)) {
            rfmProgress.setProperty(endpointId + ".ONVPE", "true");

            if (!createVirtualPE(context, service, ncsRoot, ceName, tenant,
                deploymentName)) {
                // We cannot continue with this CE until redeploy
                continue;
            }
            rfmProgress.setProperty(endpointId + ".pe-created", "DONE");
        }

        LOGGER.info("device ready, continue: " + virtualPENName);

        addToTopologyRole(ncsRoot, virtualPENName, "pe");

        // Add CE-VPE topology, reuse old topology connection if available
        NavuContainer conn = getConnection(
            topology, endpoint.leaf(l3vpn._ce_device).valueAsString(), "pe");
        if (!addToTopology(service, ncsRoot, conn,
            ceName, "GigabitEthernet0/8",
            virtualPENName, "GigabitEthernet2",
            ceName, Settings.ipPoolPE_CE)) {
            // We cannot continue with this CE until redeploy
            continue;
        }
        rfmProgress.setProperty(endpointId + ".ce-vpe-topo-added", "DONE");
    }
}

```

Before we return from the `create()` method we call the `writePlanData()` method passing in the `rfmProgress` instance:

```

writePlanData(service, rfmProgress);

return opaque;
}

```

The `writePlanData()` method first creates all components and sets the default values for all statuses. Then we read the `rfmProgress` instance and change the states for the all the Reactive FASTMAP steps that we have reached. In the end we check if the self component `ready` state has been reached. The reason for initially writing the complete plan with default values for the statuses is not to miss a component that have not made any progress yet, remember this is FASTMAP, components and states that was written in an earlier `reactive-re-deploy` but is not written now will be deleted by NSO. The `writePlanData()` method has the following design:

```

private void writePlanData(NavuNode service, Properties rfmProgress)
    throws NavuException {
    try {

        PlanComponent self = new PlanComponent(service, "self", "ncs:self");
        // Initial plan
        self.appendState("ncs:init").
            appendState("ncs:ready");
        self.setReached("ncs:init");
    }
}

```

```

String eps = rfmProgress.getProperty("endpoints");
String ep[] = eps.split(",");

boolean ready = true;
for (String p : ep) {
    boolean onvpe = false;
    if (rfmProgress.containsKey(p + ".ONVPE")) {
        onvpe = true;
    }
    PlanComponent pcomp = new PlanComponent(service,
                                              "endpoint-" + p,
                                              "13vpn:13vpn");

    // Initial plan
    pcomp.appendState("ncs:init");
    pcomp.setReached("ncs:init");
    if (onvpe) {
        pcomp.appendState("13vpn:pe-created");
        appendState("13vpn:ce-vpe-topo-added");
        appendState("13vpn:vpe-p0-topo-added");
    }
    pcomp.appendState("13vpn:qos-configured");
    appendState("ncs:ready");

    boolean p_ready = true;
    if (onvpe) {
        if (rfmProgress.containsKey(p + ".pe-created")) {
            pcomp.setReached("13vpn:pe-created");
        } else {
            p_ready = false;
        }
        if (rfmProgress.containsKey(p + ".ce-vpe-topo-added")) {
            pcomp.setReached("13vpn:ce-vpe-topo-added");
        } else {
            p_ready = false;
        }
        if (rfmProgress.containsKey(p + ".vpe-p0-topo-added")) {
            pcomp.setReached("13vpn:vpe-p0-topo-added");
        } else {
            p_ready = false;
        }
    }
    if (rfmProgress.containsKey(p + ".qos-configured")) {
        pcomp.setReached("13vpn:qos-configured");
    } else {
        p_ready = false;
    }

    if (p_ready) {
        pcomp.setReached("ncs:ready");
    } else {
        ready = false;
    }
}

if (ready) {
    self.setReached("ncs:ready");
}
} catch (Exception e) {
    throw new NavuException("could not update plan.", e);
}
}

```

Running the example and showing the plan while the chain of `reactive-re-deploy` is still in execution could look something like the following:

NAME	TYPE	STATE	STATUS	WHEN
self	self	init	reached	2016-04-08T09:22:40
		ready	not-reached	-
endpoint-branch-office	13vpn	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	13vpn	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

To support Service Progress Monitoring (SPM) there are two YANG groupings, `ncs:service-progress-monitoring-data` and `ncs:service-progress-monitoring-trigger-action`.

For an example of using service progress monitoring, see [getting-started/developing-with-ncs/25-service-progress-monitoring](#) The YANG submodule defining service progress monitoring including the groupings is named `tailf-ncs-service-progress-monitoring.yang` and contains the following:

```

submodule tailf-ncs-service-progress-monitoring {
    yang-version 1.1;
    belongs-to tailf-ncs {
        prefix ncs;
    }

    import ietf-yang-types {
        prefix yang;
    }
    import tailf-common {
        prefix tailf;
    }

    include tailf-ncs-plan;

    organization "Tail-f Systems";

    description
        "This submodule contains a collection of YANG definitions for
        Service Progress Monitoring (SPM) in NCS.

        Copyright 2018 Cisco Systems, Inc.
        All rights reserved.
        Permission is hereby granted to redistribute this file without
        modification.";

    revision 2018-06-01 {
        description
            "Initial revision";
    }

/*

```

```

* Plan Component State
*/
identity any-state {
    description
        "Can be used in SPM and plan trigger policies to denote any plan state.";
    base ncs:plan-state;
}

/*
 * Plan Component Types
*/
identity any {
    description
        "Can be used in SPM and plan triggers to denote any component type.";
    base ncs:plan-component-type;
}

/*
 * Groupings
*/
typedef spm-trigger-status {
    type enumeration {
        enum passed {
            tailf:code-name spm-passed;
        }
        enum failed {
            tailf:code-name spm-failed;
        }
    }
}

grouping service-progress-monitoring-trigger-action {
    tailf:action timeout {
        description
            "This action should be used by a custom model that is separate
            from the service (which may be made by someone else),
            and it must be refined with an actionpoint.

            Any callback action to be invoked when SPM trigger must
            always have the five leaves defined as input to this action
            as initial arguments, they are populated by the NSO system.";
        input {
            leaf service {
                description
                    "The path to the service.";
                type instance-identifier;
                mandatory true;
            }
            leaf trigger {
                description "The name of the trigger that fired.";
                type leafref {
                    path "/ncs:service-progress-monitoring/ncs:trigger/ncs:name";
                }
                mandatory true;
            }
            leaf policy {

```

```

description "The name of the policy that fired.";
type leafref {
    path "/ncs:service-progress-monitoring/ncs:policy/ncs:name";
}
mandatory true;
}

leaf timeout {
    description "What timeout has triggered.";
    type enumeration {
        enum violation {tailf:code-name spm-violation-timeout;}
        enum jeopardy {tailf:code-name spm-jeopardy-timeout;}
        enum success {tailf:code-name spm-success-timeout;}
    }
    mandatory true;
}

leaf status {
    description "SPM passed or failed.";
    type spm-trigger-status;
    mandatory true;
}
}

grouping service-progress-monitoring-data {
    container service-progress-monitoring {
        config false;

        description
            "Service Progress Monitoring triggers.
A service may have multiple SPMs.
For example, if a CPE is added at a later stage it would have
its own SPM defined, separate from the main SPM of the service.
However, in many cases there will be just one SPM per service.

The overall status for a trigger can be determined by reading
the trigger-status{name}/status leaf. The success-time
leaf will be set when the policy evaluates to true, i.e. when
that part of the product is considered to be delivered by the
policy expression. Note that this is operational data.
";

        list trigger-status {
            description
                "The operation status of the trigger.';

            key name;

            leaf name {
                type string;
                description
                    "The trigger name.";
            }

            leaf policy {
                type string;
                description
                    "Name of policy.";
            }
        }
    }
}

```

```
leaf start-time {
    type yang:date-and-time;
    tailf:cli-value-display-template "${./datetime}";
    description
        "Time when the triggers started ticking.";
}

leaf jeopardy-time {
    type yang:date-and-time;
    tailf:cli-value-display-template "${./datetime}";
    description
        "Time when the conditions are evaluated for a jeopardy trigger.";
}

leaf jeopardy-result {
    type spm-trigger-status;
    description
        "The result will be 'passed' if no jeopardy was detected at
        jeopardy-time, 'failed' if it was detected. It is not set until
        it has been evaluated. It will be set to 'passed' if the
        condition is satisfied prior to the timeout expiring as well.";
}

leaf violation-time {
    type yang:date-and-time;
    tailf:cli-value-display-template "${./datetime}";
    description
        "Time when the conditions are evaluated for a violation trigger.";
}

leaf violation-result {
    type spm-trigger-status;
    description
        "The result will be 'passed' if no violation was detected at
        violation-time, 'failed' if it was detected. It is not set until
        it has been evaluated. It will be set to 'passed' if the
        condition is satisfied prior to the timeout expiring as well.";
}

leaf status {
    type enumeration {
        enum running {
            tailf:code-name spm-running;
            description
                "Service Progress Monitoring has been started but
                not yet triggered";
        }
        enum jeopardized {
            tailf:code-name spm-jeopardized;
            description
                "The jeopardy timer has triggered and the policy has evaluated
                to false.";
        }
        enum violated {
            tailf:code-name spm-violated;
            description
                "The violation timer has triggered and the policy has evaluated
                to false.";
        }
        enum successful {
            tailf:code-name spm-successful;
            description
                "The success timer has triggered and the policy has evaluated
                to false.";
        }
    }
}
```

```

        "One of the timers have triggered and the policy has evaluated
        to true.";
    }
}

leaf success-time {
    type yang:date-and-time;
    tailf:cli-value-display-template "${./datetime}";
    description
        "Time when the conditions were evaluated to true,
        i.e SPM was successful.";
}
}

container service-progress-monitoring {
    tailf:info "Service Progress Monitoring policies";

    list policy {
        tailf:info "Policy definitions for Service Progress Monitoring";
        description
            "A list of all the policies.';

        key name;
        leaf name {
            type string;
            description
                "The name of the policy.";
        }

        leaf violation-timeout {
            tailf:info "Violation timeout in seconds";
            mandatory true;
            type uint32;
            units "seconds";
            description
                "The timeout in seconds for a policy to be violated.";
        }

        leaf jeopardy-timeout {
            tailf:info "Jeopardy timeout in seconds";
            mandatory true;
            type uint32;
            units "seconds";
            description
                "The timeout in seconds for a policy to be in jeopardy.";
        }

        list condition {
            min-elements 1;
            description
                "A list of the conditions that decides whether a policy is
                fulfilled or not.";
        }

        key name;
        leaf name {
            type string;
            description
                "Name of the condition.";
        }
    }
}

```

```

}

list component-type {
    min-elements 1;

    description
        "Each condition can specify what state must be reached for
         a portion of the components to not trigger the action below./";

    key type;

    leaf type {
        description
            "We can either specify a particular component name
             (trigger/component) or a component-type (which may
             exist in several instances).";
        type union {
            type ncs:plan-component-type-t;
            type enumeration {
                enum "component-name" {
                    tailf:code-name spm-component-name;
                }
            }
        }
    }

    leaf what {
        description
            "Condition put on the component with respect to the
             ../plan-state and ../status.

So, either:

        1. X % of the component states has the status set.

        2. All of the component states has the status set.

        3. At least one of the components states has the status set.

        ";
        mandatory true;
        type union {
            type uint32 {
                range "0..100";
            }
            type enumeration {
                enum all{
                    tailf:code-name spm-what-all;
                }
                enum at-least-one {
                    tailf:code-name spm-what-at-least-one;
                }
            }
        }
    }

    leaf plan-state {
        mandatory true;
        type ncs:plan-state-name-t;
        description
            "The plans state. init, ready or any specific for the
             component.";
    }
}

```

```

leaf status {
    type ncs:plan-state-status-t;
    default "reached";
    description
        "status of the new state for the component in the service's plan.
         reached not-reached or failed.";
}
}

container action {
    leaf action-path {
        type instance-identifier {
            require-instance false;
        }
    }
    leaf always-call {
        type boolean;
        default "false";
        description
            "If set to true, the action will be invoked also when
             the condition is evaluated to 'passed'.";
    }
}
}

list trigger {
    description
        "A list of all the triggers. A trigger is used to apply a SPM policy
         to a service.";

    key name;

    leaf name {
        type string;
        description
            "Name of the trigger.";
    }

    leaf description {
        type string;
        description
            "Service Progress Monitoring trigger description.";
    }

    leaf policy {
        tailf:info "Service Progress Monitoring Policy";
        mandatory true;
        description
            "A reference to a policy that should be used with this trigger.";
        type leafref {
            path "/ncs:service-progress-monitoring/policy/name";
        }
    }

    leaf start-time {
        type yang:date-and-time;
        tailf:cli-value-display-template "${./datetime}";
        description
            "Optionally provide a start-time.
             If this is unset the SPM server will set the start-time to
             now.";
    }
}
}

```

```
        the commit time of the trigger.";  
    }  
  
    leaf component {  
        type string;  
        description  
            "If the policy contains a condition with the key component-name,  
            this is the component to apply the condition to.";  
    }  
  
    leaf target {  
        mandatory true;  
        description  
            "Instance identifier to whichever service the SPM policy should  
            be applied. Typically this is the creator of the trigger instance.";  
        type instance-identifier {  
            require-instance true;  
        }  
    }  
}  
}
```

Performance Considerations

When using the Reactive FASTMAP technique the service tends to be re-deployed multiple times for the service to be fully deployed; i.e., the `create()` function is executed more frequently. This makes it desirable to reduce the execution time of the `create()` function as much as possible.

Normal code performance optimization methods should be used, but there are a couple of techniques that can be used that are specific to the Reactive FASTMAP pattern.

- 1 Stacked services (see the section called “Stacked Services and Shared Structures”) can be a very efficient technique to reduce both the size of the service diff-set and the execution time.
For example, if a service applies a template to configure a device, then all changes resulting from this will be stored in the diff-set of the service. During a re-deploy all changes will first be undone to later be restored when the template is applied.

A more efficient solution is to use a stacked service to apply the template. The input parameters to the stacked service will be the variables that would go into the template. The stacked service would pick them up and apply the original template. As a consequence the diff-set resulting from applying the template ends up in the stacked service, and as long as there are no changes in the input parameter to the stacked service its `create()` code will not have to run. Instead of applying the same template multiple times the template will only be applied once.

- 2 CDB subscriber refactoring. Stacked services can be used when no response is required from the factored out code. However, if the `create()` code contains a CPU intensive computation that takes a number of input parameters and produce some result, then it would be desirable to also minimize the number of times this computation is performed, and to perform it outside the database lock.

This can be done by treating the problem similarly to resource allocation above - create a configuration tree where computation requests can be written. A CDB subscriber is registered to subscribe to this tree. Whenever a new request is committed it performs the computation and writes the result into a CDB operational data leaf, and re-deploys the service that requested the computation.

As a consequence of this the computation will take place outside the lock, and the computation will only be performed once for each set of input parameters. The cost of this technique is that an extra redeploy will be performed. The service pseudo-code looks like this:

```

create(serv) {
    /* request computation */
    create("/compute-something{id}");
    setElem("/compute-something{id}/param1", value1);
    setElem("/compute-something{id}/param2", value2);

    /* check for allocation response */
    if (!exists("/compute-something{id}/response"))
        return;

    /* read result */
    res = getElem("/compute-something{id}/response");

    /* use res in device config */
    configure(res)
}

```

Services that involve virtual devices, NFV

Virtual devices are increasingly popular and it is very convenient to dynamically start them from a service. However, a service that starts virtual devices has a number of issues to deal with, for example: how to start the virtual device, how to stop it, how to react to changes in the device status (for example scale in/scale out), and how to allocate and free VM licenses.

A device cannot both be started and configured in the same transaction since it takes some time for the device to be started, and it also needs to be added to the device tree before a service can configure it.

The Reactive FASTMAP pattern is ideally suited for the task of dealing with the above issues.

Starting a virtual machine

Starting a virtual device is a multi-step process consisting of:

- 1 Instructing a VIM or VNF-M to start the virtual device with some input parameters (which image, cpu settings, day0 configuration etc).
- 2 Waiting for the virtual device to be started, the VIM/VNF-M may signal this through some event, or polling of some state might be necessary.
- 3 Mount the device in the NSO device tree.
- 4 Fetch ssh-keys and perform sync-from on the newly created device.

The device is then ready to actually be configured.

There are several ways to achieve the above process with Reactive FASTMAP. One solution is implemented in the `vm-manager` and `vm-manager-esc` packages found in the example `examples.ncs/service-provider/virtual-mpls-vpn`.

Using these packages the service does not directly talk to the VIM/VNF-M but instead registers a `vm-manager/start` request using the `vm-manager` API. This is done by adding a list instance in the `/vm-manager/start` list.

The contract with the `vm-manager` is that it should be responsible for starting the virtual device, adding it to the `/devices/device` tree, perform `sync-from`, setting the `/devices/device/vmm:ready` leaf to `true`, and finally re-deploy the service that made the start request. This greatly simplifies the implementation of the service that would otherwise have to perform all those operations itself.

The `vm-manager` package is only an interface package. It must be combined with a package that actually talks to the VIM/VNF-M. In the `virtual-mpls-vpn` example this is done through a package called

`vm-manager-esc` that interfaces with a VNF-M called ESC. The `vm-manager-esc` package subscribes to changes in the `/vm-manager/start` configuration tree provided by the `vm-manager` package. Whenever a new request is created in that tree it attempts to start the corresponding VM on the indicated ESC device.

When the `vm-manager-esc` package receives a CREATE event in the `/vm-manager/start` list it initiates starting the VM. This involves a number of steps and components. In addition to the CDB subscriber for the `/vm-manager/start` tree it also has the following parts.

- 1 A CDB subscriber (`notif` subscriber) that subscribes to NETCONF notifications from the ESC device. NETCONF notification are used to communicate the state of the virtual machine. Events are sent when a new VM is registered, when it is started, when it has become alive, when it stops etc. The `vm-manager-esc` package needs to react differently to the different events, and ignore some of them.
- 2 A local service (`vm-manager/esc`) for starting the VM on the ESC. The CDB subscriber that subscribes to the `/vm-manager/start` list will create new instances of this service whenever a new `vm-manager/start` entry is received, and delete the corresponding service when a `vm-manager/start` entry is deleted. The reason the CDB subscriber doesn't configure the ESC directly is that it would then have to keep track of what to delete when the `vm-manager/start` entry is deleted, but perhaps more importantly, if resources should be allocated, for example a management IP, then this can done conveniently from inside a service using the resource manager package.

The `vm-manager/esc` service writes configuration to the ESC device to start a new VM, to monitor it, and to send NETCONF notifications on state changes. It may also perform resource allocation and other activities.

When the `notif` subscriber receives a VM ALIVE event it mounts the device in the device tree, performs fetch ssh keys and sync-from, sets the ready leaf to true, and re-deploys the service that requested the VM. The user of the ready leaf is critical. The original service cannot just inspect the devices tree to see if the device is there. The device being in the devices tree is no guarantee for it being ready to configure.

Stopping a virtual machine

Stopping a virtual machine is almost as complicated as starting it. If a service both starts a virtual device and configures it there will be a problem when the service is deleted (or when the service is reconfigured to not start the virtual device). When the service is deleted the configuration that the service has created will be deleted, including both the configuration to start the VM and the configuration on the VM.

If the service configured the VIM/VNF-M directly the result would be that the VIM/VNF-M would be told to stop the VM at the same time as NSO is trying to change the configuration on the VM (deleting the configuration that the service created). This results in a race condition that frequently results in an error (the VM is spun down while NSO is talking to it, trying to delete its configuration).

This problem is handled by using the `vm-manager` package between the service and the VIM/VNF-M. When a service is deleted the `vm-manager/start` configuration is deleted. This in turn will trigger the CDB subscriber to stop the service, but this will be done after the service delete transaction has been completed, and consequently after NSO has removed the configuration that the service created on the device. The race condition is avoided.

Another problem is how to remove the device from the NSO device tree. A service that directly configures the VIM/VNF-M would have to use some trick to deal with this. The `vm-manager-esc` package can handle this directly in the CDB `vm-manager/start` subscriber. When it registers a delete of a `vm-manager/start` instance it deletes the corresponding `vm-service`, but also the devices that have been mounted. If scaling is supported by the VIM/VNF-M there might be multiple entries in the NSO device tree that must be deleted. The `vm-manager` YANG model contains a list of names of all devices mounted as a response to a `vm-`

manager/start request. This list can be read both by the initiating service, but also by the vm-manager-esc CDB subscriber to know which devices to delete.

How to handle Licenses

Licences for virtual machines may or may not be a problem. If a license server is used the VM has to register for a license when it is started. This procedure would typically consist of some configuration that the initiating service would apply, or it can be part of the day0 config, in which case it is applied when the VM is started.

The real problem is usually to de-register a license. When a VM is stopped it is desirable to release the license if a license server is used. This process typically consists of deleting some configuration on the device and then waiting for the device to talk to the license server.

This complicates the device delete process a bit. Not only should the device be stopped but it must be a staged process where the device config first is removed, then the license released, and then, when the device has actually released the license, instruct the VIM/VNF-M to stop the device.

There are at least two solutions to this problem, with slightly different trade-offs.

- 1 The device NED is modified to deal with license release such that when it receives a license delete command, it detects this and waits until the license has actually been released before returning. This assumes that the license was applied as part of the device configuration that the initial service applied. The drawback of this approach is that the commit may be slow since it will delay until the license has been released. The advantage is that it is easy to implement.
- 2 The specific vm-manager package, vm-manager-esc in our example, could be modified to release the license before instructing the VIM/VNF-M to stop the VM. This is more efficient, but also a bit more complicated. The CDB subscriber that listens to vm-manager/start modifications would detect a DELETE operation and before removing the device from the NSO device tree it would invoke a license release action on the device. The NED implementing this action (as a NED command) would release the license and then wait until the device has actually released the license before returning. The CDB subscriber would then proceed to delete the device from the NSO device tree, and the vm-service instance. This whole procedure could be spawned off in a separate thread to avoid blocking other vm-manager/start operations.

Advanced Mapping Techniques

Create Methods

What happens when several service instances share a resource that may or may not exist before the first service instance is created? If the service implementation without any distinction just checks if the resource exists and creates it if it doesn't, then the create will be stored in the first created service instance's reversed diff. This implies that if the first instance is removed, then the shared resource is also be removed with it, leaving all other service instances without the shared resource.

A solution to this problem is the `sharedCreate()` and `sharedSet()` functionality that is part of both the Maapi and Navu APIs. The `sharedCreate()` method is used to create data in the `/ncs:devices/device` tree that may be shared by several service instances. Everything that is created gets a reference counter associated to it. With this counter the FASTMAP algorithm can keep track of the usage and only delete data when the last service instance referring to this data is removed. Furthermore, everything that is created using the `sharedCreate()` method also gets an additional attribute set, called "Backpointer", which points back to the service instance that created the entity in the first place.

This makes it possible to look at the /devices tree and answer the question which parts of the device configuration was created by which service(s)

In the examples.ncs/getting-started/developing-with-ncs/4-rfs-service example there is a *vlan* package that uses the shared create functionality:

```
//Now we will need to iterate over all of our managed
//devices and do a shareCreate of the interface and the unit

//Get the list of all managed devices.
NavuList managedDevices = root.container("devices").list("device");

for(NavuContainer deviceContainer : managedDevices.elements()){

    NavuContainer ifs = deviceContainer.container("config").
        container("r", "sys").container("interfaces");

    // execute as shared create of the path
    // /interfaces/interface[name='x']/unit[name='i']

    NavuContainer iface =
        ifs.list("interface").sharedCreate(
            vlan.leaf("iface").value());
    iface.leaf("enabled").sharedCreate();

    NavuContainer unit = iface.
        list("unit").sharedCreate(
            vlan.leaf("unit").value());

    unit.leaf("vlan-id").sharedSet(vlan.leaf("vid").value());
    unit.leaf("enabled").sharedSet(new ConfBool(true));
    unit.leaf("description").sharedSet(
        vlan.leaf("description").value());
    for (ConfValue arpValue : vlan.leafList("arp")) {
        unit.leafList("arp").sharedCreate(arpValue);
    }
}
```

Build the example and create two services on the same interface:

```
$ cd $NCS_DIR/examples.ncs/getting-started/developing-with-ncs/4-rfs-service
$ make clean all
$ ncs-netsim start
$ ncs
$ ncs_cli -C -u admin
admin@ncs# configure
admin@ncs(config)# devices sync-from
admin@ncs(config)# services vlan s1 iface ethX unit 1 vid 1 description descr1
admin@ncs(config-vlan-s1)# commit
admin@ncs(config-vlan-s1)# top
admin@ncs(config)# services vlan s2 iface ethX unit 2 vid 2 description descr2
admin@ncs(config-vlan-s2)# commit
admin@ncs(config-vlan-s2)# top
```

We can now look at the device data for one of the relevant devices. We are especially interested in the *RefCount* and the *Backpointer* attributes that are used by the NSO FASTMAP algorithm to deduce when the data is eligible for deletion:

```
admin@ncs(config)# show full-configuration devices device ex0 \
    config r:sys interfaces interface | display service-meta-data
...
/* Refcount: 2 */
```

```
/* Backpointer: [ /ncs:services/vl:vlan[vl:name='s1'] /ncs:services/vl:vlan[vl:name='s2'] ] */
r:sys interfaces interface ethX
...
```

If we now delete the first service instance, the device interface still exists, but with a decremented reference counter:

```
admin@ncs(config)# no services vlan s1
admin@ncs(config)# commit
admin@ncs(config)# show full-configuration devices device ex0 \
    config r:sys interfaces interface | display service-meta-data
...
/* Refcount: 1 */
/* Backpointer: [ /ncs:services/vl:vlan[vl:name='s2'] ] */
r:sys interfaces interface ethX
...
```

Persistent FASTMAP Properties

In the service application there are often cases where the code needs to make some decision based upon something that cannot be derived directly from the input parameters to the service.

One such example could be allocating an new IP address. Suppose the implementation keeps a list of allocated addresses in the configuration. When the `create()` code runs, it finds a free address, stores it in the list, and uses it in the device configuration. This would work and for a while it would seem that all is well, but there is a fallacy in this implementation. The problem is that if a service is modified, the FASTMAP algorithm first removes all settings of the original create, including also the IP allocation. When the `create()` method is called to recreate the service there is no guarantee that the implementation will find the same free address in the list. This implies that a simple update of any service model leaf may change the allocated IP address.

NSO has built-in support to prevent this. What it comes down to is that the service implementation need to have persistently stored properties for each service instance that can be used in conjunction with the FASTMAP algorithm. These properties are found in the `opaque` argument in the Java API service interface.

```
public Properties create(ServiceContext context,
                        NavuNode service,
                        NavuNode root,
                        Properties opaque)
```

The `opaque` properties object is made available as an argument to the service `create()` method. When a service instance is first created this object is `null`. The code can add properties to it, and it returns the possibly updated `opaque` object, which NSO stores with the service instance. Later when the service instance is updated NSO will pass the stored `opaque` to `create()`.



Note

It is vital that the `create()` returns the `opaque` object that was passed to it, even if the method itself does not use it. The reason for this is that, as we will see in the section called “[Pre and post hooks](#)” the `create()` method is not the only callback that uses this `opaque` object. The `opaque` object can actually be chained in several different callbacks. Having a `return null;` in the `create()` method is not good practice.

A pseudo code implementation of our IP allocation scenario could then look something like the following:

```
@ServiceCallback(servicePoint="my-service",
    callType=ServiceCBType.CREATE)
```

```

public Properties create(ServiceContext context,
                        NavuNode service,
                        NavuNode root,
                        Properties opaque)
    throws DpCallbackException {
    String allocIP = null;
    if (opaque != null) {
        allocIP = opaque.getProperty("ALLOCATED_IP");
    }

    if (allocIP == null) {
        // This implies that the service instance is created for the first
        // time the allocation algorithm should execute

        ...

        // The allocated IP should be stored in the opaque properties
        if (opaque == null) {
            opaque = new Properties();
        }
    }

    ...

    opaque.setProperty.setProperty("ALLOCATED_IP", allocIP);
}

// It is important that the opaque Properties object is returned
// or else it will not be stored together with the service instance
return opaque;
}

```

Pre and post hooks

There are scenarios where some handling in a service implementation still should be outside the scope of the FASTMAP algorithm. For instance when a service require some functionality in a device to be enabled and should make settings to enable this functionality if it is not enabled already. If that enabling should still be set on the device even though the service interface is later removed, then the FASTMAP comes short in doing this.

For this reason there are two extra methods in the `DpServiceCallback` interface, `preModification` and `postModification` that if registered will be called before respectively after the FASTMAP algorithm modifies device data:

```

@ServiceCallback(servicePoint = "",
                callType = ServiceCBType.PRE_MODIFICATION)
public Properties preModification(ServiceContext context,
                                  ServiceOperationType operation,
                                  ConfPath path,
                                  Properties opaque)
    throws DpCallbackException;

@ServiceCallback(servicePoint = "",
                callType = ServiceCBType.POST_MODIFICATION)
public Properties postModification(ServiceContext context,
                                   ServiceOperationType operation,
                                   ConfPath path,
                                   Properties opaque)
    throws DpCallbackException;

```

The `pre/postModification` methods have an `context` argument of type `ServiceContext` which contains methods to retrieve `NavuNodes` pointing to the service instance and the ncs model rootnode. Data that

is modified using these NavuNodes will be handled outside the scope of the FASTMAP algorithm and therefore untouched by changes of the service instance (if not changed in another pre/postModification callback):

```
public interface ServiceContext {
    ...
    public NavuNode getServiceNode() throws ConfException;
    public NavuNode getRootNode() throws ConfException;}
```

The pre/postModification also has an operation argument of enum type `ServiceOperationType` which describes the type of change that current service instance is subject to:

```
public enum ServiceOperationType {
    CREATE,
    UPDATE,
    DELETE;
    ...
}
```

In addition to the above arguments the pre/postModification methods also has an path argument that points to the current service instance and the opaque Properties object corresponding to this service instance. Hence the opaque can be first created in a preModification methods, passed to, and modified, in the FASTMAP `create()` method and in the end also handled in a postModification method before stored with the service instance.

The `examples.ncs/getting-started/developing-with-ncs/15-pre-modification` example show how a preModification method can be used to permanently set a dns server in the device configuration. This dns server is thought of as a prerequisite for the service instances and should always be set for the devices. Instead of having to fail in the fastmap service when the prerequisite is not fulfilled. The preModification can instead check and set the config. We have the following preModification code:

```
@ServiceCallback(servicePoint = "vpnep-servicepoint",
                 callType = ServiceCBType.PRE_MODIFICATION)
public Properties preModification(ServiceContext context,
                                  ServiceOperationType operation,
                                  ConfPath path,
                                  Properties opaque)
throws DpCallbackException {
    try {
        vpnev vep = new vpnev();
        if (ServiceOperationType.DELETE.equals(operation)) {
            return opaque;
        }

        // get the in transaction changes for the current
        // service instance
        NavuNode service = context.getRootNode().container(Ncs._services_).
            namespace(vpnep.id).list(vpnep._vpn_endpoint_).
            elem((ConfKey) path.getKP()[0]);
        List<NavuNode> changedNodes = service.getChanges(true);

        for (NavuNode n : changedNodes) {
            if (n.getName().equals(vpnep._router_)) {
                NavuLeaf routerName = (NavuLeaf) n;
                NavuNode deviceNameNode = routerName.deref().get(0);
                NavuContainer device =
                    (NavuContainer) deviceNameNode.getParent();

                String routerNs = "http://example.com/router";
                NavuContainer sys = device.container(Ncs._config_).

```

```

        namespace(routerNs).container("sys");

        NavuList serverList = sys.container("dns").list("server");

        if (!serverList.containsNode("10.10.10.1")) {
            serverList.create("10.10.10.1");
        }
        break;
    }
}
} catch (Exception e) {
    throw new DpCallbackException("Pre modification failed", e);
}
return opaque;
}

```

We walk through this example code and explain what it does. The first part is a check of which operation is being performed. If the operation is a delete we can return. We always return the opaque passed to us as an argument. Even though this is a delete it is not necessarily the last callback in the callback chain, if we would return null we would impose a null opaque to later callbacks.

```

if (ServiceOperationType.DELETE.equals(operation)) {
    return opaque;
}

```

Next we need check if the *router* leaf of the service has changed in the transaction. This leaf is mandatory, but if the operation is an *UPDATE* then this leaf is not necessarily changed. The following code snippet navigates to the relevant service instance NavuNode and get the list of all changed NavuNodes in this transaction and for this service instance:

```

NavuNode service = context.getRootNode().container(Ncs._services_);
namespace(vpnev.id).list(vpnev._vpn_endpoint_).
    elem((ConfKey) path.getKP()[0]);
List<NavuNode> changedNodes = service.getChanges(true);

```

We check if any of the changed NavuNodes is the *router* leaf which is of type leafref to a device name under the /ncs:devices/device tree:

```

for (NavuNode n : changedNodes) {
    if (n.getName().equals(vpnev._router_)) {
        NavuLeaf routerName = (NavuLeaf) n;
    }
}

```

If the *router* leaf has changed, since it is an leafref to another leaf we can deref it and get the device name leaf in the /ncs:devices/device tree. Note that in the general case a deref will not necessarily return a singular NavuNode, but in this case it will and therefore we can just call `get(0)` on the deref list of NavuNodes. We want to get the device container NavuNode and we can retrieve this as the parent node of the *deviceName* leaf.

```

NavuNode deviceNameNode = routerName.deref().get(0);
NavuContainer device =
    (NavuContainer) deviceNameNode.getParent();

```

We now know that the router leaf has changed, we have the device container NavuNode for this device and we can check the device configuration for the dns servers. If the IP address 10.10.10.1 does not appear in the list we add it.

```

String routerNs = "http://example.com/router";
NavuContainer sys = device.container(Ncs._config_).
    namespace(routerNs).container("sys");

```

```

NavuList serverList = sys.container("dns").list("server");

if (!serverList.containsNode("10.10.10.1")) {
    serverList.create("10.10.10.1");
}

```

We have here used the preModification callback to hardwire a enabling for a service. This setting will stay on the device independently of the lifecycle changes of the service instance which created it.

Stacked Services and Shared Structures

It is possible for one high level service to create another low level service instance. In this case, the low level service is FASTMAPed similar to how the data in /ncs:devices/device is FASTMAPed when a normal RFS manipulates the device tree. We can imagine a high level service (maybe a customer facing service, CFS) called *email* that in its turn creates real RFS services *pop* and/or *imap*.

The same principles apply on the FASTMAP data when services are stacked as in the regular RFS service scenario. The most important principle is that the data created by a FASTMAP service is owned by the service code. Regardless of whether we use a template based service or a Java based service, the service code *creates* data, and that data is then associated with the service instance. If the user deletes a service instance, FASTMAP will automatically delete whatever the service created, including any other services. Thus, if the operator directly manipulates data that is created by a service, the service becomes "out of sync". Each service instance has a "check-sync" action associated to it. This action checks if all the data that the service creates or writes is still there.

This is especially important to realize in the case of stacked services. In this case the low level service data is under the control of the high level service. It is thus forbidden to directly manipulate that data. Only the high level service code may manipulate that data. NSO has no built-in mechanism that detects when data created by service code is manipulated "out of band".

However, two high level services may manipulate the same structures. Regardless of whether an RFS creates data in the device tree or if a high level service creates low level services (that are true RFS services), the data created is under the control of FASTMAP and the attributes Refcount and Backpointer that automatically gets created are used by FASTMAP to ensure that structures shared by multiple service instances are not deleted until there are no users left.

FASTMAP pre-lock create option



Note This option is deprecated. It is recommended to use the FASTMAP `create()` function in all cases.

The original FASTMAP algorithm accepts concurrent transactions. However the `create()` function will be called after acquiring a common transaction lock. This implies that only one service instance's `create()` function is called at the time.



Note This above serialization of the transaction is part of the NSO service manager's FASTMAP algorithm. It should NOT be mistaken for the NSO device managers propagation of data to the relevant devices, which is performed at a later stage of the transaction commit. The latter is performed in a fully concurrent manner.

The reasons for the serialization of FASTMAP transactions are transaction consistency and making it simpler to write `create()` functions, since they do not need to be thread-safe.

However in certain scenarios with services where the `create()` function requires heavy computation and in the same time have no overlap in written data, this serialization is not necessary and will prevent higher throughput. For this reason a `preLockCreate()` function has been introduced. This function serves exactly the same purpose as the `create()` function but is called before the common transaction lock is acquired.

The guidelines for using a `preLockCreate()` function instead of the ordinary `create()` are:

- The service creation is computationally heavy, i.e., consumes substantial CPU time.
- The service creation can be coded in a thread-safe fashion.
- Different service instances has no config data overlap, or the probability for config data overlap is low.

The `preLockCreate` FASTMAP algorithm has internal detection of conflicting concurrent transaction data updates. This implies that there is no risk of persistent data inconsistencies but instead a conflicting transaction might fail in commit.

For services that also uses the `preModification()` function, this function will also be called before the transaction lock if `preLockCreate()` is used.

If a stacked service (see [the section called “Stacked Services and Shared Structures”](#)) has a `preLockCreate()`, and the stacked service is created by another service's `create()` function, then the stacked service's `preLockCreate()` will be called inside the lock.

Service Caveats

Under some circumstances the mapping logic of a service needs special consideration. Services can either map to disjunctive data sets or shared data sets.

If the services map to disjunctive data sets, which means no other service will manipulate the same data, there are no known caveats.

If on the other hand several services manipulate the same data there are some things to consider. All these special cases will be discussed below.

Finding Caveats

A useful tool for finding potential problems with overlapping data is the CLI `debug service` flag. Example:

```
admin@ncs(config)# commit dry-run | debug service
```

The `debug service` flag will display the net effect of the service create code as well as issue warnings about potential problematic usage. Note these warnings are only for situations where services have overlapping shared data.

In all examples below the WARNING message is the result of using the flag `debug service`.

delete

A general rule of thumb is to never use `delete` in service create code.

If a `delete` is used in service create code the following warning is displayed:

```
*** WARNING ***: delete in service create code is unsafe if data is
shared by other services
```

The deleted elements will be restored when the service instance which did the delete is deleted. Other services which relied on the same configuration will be out of sync.

The explicit delete is easy to detect in the XML of a template or in the Java source code. The not so easy detection are the `when` and `choice` statements in the YANG data model.

If a `when` statement is evaluated to false the configuration tree below that node will be deleted.

If a `case` is set in a `choice` statement the previously set `case` will be deleted.

Both the above `when` and `case` scenarios will behave the same as an explicit delete.

One working design pattern for these use cases is to let one special init service be responsible for the deletion and initialization. This init service should be a singleton and be shared created by other services depending on the specific delete and initialization.

By using this stacked service design the other services just share create that init service. When the last one of the other services is deleted the init service is also deleted as it is reference counted.

Another design pattern is to have such delete and initialization code in the pre- and post-modification code of a service. This is possible but generally results in more complex code than the stacked service approach above.

set

If a set operation instead of a shared set operation is used in service create code the following warning is displayed:

```
*** WARNING ***: set in service create code is unsafe if data is
shared by other services
```

The set operation does not add the service meta-data reference count to the element. If the first service, which set the element, is deleted the original value will be restored and other services will be out of sync.

create

If a create operation instead of a shared create operation is used in service create code the following warning is displayed:

```
*** WARNING ***: create in service create code is unsafe if data is
shared by other services
```

The create operation does not add the service meta-data back-pointer and reference count to the element. If the first service, which created the element, is deleted the created item is deleted and other services will be out of sync.

move

If items in an ordered by user list are moved and these items were created by another service the following warning is displayed:

```
*** WARNING ***: due to the move the following services will be
out of sync:
```

and a list of affected services is listed.

Moving items which other services relies on is a service design flaw. This has to be analyzed and taken care of in user code.

Service Discovery

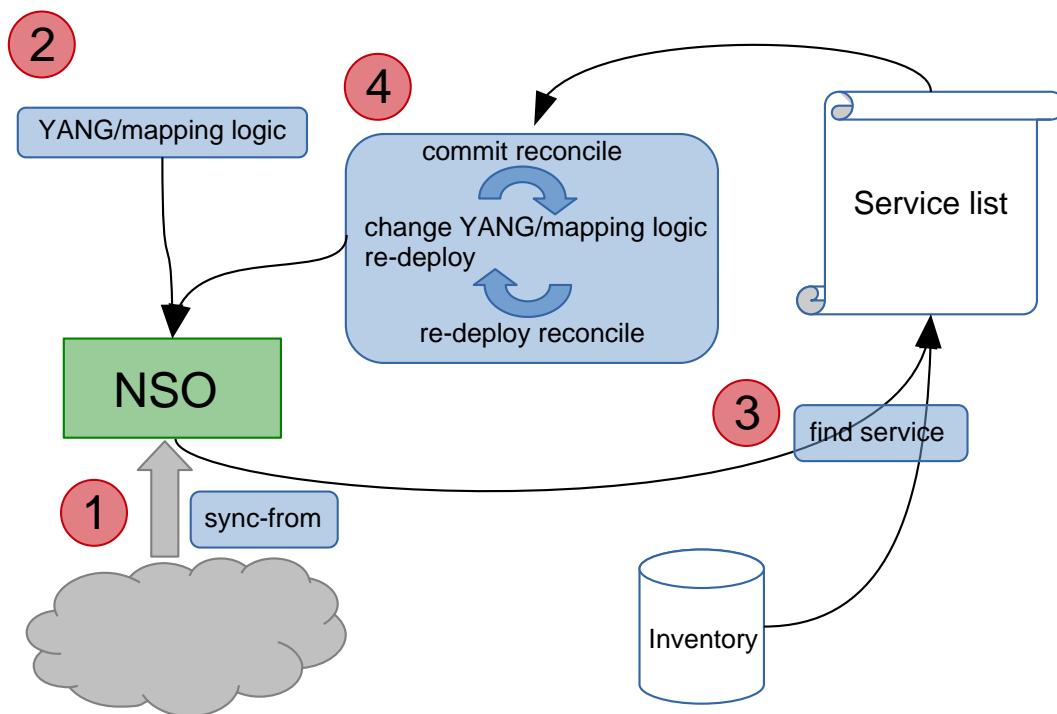
Discovery basics

A very common situation when NSO is deployed in an existing network is that the network already has services implemented. These services may have been deployed manually or through an older provisioning system. The task is to introduce NSO and 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.

The whole process of identifying services and importing them into NSO is called *Service Discovery*. Some steps in the process can be automated others are highly manual. The amount of work differs a lot depending on how structured and consistent the original deployment is.

The process can be broken down in a number of steps:

Figure 165. Service Discovery



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 must also be possible to:

- 1 Import all managed devices into NSO;
- 2 Write the YANG data model for the service and the the mapping logic.
- 3 Write a program, using Python/Maapi or Java/Maapi which traverses the entire network configuration and computes the services list.
- 4 Verify the mapping logic is correct.

The last step, verifying the mapping logic is an iterative process. The goal is to ensure all relevant device configuration is covered by the mapping logic.

Verifying the mapping logic is achieved by using the action **re-deploy reconcile { } dry-run** of NSO. When the output is empty the data is covered.

NSO uses special attributes on instance data to indicate the data used by a service. Two attributes are used for this *RefCount* and *Backpointer*.

By using the flag **display service-meta-data to show full-configuration** these attributes can be inspected.

Even if all data is covered in the mapping there might still be manual configured data below service data. If this is not desired use the action **re-deploy reconcile { discard-non-service-config } dry-run** to find such configuration.

Below the steps to reconcile a service are shown, first in a visual form and later as commands in one of the examples.

Figure 166. Service v1 and v2 has been created with original data O



The service *v1* and *v2* has been created on top the existing original data.

The service *v1* has sole control of the instance data in α , which is not part of δ , and service *v2* has sole control of the instance data β , which is not part of ϵ .

The data solely owned by service *v1* and *v1* has a reference count of one.

The data in δ and in ϵ is both part of the original data and part of service data. The reference counter in these areas is two.

If the service *v1* was to be deleted the data with reference count of one would be removed. The data in δ would be kept but the reference count would be removed.

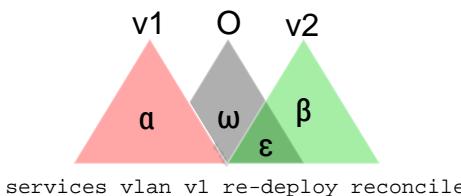
After throughly inspection of the service and the affected data the service can become the sole owner of the data which is part of the original data.

Check the effect of the reconciliation by use of the **dry-run** option to **re-deploy**:

```
admin@ncs(config)# services vlan v1 re-deploy reconcile { } dry-run
```

The output of the **dry-run** will only display configuration changes, not changes in service-meta-data like reference count and back-pointers.

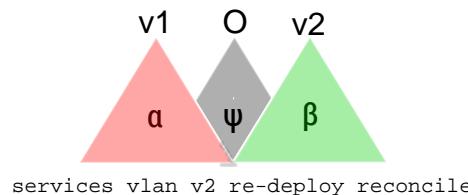
Figure 167. Reconcile Service v1



After reconciliation of $v1$ the service is the sole owner of the data α . All the data in α now has the reference count set to one after the operation.

Complete the process by reconciling service $v2$ as well.

Figure 168. Reconcile Service v2



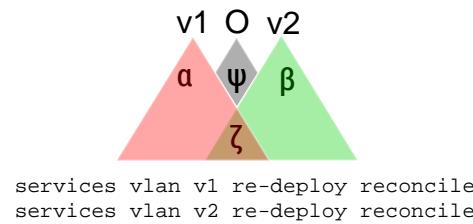
All data in α and β now has a reference count of one and will thus be removed when services $v1$ and $v2$ are removed or un-deployed.

If at later stage it shows parts of ψ should belong to a service just change the mapping logic of the service and execute the action again:

```
admin@ncs(config)# services vlan v1 re-deploy reconcile
admin@ncs(config)# services vlan v2 re-deploy reconcile
```

If the service mapping logic is changes so services starts to overlap each other and start to control more of the original data like in the following figure:

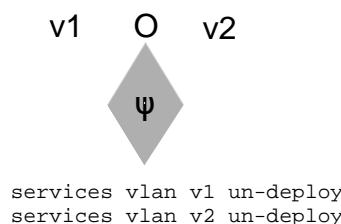
Figure 169. Overlapping services



just reconcile the services again. After reconciliation α and β has a reference count of one and the reference count of ζ is two.

The command **re-deploy reconcile** can be executed over and over again, if the service is already reconciled nothing will happen.

Figure 170. Un-deploy Service v1 and v2



The data ψ is outside any service and is kept after the services are gone. If the services $v1$ and $v2$ had been deleted ψ would still look the same.

Now after the visualization try this by hand in one of the examples: `examples.ncs/getting-started/developing-with-ncs/4-rfs-service`

First we create two service instances:

```
$ cd $NCS_DIR/examples.ncs/getting-started/developing-with-ncs/4-rfs-service
$ make clean all
$ ncs-netsim start
$ ncs
$ ncs_cli -C -u admin
admin@ncs# config
admin@ncs(config)# devices sync-from
admin@ncs(config)# services vlan v1 description v1-vlan iface eth1 unit 1 vid 111
admin@ncs(config-vlan-v1)# top
admin@ncs(config)# services vlan v2 description v2-vlan iface eth2 unit 2 vid 222
admin@ncs(config-vlan-v2)# top
admin@ncs(config)# commit
```

That created two services in the network. Now let's destroy that.

```
admin@ncs(config)# devices device * delete-config
admin@ncs(config)# no services
admin@ncs(config)# commit no-networking
```

We now have a situation with two services deployed in the network, but no services, nor any device configuration in NSO.

This is the case when NSO first is set up in a network. Now start by getting all the device data into the data base.

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

This resembles the point were brown field deployment starts. Lets introduce the two service instances in NSO.

```
admin@ncs(config)# services vlan v1 description v1-vlan iface eth1 unit 1 vid 111
admin@ncs(config-vlan-v1)# top
admin@ncs(config)# services vlan v2 description v2-vlan iface eth2 unit 2 vid 222
admin@ncs(config-vlan-v2)# top
admin@ncs(config)# commit no-networking
```

We're almost there now. If we take a look at the deployed configuration in NSO, we see for example:

```
admin@ncs(config)# show full-configuration devices device ex0 \
    config r:sys interfaces | display service-meta-data
...
! Refcount: 2
! Backpointer: [ /ncs:services/vl:vlan[vl:name='v1'] ]
r:sys interfaces interface eth1
! Refcount: 2
! Backpointer: [ /ncs:services/vl:vlan[vl:name='v1'] ]
enabled
! Refcount: 2
! Backpointer: [ /ncs:services/vl:vlan[vl:name='v1'] ]
unit 1
! Refcount: 2
! Originalvalue: true
enabled
! Refcount: 2
```

```

    ! Originalvalue: vl-vlan
description vl-vlan
! Refcount: 2
! Originalvalue: 111
vlan-id      111
!
!
! Refcount: 2
! Backpointer: [ /ncs:services/vl:vlan[vl:name='v2'] ]
r:sys interfaces interface eth2
! Refcount: 2
! Backpointer: [ /ncs:services/vl:vlan[vl:name='v2'] ]
enabled
! Refcount: 2
! Backpointer: [ /ncs:services/vl:vlan[vl:name='v2'] ]
unit 2
! Refcount: 2
! Originalvalue: true
enabled
! Refcount: 2
! Originalvalue: v2-vlan
description v2-vlan
! Refcount: 2
! Originalvalue: 222
vlan-id      222
!
!
```

When we commit a service to the network, the FASTMAP code will create the *Refcount* and the *Backpointer* attributes. These attributes are used to connect the device config to services. They are also used by FASTMAP when service instances are changed or deleted. In the configuration snippet above you can see the interface "eth1" and "eth2" has a refcount of 2 but only one back-pointer, pointing back to the services. This is the state when the data is not owned by the service but still is part of the original data..

```
admin@ncs(config)# services vlan v1 re-deploy reconcile
admin@ncs(config)# services vlan v2 re-deploy reconcile
```

Now the services *v1* and *v2* are in the same state as in the figure: [Figure 168, “Reconcile Service v2”](#) above.

```
admin@ncs(config)# show full-configuration devices device ex0 \
    config r:sys interfaces | display service-meta-data
...
! Refcount: 1
! Backpointer: [ /ncs:services/vl:vlan[vl:name='v1'] ]
r:sys interfaces interface eth1
! Refcount: 1
! Backpointer: [ /ncs:services/vl:vlan[vl:name='v1'] ]
enabled
! Refcount: 1
! Backpointer: [ /ncs:services/vl:vlan[vl:name='v1'] ]
unit 1
! Refcount: 1
enabled
! Refcount: 1
description vl-vlan
! Refcount: 1
vlan-id      111
!
!
! Refcount: 1
! Backpointer: [ /ncs:services/vl:vlan[vl:name='v2'] ]
r:sys interfaces interface eth2
```

Reconciliation caveats

```

! Refcount: 1
! Backpointer: [ /ncs:services/vl:vlan[vl:name='v2' ] ]
enabled
! Refcount: 1
! Backpointer: [ /ncs:services/vl:vlan[vl:name='v2' ] ]
unit 2
! Refcount: 1
enabled
! Refcount: 1
description v2-vlan
! Refcount: 1
vlan-id    222
!
!
```

The two services *v1* and *v2* have been reconciled. The reference counter as well as the back pointers are correct and indicates the data is owned by the services.

Reconciliation caveats

This scheme works less well sometimes depending on the service type. If the services *delete* data on the managed devices expecting FASTMAP to recreate that data when the service is removed, this technique doesn't work.

Also, if the service instances have allocated data, this scheme has to be modified to take that allocation into account.

A reconcile exercise is also a cleanup exercise, and every reconciliation exercise will be different.

Reconciling in bulk

Once we have convinced ourselves that the reconciliation process works, we probably want to reconcile all services in bulk. One way to do that would be to write a shell script to do it. The script needs input; assume we have a file *vpn.txt* that contains all the already existing VPNs in the network as a CSV file.

```
$ cat vpn.txt
volvo,volvo VLAN,eth4,1,444
saab,saab VLAN,eth4,2,445
astra,astra VLAN,eth4,3,446
```

A small shell script to generate input to the CLI could look like

```
#!/bin/sh
infile=$1
IFS=,
echo "config" > out.cli
cat $infile |
while read id d iface unit vid; do
    c="services vlan $id description \"$d\" iface $iface unit $unit vid $vid"
    echo $c >> out.cli
    echo "top" >> out.cli
done

echo "commit" >> out.cli

cat $infile |
while read id desc iface unit vid; do
    echo "Reconcile of '$id'"
    echo "services vlan $id re-deploy reconcile" >> out.cli
done
```

```

echo exit >> out.cli
echo exit >> out.cli

ncs_cli -C -u admin < out.cli

```

Partial sync

In some cases a service may need to rely on the actual devices configuration to compute the changeset. It is often a requirement to pull the current devices' configurations from the network before executing such service. Doing a full sync-from on a number of devices is an expensive task especially if it needs to be performed often, so the suggested way in this case is using partial-sync-from.

In cases where multitude of service instances touch a device that is not entirely orchestrated using NSO, i.e. relying on the partial-sync-from feature described above, and the device needs to be replaced then all services need to be re-deployed. This can be expensive depending on the number of service instances. Partial-sync-to enables replacement of devices in a more efficient fashion.

Partial-sync-from and partial-sync-to actions allow to specify certain portions of the devices' configuration to be pulled or pushed from or to the network, respectively, rather than the full config. These are more efficient operations on NETCONF devices and NEDs that support partial-show feature. NEDs that do not support partial-show feature will fall back to pulling or pushing the whole configuration.

Even though partial-sync-from and partial-sync-to allows to pull or push only a part of device's configuration, the actions are not allowed to break consistency of configuration in CDB or on the device as defined by the YANG model. Hence extra consideration needs to be given to dependencies inside the device model. If some configuration item A depends on configuration item B in the device's configuration, pulling only A may fail due to unsatisfied dependency on B. In this case both A and B need to be pulled, even if the service is only interested in the value of A.

It is important to note that partial-sync-from and partial-sync-to does not update the transaction ID of the device, when pushing to the device, or NSO, when pulling from the device, unless the whole configuration has been selected (e.g. /ncs:devices/ncs:device[ncs:name='ex0']/ncs:config).

Partial sync-from

Pulling the configuration from the network needs to be initiated outside the service code. At the same time the list of configuration subtrees required by a certain service should be maintained by the service developer. Hence it is a good practice for such service to implement a wrapper action that invokes the generic /devices/partial-sync-from action with the correct list of paths. The user or application that manages the service would only need to invoke the wrapper action without needing to know which parts of configuration the service is interested in.

The snippet in [Example 171, “Example of running partial-sync-from action via Java API”](#) gives an example of running partial-sync-from action via Java, using "router" device from examples.ncs/getting-started/developing-with-ncs/0-router-network.

Example 171. Example of running partial-sync-from action via Java API

```

ConfXMLParam[] params = new ConfXMLParam[] {
    new ConfXMLParamValue("ncs", "path", new ConfList(new ConfValue[] {
        new ConfBuf("/ncs:devices/ncs:device[ncs:name='ex0']/"
            + "ncs:config/r:sys/r:interfaces/r:interface[r:name='eth0']/"
            + "new ConfBuf("/ncs:devices/ncs:device[ncs:name='ex1']/"
            + "ncs:config/r:sys/r:dns/r:server")
    })
}

```

```
        })),  
        new ConfXMLParamLeaf( "ncs" , "suppress-positive-result" ));  
    ConfXMLParam[] result =  
        maapi.requestAction(params, "/ncs:devices/ncs:partial-sync-from");
```



CHAPTER 15

Templates

- [Introduction, page 347](#)
- [Config templates, page 347](#)
- [Basic principles, page 350](#)
- [Service Templates, page 362](#)
- [Templates applied from an API, page 370](#)
- [Service and API Templates in multi-NED environment, page 373](#)

Introduction

Templates is a flexible and powerful mechanism in NSO which simplifies how changes can be made across the configuration data, for example across devices of different type. They also allow a declarative way to describe such manipulations.

Two types of Templates exist, *device-templates* and *config-templates*. The former is invoked as an action, where the latter is invoked either because of changes to some service data or through a programmatic API (e.g Java). For more information about *device-templates*, refer to: the section called “Device Templates” in *NSO 5.7 User Guide*. The rest of the text in this chapter mainly describes *config-templates*.



Note

config-templates are often called “Service Templates”, but this is only partly true. A *config-template* invoked through an API does not have to deal with NSO services.

When a template is used as part of a service implementation, thanks to NSO FASTMAP, NSO remembers the configuration changes made towards the devices and the template changes can for example be reverted.

There exist an API for applying config templates so they can replace large portions of boilerplate code when configuration data needs to be manipulated from a programming language.

config-templates gets loaded as part of packages. The *config-templates* are stored in XML files in the `templates` sub directory of a package. *config-templates* can be used to update any part of the configuration.

Config templates

Config template terminology:

Create a Config-template from a Device-template

- *Config template*: The use of templates in a programmatic way. Based on the same basic principles as the *device-template* with the distinction that a config template is part of the implementation (device-templates on the other hand, are dynamically created by the operator as needed, for example, in the CLI and stored in the configuration). *config-templates* gets loaded when NSO starts. If a NSO package has a `templates` sub directory it gets scanned for files with the suffix '`.xml`' and those files gets loaded. With *config-templates* one can implement full services or abstract functionality. *config-templates* makes it possible to divide the work when implementing a service. One group may focus on the logic and the programming and another focus on the networking details declaring the modifications of devices in a template.
- *Service template*: A config-template connected to a service-point. These are services which are entirely implemented with a config-template. A service template is useful in cases when only a mapping of service instance data to device data is needed to implement the service, and no logic is involved. See [the section called “Service Templates”](#).
- *Feature Template*: A config-template used to configure a specific feature (for example acl) on a set of - different - devices. It is a convenient way of updating structured configuration data and can save lots of boiler plate code. Feature templates are normally applied from service code. See `$NCS_DIR/examples.ncs/service-provider/mpls-vpn/packages/13vpn/templates/13vpn-acl.xml` which is used in the `13vpn` service, in `mpls-vpn` example.

The typical way to design a template is to start with direct manipulation of devices until the desired result is reached or to write a *device-template* and apply it and watch the changes, correct it until the result is correct. This process can be performed in the NSO CLI or in the WebUI. Below will show how to do this in the CLI.

The process of defining a *config-template* is illustrated in two ways:

- 1 Define a device-template and convert it to a config-template
- 2 Define a config-template based on direct device configuration snippets.

both methods will result in the same *config-template* [Example 172, “Created Config-template 13vpn-ce.xml”](#) being constructed.



Note

The name of the *config-template* is the name of the file without the extension `.xml`. All the template names reside in the same namespace. A good practice is to name the file: `<package name>-<feature>.xml`. That way the name of the template will always be unique.

Create a Config-template from a Device-template

The idea is to start off by writing a *device-template*, apply it and inspect the changes to the devices, modify the template and do the same over again until the desired result is reached.

Create a device template `13vpn-ce`, add some settings to it and commit the changes:

```
admin@ncs(config)# devices template 13vpn-ce config ios:interface GigabitEthernet 0/1
admin@ncs(config)# description "Link to PE"
admin@ncs(config)# ip address primary address 10.1.1.1 mask 255.255.255.252
```

Apply the *device-template* to the `ce0` device and check the changes by doing a dry run using the native output format of the device:

```
admin@ncs(config)# devices device ce0 apply-template template-name 13vpn-ce
admin@ncs(config)# commit dry-run outformat native
native {
    device {
```

```

        name ce0
        data interface GigabitEthernet0/1
            description Link to PE
            ip address 10.1.1.1 255.255.255.252
            exit
        }
    }
}

```

If you spot an error, revert the changes done by the template, add the missing configuration to the template and commit.

When you are satisfied with the result, save the *device-template* to a file:

```
admin@ncs(config)# show full-configuration devices template l3vpn-ce | display xml | save l3vpn-ce.xml
```

Pick up the file `l3vpn-ce.xml` in any editor. Change the beginning of the file from:

```
<config xmlns="http://tail-f.com/ns/config/1.0">
<devices xmlns="http://tail-f.com/ns/ncs">
<template>
    <name>l3vpn-ce</name>
```

to:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0">
<devices xmlns="http://tail-f.com/ns/ncs">
<device>
    <name>ce0</name>
```

Note the name of the device, it just serves as an example. The device name can be a static value, an XPath variable or the result of an XPath expression.

At the end the XML elements needs to be closed, change from:

```
    </template>
</devices>
</config>
```

to:

```
    </device>
</devices>
</config-template>
```

The final result should look like [Example 172, “Created Config-template l3vpn-ce.xml”](#).

Create a Config-template from a device configuration

The idea is to make direct changes to the configuration and dry-run the changes to confirm the right commands are sent to the devices. From these changes a template can, as shown below, be constructed.

Make changes:

```
admin@ncs(config)# devices device ce0 config ios:interface GigabitEthernet 0/1
admin@ncs(config)# description "Link to PE"
admin@ncs(config)# ip address 10.1.1.1 255.255.255.252
```

Check that modifications of the network is the desired. Here is what is actually sent to the device by using the native format:

```
admin@ncs(config)# commit dry-run outformat native
native {
    device {
```

```

        name ce0
        data interface GigabitEthernet0/1
            description Link to PE
            ip address 10.1.1.1 255.255.255.252
            exit
    }
}

```

The same changes in the XML format serves as a good skeleton for a template. Save the modifications to a file:

```
admin@ncs(config)# commit dry-run outformat xml | save l3vpn-ce.xml
```

pick up the file l3vpn-ce.xml in any text editor. Remove the leaf name `result-xml`. Insert the template specific tag at the beginning of the file:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0"
    <devices xmlns="http://tail-f.com/ns/ncs">
```

and to close the XML put at the end of the file:

```
</devices>
</config-template>
```

The final result should look like [Example 172, “Created Config-template l3vpn-ce.xml”](#).

Example 172. Created Config-template l3vpn-ce.xml

```
<config-template xmlns="http://tail-f.com/ns/config/1.0">
    <devices xmlns="http://tail-f.com/ns/ncs">
        <device>
            <name>ce0</name>
            <interface xmlns="urn:ios">
                <GigabitEthernet>
                    <name>0/1</name>
                    <description>Link to PE</description>
                    <ip>
                        <address>
                            <primary>
                                <address>10.1.1.1</address>
                                <mask>255.255.255.252</mask>
                            </primary>
                        </address>
                    </ip>
                </GigabitEthernet>
            </interface>
        </device>
    </devices>
</config-template>
```

This above skeleton is now ready to be edited, have expressions introduced instead of static values to make it more useful. Look at `$NCS_DIR/examples.ncs/service-provider/mpls-vpn/packages/l3vpn/templates/l3vpn-ce.xml` to see how the above skeleton could be extended. [the section called “Service Templates”](#) Shows how Config-template is used as a service template and [the section called “Templates applied from an API”](#) shows how a more complex Config-template is used from Java API.

Basic principles

A template is declared in accordance with the YANG data model. You just set and create node elements to build the desired structure. Or paraphrased, you simply declare the result.

A template can be defined across different vendors, but still in the same template. Furthermore, the templates also allow for defining different behaviour when applying the template. This is accomplished by setting tags such as `merge`, `replace`, `delete`, `create` or `nocreate` on relevant nodes in the template.

Values in a template



Note The variables `$DEVICE` and `$TEMPLATE_NAME` are set internally by NSO. `$DEVICE` is set to the name of the current device. The variable `$TEMPLATE_NAME` is set to the name of the current template. None of these variables can be set by a user, it can however be used in a template as any other variable.

The variable `$SCHEMA_OPAQUE` is set internally if the template is registered for the servicepoint (the top node in the template has `servicepoint` attribute) and the corresponding `ncs:servicepoint` statement in the YANG model has `tailf:opaque` substatement. The variable is then set to the argument of the `tailf:opaque` statement for the corresponding schema path.

Each value in a template is stored as a string. This string value is converted to the actual value type of the YANG model when the template is applied.

If the value contains a pair of `{ . . . }` the string between the curly braces is treated as an XPath 1.0 expression. The simplest form of an XPath expression is a plain XPath variable:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>{$CE}</name>
      ...
    </device>
  </devices>
</config-template>
```

Any value assigned to the variable `CE`, which can be done via the Java API, will be used when the template is applied:

```
...
TemplateVariables var = new TemplateVariables();
var.putQuoted("CE", "ce0");
...
```

A value can contain any number of `{ . . . }` and strings, the total result will be the concatenation of all the strings and all the XPath expressions.

```
<config-template xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>{$CE}</name>
      ...
      <config>
        <interface xmlns="urn:ios">
          <GigabitEthernet>
          ...
          <description>Link to PE / {$PE} - {$PE_INT_NAME}</description>
        </device>
      </config>
    </devices>
</config-template>
```

This would for example, when applied, evaluate the XPath expressions `{$PE}` and `>{$PE_INT_NAME}` to "pe0" and "GigabitEthernet0/0/0/3". giving the the total result of

```
<description>Link to PE / pe0 - GigabitEthernet0/0/0/3</description>
```

As the text between the { . . . } is an XPath 1.0 expression we can select any reachable node:

```
/endpoint/ce/device
```

this selects a leaf node, `device`. The value of the selected leaf will be used to assign a new value where it is used when the template is applied.

The result of a selection is internally converted to string representation. All concatenations are done on strings. When the string value is assigned to a leaf element a string to value conversion takes place in the context of the target node.

If the result of the selection is an empty node-set *nothing* is set.

Processing instructions

NSO template engine supports a number of processing instructions to allow for more dynamic templates. The following table lists the available processing instructions.

Table 173. Template processing instructions

Syntax	Description
<code><?set variable = value?></code>	Allows to assign new variables or manipulate existing variable value. If used to create a new variable, then the scope of visibility of this variable is limited to the parent tag of the processing instruction or the current processing instruction block. Specifically, if a new variable is defined inside a loop, then it is discarded at the end of each iteration.
<code><if {expression}?>... <?elif {expression}?>... <?else?>...<?end?></code>	Processing instruction block that allows conditional execution based on the boolean result of the expression. For the detailed description see the section called “Conditional Statements”
<code><foreach {expression}?>...<?end?></code>	The expression must evaluate to a (possibly empty) XPath node-set. The template engine will then iterate over each node in the node-set by changing the XPath current context node to this node and evaluating all children tags within this context. For the detailed description see the section called “Loop Statements”
<code><for [variable = initial value]; {progress condition}; [variable = next value]?>...<?end?></code>	This processing instruction allows to iterate over the same set of template tags by changing a variable value. The variable visibility scope obeys the same rules as in the case of <code>set</code> processing instruction, except the variable value is carried over to the next iteration instead of being discarded at the end of each iteration. The square brackets indicate optional clauses, so only the condition expression is mandatory. For the detailed description see the section called “Loop Statements”
<code><copy-tree {source}?></code>	This instruction is analogous to <code>copy_tree</code> function available in the MAAPI API. The parameter is an XPath expression which must evaluate to exactly one node in the data tree and indicates the source path to copy from. The target path is defined by the position of the <code>copy-tree</code> instruction in the template within the current context.

Syntax	Description
<code><?set-context-node {expression}?></code>	Allows to manipulate the current context node used to evaluate XPath expressions in the template. The expression is evaluated within the current XPath context and must evaluate to exactly one node in the data tree.
<code><?set-root-node {expression}?></code>	Allows to manipulate the root node of the XPath accessible tree. This expression is evaluated in an XPath context where the accessible tree is the entire datastore, which means that it is possible to select a root node outside the current accessible tree. The current context node remains unchanged. Just like with the <code>set-context-node</code> instruction the expression must evaluate to exactly one node in the data tree.
<code><?save-context name?></code>	Store both the current context node and the root node of the XPath accessible tree with <code>name</code> being the key to access it later. It is possible to switch to this context later using <code>switch-context</code> with the name. Multiple contexts can be stored simultaneously under different names. Using <code>save-context</code> with the same name multiple times will result in the stored context being overwritten.
<code><?switch-context name?></code>	Used to switch to a context stored using <code>save-context</code> with the specified name. This means that both the current context node and the root node of the XPath accessible tree will be changed to the stored values. <code>switch-context</code> does not remove the context from the storage and can be used as many times as needed, however using it with a name that does not exist in the storage would cause an error.
<code><?if-ned-id ned-ids?>... <?elif-ned-id ned-ids?>... <?else?>... <?end?></code>	If there are multiple versions of the same NED known or expected to be loaded in the system, defining different versions of the same namespace, this processing instruction helps to resolve ambiguities in the schema between different versions of the NEDs. The part of the template following this processing instruction, and until matching <code>elif-ned-id</code> , <code>else</code> or <code>end</code> processing instruction are only applied to devices with the ned-id matching one of the ned-ids specified as a parameter to this processing instruction. If there are no ambiguities to resolve, then this processing instruction is not required. The <code>ned-ids</code> must contain one or more qualified ned-id identities separated by spaces. <code>elif-ned-id</code> is optional and used in combination with <code>if-ned-id</code> processing instruction to define a part of template that applies to devices with another set of ned-ids than previously specified. Multiple <code>elif-ned-id</code> instructions are allowed in a single block of <code>if-ned-id</code> instructions. The set of ned-ids specified as a parameter to <code>elif-ned-id</code> instruction must be non-intersecting with the previously specified ned-ids in this block. <code>else</code> processing instruction should be used with care in this context, as the set of the ned-ids it handles depends on the set of ned-ids loaded in the system, which can be hard to predict at the time of developing the template. To mitigate this problem it is recommended that the package containing this template defines a set of supported-ned-ids as described in the section called “Service and API Templates in multi-NED environment” .

The variable value in both `set` and `for` processing instructions is evaluated in the same way as the values within XML tags in a template (see [the section called “Values in a template”](#)). So, it can be a mix of literal values and XPath expressions surrounded by {...}.

The variable value is always stored as a string, so any XPath expression will be converted to literal using XPath `string()` function. Namely, if the expression results in an integer or a boolean, then the resulting value would be a string representation of the integer or boolean. If the expression results in a node-set, then the value of the variable is a concatenated string of values of nodes in this node-set.

It is important to keep in mind that while in some cases XPath converts the literal to another type implicitly (for example, in an expression `{ $x < 3 }` a possible value `x='1'` would be converted to integer 1 implicitly), in other cases an explicit conversion is needed. For example, in the following expression `{ $x > $y }` if `x='9'` and `y='11'`, then the result of the expression is true due to alphabetic order. In order to compare the values as numbers an explicit conversion of at least one argument is required: `{ number($x) > $y }`.

XPath Context in config-templates

When the evaluation of a template starts the XPath context node is set to the root of the service instance data that was created as described above.



Note

When a template is applied from an API, the context node can be specified. See the documentations for the API of your choice.

The root context node can also be changed from within template with help of `set-root-node` processing instruction. It takes an XPath expression as a parameter, and this expression is evaluated in a special context, where the root node is the root of the datastore. This makes it possible to change to a node outside the current evaluation context. For example: `<?set-root-node { / } ?>` changes the accessible tree to the whole datastore. As with other processing instruction, the effect of `set-root-node` only applies until the closing parent tag.

Under certain criteria the evaluation context node is changed to make it easier to work with lists in the templates. To understand the evaluation of templates it is important to understand how and when the context node is changed.

The evaluation context will change if the value being set in the template is the key of a list, and the XPath expression evaluates to a node set. However if the expression evaluates to a value, the context will not change. To explain the algorithm above, let's look at the following example.

The device YANG model has the following snippet.

```
container vrf {
    list definition {
        key "name";
        leaf name {
            type string;
        }
    }
}
```

If the template looks like this:

```
<vrf xmlns="urn:ios" tags="merge">
    <definition>
        <name>{/vpn-service[name='volvo']/leg[node='branch']/name}</name>
        <rd>{as-number}:1</rd>
    </definition>
</vrf>
```

Since `/vpn-service[name='volvo']/leg[node='branch']/name` evaluates to a node, the evaluation context node will be changed to the parent of that node. I.e. `/vpn-`

service[name='volvo']/leg[node='branch']. This means that the second template parameter {as-number} will be relative to the context change, i.e. /vpn-service[name='volvo']/leg[node='branch']/as-number

In some cases it is not desirable to change the context. This can be achieved by forcing the XPath expression to be evaluated to a value rather than a node. If the template looks like this:

```
<vrf xmlns="urn:ios" tags="merge">
<definition>
    <name>{string(/vpn-service[name='volvo']/leg[node='branch']/name)}</name>
    <rd>{as-number}:1</rd>
</definition>
</vrf>
```

The XPath function `string()` is used within the first expression. This evaluates to a value which means the context will remain unchanged. This also means that the second template parameter {as-number} will still be relative to what the context was before, i.e. it remains unchanged.

Another way to write the latter example would be:

```
<vrf xmlns="urn:ios" tags="merge">
<?set-root-node {.}?>
<definition>
    <name>{/vpn-service[name='volvo']/leg[node='branch']/name}</name>
    <rd>{/as-number}:1</rd>
</definition>
</vrf>
```

This example temporarily changes the accessible tree to the subtree under the current context node which makes it easier to write paths under the current node. The accessible tree is restored after the closing `</vrf>` tag.

It is also possible to change the current context node using the processing instruction `context`. For example: `<?set-context-node { . . }?>` will change the context node to the parent of the current context node.

Conditional Statements

Sometimes it is necessary to control which parts of a template that should be evaluated. The `if` and `elif` processing instructions make it possible to set a conditional statement that controls if the sub-tree should be evaluated or not.

```
<policy-map xmlns="urn:ios" tags="merge">
<name>{$POLICY_NAME}</name>
<class>
    <name>{$CLASS_NAME}</name>
    <?if {qos-class/priority = 'realtime'}?>
        <priorty-realtime>
            <percent>{$CLASS_BW}</percent>
        </priority-realtime>
    <?elif {qos-class/priority = 'critical'}?>
        <priorty-critical>
            <percent>{$CLASS_BW}</percent>
        </priority-critical>
    <?else?>
        <bandwidth>
            <percent>{$CLASS_BW}</percent>
        </bandwidth>
    <?end?>
    <set>
        <ip>
```

```

<dscp>{$CLASS_DSCP}</dscp>
</ip>
</set>
</class>
</policy-map>

```

The template example above shows the use of the `if`, `elif` and `else` processing instructions to select between tags `priority-realtime`, `priority-critical` and `bandwidth`. The sub-tree containing the `priority-realtime` tag will only be evaluated if the XPath statement, `qos-class/priority`, in the `if` processing instruction evaluates to the string 'realtime'. The subtree under the `elif` processing instruction will be executed if the preceding `if` expression evaluated to `false`, i.e. `qos-class/priority` is not equal to the string 'realtime', and the XPath statement inside of the `elif` processing instruction evaluates to the string 'critical'. The subtree under the `else` processing instruction will be executed when both the preceding `if` and `elif` expressions evaluated to `false`, i.e. `qos-class/priority` is neither equal to the string 'realtime' nor 'critical'.

The evaluation of the XPath statements used in the `if` and `elif` processing instructions follow the XPath standard for computing boolean values. In summary the conditional expression will evaluate to `false` when:

- The argument evaluates to an empty node set. As in the example above.
- The value of the argument is either an empty string or numerically zero.
- The argument evaluates to a boolean false, using an XPath function like `not()`.

Loop Statements

Sometimes statements in a sub-tree needs to be applied several times. `foreach` and `for` processing instructions can be used to accomplish this iteration.

```

<ip xmlns="urn:ios">
  <route>
    <vrf>
      <name>VPN{/vpn-number}</name>
      <?foreach {tunnel}?>
        <ip-route-forwarding-list>
          <prefix>{network}</prefix>
          <mask>{netmask}</mask>
          <forwarding-address>10.255.254.{(tunnel-number-1)*4+2}</forwarding-address>
        </ip-route-forwarding-list>
      <?end?>
    </vrf>
  </route>
</ip>

```

The template example above shows the use of the `foreach` processing instruction to populate the list `ip-route-forwarding-list`. If the result of the XPath expression `tunnel` is a non-empty node-set then the sub-tree containing `ip-route-forwarding-list` tag will be evaluated for every node in that node-set.

For each iteration the initial context will be set to the node in that iteration. The XPath function `current()` can be used to retrieve that initial context.

There can be only one XPath statement in the `foreach` processing instruction and the result needs to be a node-set, not a simple value. It is however possible to use XPath union operator to join multiple node-sets in a single expression: `{some-list-1 | some-leaf-list-2}`

`for` is a processing instruction that allows to control flow by means of changing a variable rather than changing context node. For example, the following template snippet could be used to disable a range of interfaces:

```

<interface xmlns="urn:ios">
  <?for i=0; {$i < 4}; i={$i + 1}?>
    <FastEthernet>
      <name>0/{$i}</name>
      <shutdown/>
    </FastEthernet>
  <?end?>
</interface>

```

In this example the `for` keyword is followed by three semicolon-separated clauses with the following meaning:

- The first clause is the initial step executed before the loop is entered the first time. The format of the clause is variable name followed by equal sign and an expression that combines literal string and XPath expressions surrounded by `{}`. This expression is evaluated in the same way as the XML tag contents in templates. This clause is optional.
- The second clause is the progress condition. The loop will execute as long as this condition evaluates to true, using the same rules as the `if` processing instruction. The format of this clause is an XPath expression surrounded by `{}`. This clause is mandatory.
- The third clause is executed after each iteration. It has the same format as the first clause (variable assignment) and is optional.

Capabilities and Namespaces

When a device makes itself known to NSO it presents a list of capabilities, see: the section called “Capabilities, Modules and Revision Management” in *NSO 5.7 User Guide*, which includes what YANG modules that particular device support. Since each YANG module define a unique namespace, this information can be used in a Template. Hence, a Template may refer to many diverse devices, where only those pieces of the Template with a matching namespace will be applied for a particular device.

Template tag operations

Templates allow for defining different behaviour when applying the template. This is accomplished by setting tags, as an attribute. Existing tags are: `merge`, `replace`, `delete`, `create` or `nocreate` on relevant nodes in the template.

Tags `mergemerge` and `nocreate` are inherited to their sub-nodes until a new tag is introduced. Tag `delete` applies only to the current node and any children (except keys specifying the list/leaf-list entry to delete) are ignored. Tags `create` and `replace` are not inherited and only apply to the node they are specified on. Children of the nodes with `create` or `replace` tags have `merge` behaviour.

- *merge*: Merge with a node if it exists, otherwise create the node. This is the default operation if no operation is explicitly set.

```

...
<config tags="merge">
  <interface xmlns="urn:ios">
...

```

- *replace*: Replace a node if it exists, otherwise create the node.

```

...
<GigabitEthernet tags="replace">
  <name>{link/interface-number}</name>
  <description tags="merge">Link to PE</description>
...

```

- *create*: Creates a node. The node can not already exist. An error is raised if the node exists.

Operations on ordered-by user lists and leaf-lists

```

...
<GigabitEthernet tags="create">
  <name>{link/interface-number}</name>
  <description tags="merge">Link to PE</description>
...
• nocreate: Merge with a node if it exists. If it does not exist, it will not be created.

...
<GigabitEthernet tags="nocreate">
  <name>{link/interface-number}</name>
  <description tags="merge">Link to PE</description>
...
• delete: Delete the node.

...
<GigabitEthernet tags="delete">
  <name>{link/interface-number}</name>
  <description tags="merge">Link to PE</description>
...

```

Operations on ordered-by user lists and leaf-lists

For ordered-by user lists and leaf-lists, the *insert* attribute can be used to specify where in the list, or leaf-list, the node should be inserted. You need to specify if the node should be inserted first or last in the node-set, or before or after a specific instance. For example, if you have a list of rules where order is important>

```

...
<rule insert="first">
  <name>{$FIRSTRULE}</name>
  ...
</rule>
...
<rule insert="last">
  <name>{$LASTRULE}</name>
  ...
</rule>
...
<rule insert="after" value={$FIRSTRULE}>
  <name>{$SECONDRULE}</name>
  ...
</rule>
...
<rule insert="before" value={$LASTRULE}>
  <name>{$SECONDTOLASTRULE}</name>
  ...
</rule>
...

```

It is not uncommon that there are multiple services managing the same ordered-by user list or leaf-list. The relative order of elements inserted by these services might not matter, but there are constraints on element positions that need to be fulfilled. Following the list of rules example, suppose that initially the list contains only the "deny-all" rule:

```

<rule>
  <ip>0.0.0.0</ip>
  <mask>0.0.0.0</mask>
  <action>deny</action>
</rule>

```

There are services that prepend permit rules to the beginning of the list using insert="first" operation. If there are two services creating one entry each, 10.0.0.0/8 and 192.168.0.0/24 respectively, then the resulting configuration looks like:

```
<rule>
  <ip>192.168.0.0</ip>
  <mask>255.255.255.0</mask>
  <action>permit</action>
</rule>
<rule>
  <ip>10.0.0.0</ip>
  <mask>255.0.0.0</mask>
  <action>permit</action>
</rule>
<rule>
  <ip>0.0.0.0</ip>
  <mask>0.0.0.0</mask>
  <action>deny</action>
</rule>
```

If we now try to check-sync the first service (10.0.0.0/8), then it will report out-of-sync, and re-deploying it would move the 10.0.0.0/8 rule first. But all we want is for the deny-all rule to be last. This is when *guard* attribute comes in handy. If both *insert* and *guard* attributes are specified on a list entry in template, then the template engine will first check whether the list entry already exists in the resulting configuration between the target position (as indicated by the *insert* attribute) and the position of an element indicated by the *guard*.

- If the element exists and fulfills this constraint, then its position will be preserved. If a template list entry results in multiple configuration list entries, then all of them need to exist in the configuration in the same order as calculated by template, and all of them need to fulfill the guard constraint in order for their position to be preserved.
- If the list entry/entries do not exist, are not in the same order or do not fulfill the constraint, then the list is reordered as instructed by the insert statement.

So, assuming the following data model in the example above

```
...
list rule {
  key "ip mask";
  ordered-by user;
  leaf ip {
    type inet:ipv4-address;
  }
  leaf mask {
    type inet:ipv4-address;
  }
  leaf action {
    type enumeration {
      enum permit;
      enum deny;
    }
    mandatory true;
  }
...
}
```

The template could look as following:

```
...
```

```
<rule insert="first" guard="0.0.0.0 0.0.0.0">
  <ip>{$IP}</ip>
  <mask>{$MASK}</mask>
  <action>permit</action>
</rule>
...
```

A *guard* can be specified literally (e.g. `guard="0.0.0.0 0.0.0.0"` if "ip mask" is the key of the list) or using an XPath expression (e.g. `guard="{$GUARDIP} {$GUARDMASK}"`). If the *guard* evaluates to a node-set consisting of multiple elements, then only the first element in this node-set is considered as the *guard*. The constraint defined by the *guard* is evaluated as follows:

- If `insert="first"`, then the constraint is fulfilled if the element exists in the configuration *before* the element indicated by the *guard*.
- If `insert="last"`, then the constraint is fulfilled if the element exists in the configuration *after* the element indicated by the *guard*.
- If `insert="after"`, then the constraint is fulfilled if the element exists in the configuration *before* the element indicated by the *guard*, but *after* the element indicated by the *key* or *value* statement.
- If `insert="before"`, then the constraint is fulfilled if the element exists in the configuration *after* the element indicated by the *guard*, but *before* the element indicated by the *key* or *value* statement.
- If the *guard* evaluates to an empty node-set (i.e. the node indicated by the *guard* does not exist in the target configuration), then the constraint is not fulfilled.

Debugging templates

Sometimes there is a need of some extra information when applying templates in order to understand what is going on. When applying or committing a template there is a cli pipe command `debug` that enables debug information:

```
admin@ncs(config)# commit dry-run | debug template
admin@ncs(config)# commit dry-run | debug xpath
```

`debug template` will output XPath expression results from the template, under which context it is evaluated, what operation is used and how it effects the configuration, for all templates invoked. It can be narrowed down to only show debugging information for a specific template:

```
admin@ncs(config)# commit dry-run | debug template l3vpn
```

`debug xpath` will output *all* XPath evaluations for the transaction, and is not limited to the XPath expressions inside templates.

Template and `xpath` debugging can be combined:

```
admin@ncs(config)# commit dry-run | debug template | debug xpath
```

debug template output examples



Note

The example-snippets in this section are based on the service template *l3vpn* which is described in detail in the section called “Service Templates”. The template itself can be found under `$NCS_DIR/examples.ncs/service-provider/simple-mpls-vpn`. Create the service as described in the README in the example directory, to try it out yourself.

Using the cli pipe flag `debug template` when committing a service which uses template(s) will give you detailed information on what the template will do. Below are selected snippets, with explanations, on

output when committing the *l3vpn* service template with the debug flag. Some lines has been shortened to fit the page.

```
admin@ncs(config)# commit dry-run | debug template
Evaluating "/endpoint/ce/device" (from file "l3vpn.xml", line 5)
Context node: /vpn/l3vpn[name='volvo']
Result:
For /vpn/l3vpn[name='volvo']/endpoint[id='c1']/ce, it evaluates to "ce0"
For /vpn/l3vpn[name='volvo']/endpoint[id='c2']/ce, it evaluates to "ce2"
Operation 'nocreate' on node /devices/device[name='ce0'] (from file "l3vpn.xml", line 5)
Node exists, continuing...
...
```

The output shows that a config-template is in play (as the origin of the template is an xml-file *l3vpn.xml*). It shows that */endpoint/ce/device* from line 5 is evaluated under the context */vpn/l3vpn[name='volvo']*, and that it evaluates to 2 values, *ce0* and *ce2*, as */endpoint/ce/device* is a node set of 2 instances, *c1* and *c2*. Next, it shows that it will start with the first node in the set, and perform a 'nocreate' operation, and that the node exists. Line 4-6 in the *l3vpn.xml*:

```
4 <device tags="nocreate" >
5 <name>{/endpoint/ce/device}</name>
6 <config tags="merge">
```

Continuing with the debug output:

```
Operation 'merge' on non-existing node: /devices/device[name='ce0']/.../description
(from file "l3vpn.xml", line 11)
Fetching literal "Link to PE" (from file "l3vpn.xml", line 11)
Setting /devices/device[name='ce0']/.../description to "Link to PE"
```

Above it shows that the node *description*, which did not exist beforehand, will be set to the value *Link to PE*. Line 11 in the *l3vpn.xml*:

```
11 <description tags="merge">Link to PE</description>
```

Continuing with the debug output:

```
Operation 'nocreate' on node /devices/device[name='pe2']
(from file "l3vpn.xml", line 52)
Node exists, continuing...
The device /devices/device[name='pe2'] does not support
namespace 'urn:ios' for node "interface" (from file "l3vpn.xml", line 69)
```

Further down in the output it is shown that the device *pe2* does not support configuration with namespace *urn:ios* and therefore that part of the template, and its siblings, will be skipped. Line 69 in the *l3vpn.xml*:

```
69 <interface xmlns="urn:ios" tags="nocreate">
```

This was just selected extracts of the output; the real output shows each and every effect the template will have. As seen the debug command is very useful for the understanding on how a template is interpret by the system.

XPath help

In addition to the debug command there are some other options available which can be used in other scenarios. To get the XPath selections right use the NSO CLI show command with the *xpath display* flag to find out the correct path to an instance node. This shows the name of the key elements and also the name space changes.

```
% show full-configuration devices device c0 config ios:interface | display xpath
/devices/device[name='c0']/config/ios:interface/FastEthernet[name='1/0']
/devices/device[name='c0']/config/ios:interface/FastEthernet[name='1/1']
/devices/device[name='c0']/config/ios:interface/FastEthernet[name='1/2']
/devices/device[name='c0']/config/ios:interface/FastEthernet[name='2/1']
/devices/device[name='c0']/config/ios:interface/FastEthernet[name='2/2']
```

When using more complex expressions the ncs_cmd utility can be used to experiment and to debug expressions. ncs_cmd is used in a command shell. The command does not print the result as XPath selections but is still of great use when debugging XPath expressions. The below example selects FastEthernet interface names on device c0:

```
$ ncs_cmd -c \
  "x /devices/device[name='c0']/config/ios:interface/FastEthernet/name"
/devices/device{c0}/config/interface/FastEthernet{1/0}/name [1/0]
/devices/device{c0}/config/interface/FastEthernet{1/1}/name [1/1]
/devices/device{c0}/config/interface/FastEthernet{1/2}/name [1/2]
/devices/device{c0}/config/interface/FastEthernet{2/1}/name [2/1]
/devices/device{c0}/config/interface/FastEthernet{2/2}/name [2/2]
```

XPath trace log

To understand the XPath expressions when the template is applied the xpath trace is a valuable tool. XPath trace can be enabled in the NSO configuration file ncs.conf. In the examples it is enabled by default.

In a terminal window one can follow the evaluation of XPath expressions in the log file:

```
$ tail -f logs/xpath.trace
```

Service Templates

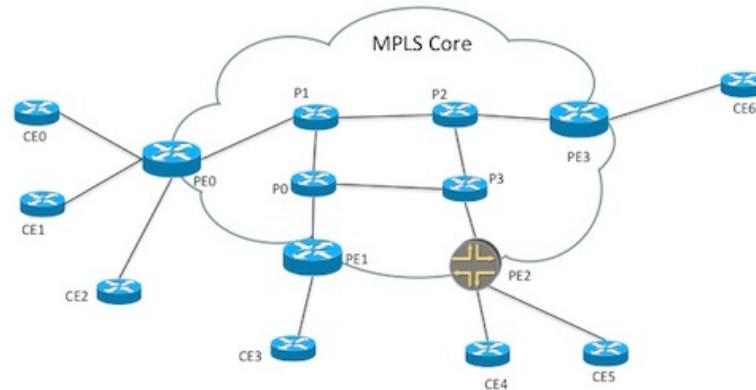
The core function of NSO is the capability to map service models to device models. When the mapping is a pure function from service parameters to device parameters the template mechanism can be used to define this mapping in a declarative way. When the mapping includes calls to external systems, complex algorithms etc this can be expressed in programmatic mapping logic instead. NSO supports Java for defining Mapping Logic.

Templates can express the mapping in many cases and have the benefit of being expressed in a way network engineers think. Based on the concepts presented above you can look at using templates to specify service models and how the service models can be transformed to device configurations. This would be of limited use without FASTMAP. The template mechanism declaratively maps the service configuration to device configurations. But the NSO FASTMAP algorithm will enable NSO users to modify service instances and deleting service instances and FASTMAP in combination with the template definition will calculate and apply the minimum diff to apply the changes to the network.

The example will illustrate this by setting up Layer3 VPNs in a service provider MPLS network. The example consists of Cisco ASR 9k core routers (P and PE) and Cisco IOS based CE routers, as shown in Figure 174, “The example network” All the code for the example can be found as a running example in:

```
$NCS_DIR/examples.ncs/service-provider/simple-mpls-vpn
```

Figure 174. The example network



Imagine the following service model:

Example 175. Service Model using Templates

```

list l3vpn {
    description "Layer3 VPN";

    key name;
    leaf name {
        tailf:info "Unique service id";
        tailf:cli-allow-range;
        type string;
    }

    uses ncs:service-data;
    ncs:servicepoint "l3vpn-template";

    list endpoint {
        key "id";
        leaf id{
            tailf:info "Endpoint identifier";
            type string;
        }
        leaf as-number {
            description "AS used within all VRF of the VPN";
            tailf:info "MPLS VPN AS number.";
            mandatory true;
            type uint32;
        }
        container ce {
            leaf device {
                mandatory true;
                type leafref {
                    path "/ncs:devices/ncs:device/ncs:name";
                }
            }
        container local {
            uses endpoint-grouping;
        }
        container link {
            uses endpoint-grouping;
        }
    }
}

```

```

container pe {
    leaf device {
        mandatory true;
        type leafref {
            path "/ncs:devices/ncs:device/ncs:name";
        }
    }
    container link {
        uses endpoint-grouping;
    }
}

grouping endpoint-grouping {
    leaf interface-name {
        tailf:info "Interface name. For example FastEthernet.";
        type string;
    }
    leaf interface-number {
        tailf:info "Interface number. For example 0 or 1/0";
        type string;
    }
    leaf ip-address {
        tailf:info "Local interface address.";
        type inet:ipv4-address;
    }
}

```

Note the line containing the statement: `ncs:servicepoint "l3vpn-template"`. A reference to this servicepoint can be found in the corresponding template that implements this service. As soon as this service data is manipulated, the Template Engine will be invoked via this servicepoint in order to apply the corresponding template.

The corresponding template that maps the service data to the device data is shown in [Example 176, “Service to Device Model mapping using Templates”](#)



Note

In order to save space, only some parts of the template will be shown here. Refer to the full example for the complete template.

In several places the template uses data from the service data-model, note the expressions enclosed in curly brackets in the listing below

Example 176. Service to Device Model mapping using Templates

The templates gets loaded from the `templates` sub directory of packages.

```

T 1:<config-template xmlns="http://tail-f.com/ns/config/1.0"
T 2:  servicepoint="l3vpn-template">
T 3:  <devices xmlns="http://tail-f.com/ns/ncs">
T 4:    <?foreach {endpoint/ce}?>
T 5:      <device tags="nocreate">
T 6:        <name>{device}</name>
T 7:        <config tags="merge">
T 8:          <!-- CE template for Cisco IOS routers -->
T 9:          <interface xmlns="urn:ios">
T 10:            <?foreach {link}?>
T 11:              <GigabitEthernet tags="nocreate">

```

```

T 12:          <name>{interface-number}</name>
T 13:          <description tags="merge">Link to PE</description>
T 14:          <ip tags="merge">
T 15:              <address>
T 16:                  <primary>
T 17:                      <address>{ip-address}</address>
T 18:                      <mask>255.255.255.252</mask>
T 19:                  </primary>
T 20:              </address>
T 21:          </ip>
T 22:      </GigabitEthernet>
T 23:  <?end?>
T 24:  <?foreach {local}?>
T 25:      <GigabitEthernet tags="nocreate">
T 26:          <name>{interface-number}</name>
T 27:          <description tags="merge">Local network</description>
T 28:          <ip tags="merge">
T 29:              <address>
T 30:                  <primary>
T 31:                      <address>{ip-address}</address>
T 32:                      <mask>255.255.255.0</mask>
T 33:                  </primary>
T 34:              </address>
T 35:          </ip>
T 36:      </GigabitEthernet>
T 37:  <?end?>
T 38:  </interface>
T 39:  <router xmlns="urn:ios">
T 40:      <bgp>
T 41:          <as-no>{.../as-number}</as-no>
T 42:          <neighbor>
T 43:              <id>{pe/link/ip-address}</id>
T 44:              <remote-as>100</remote-as>
T 45:              <activate/>
T 46:          </neighbor>
T 47:          <redistribute>
T 48:              <connected>
T 49:              </connected>
T 50:          </redistribute>
T 51:      </bgp>
T 52:  </router>
T 53:  </config>
T 54:  </device>
T 55: <?end?>

    ...

T 181:  </devices>
T 182:</config-template>}

```

The template will have the same name as the file it was loaded from except the .xml suffix. Note the attribute at *Line 'T 2'*: this template will be applied when changes are made at servicepoint: 13vpn-template. Or paraphrased: this template implements the service at servicepoint 13vpn-template.

The service data used in this example is:

Example 177. Service instance data

```

S 1:vpn 13vpn volvo
S 2:    endpoint c1
S 3:    as-number 65001
S 4:    ce device ce0

```

```

S 5:      ce local interface-name GigabitEthernet
S 6:      ce local interface-number 0/9
S 7:      ce local ip-address 192.168.0.1
S 8:      ce link interface-name GigabitEthernet
S 9:      ce link interface-number 0/2
S 10:     ce link ip-address 10.1.1.1
S 11:     pe device pe2
S 12:     pe link interface-name GigabitEthernet
S 13:     pe link interface-number 0/0/0/1
S 14:     pe link ip-address 10.1.1.2
S 15:   !
S 16:   endpoint c2
S 17:   as-number 65001
S 18:   ce device ce2
S 19:   ce local interface-name GigabitEthernet
S 20:   ce local interface-number 0/3
S 21:   ce local ip-address 192.168.1.1
S 22:   ce link interface-name GigabitEthernet
S 23:   ce link interface-number 0/1
S 24:   ce link ip-address 10.2.1.1
S 25:   pe device pe2
S 26:   pe link interface-name GigabitEthernet
S 27:   pe link interface-number 0/0/0/2
S 28:   pe link ip-address 10.2.1.2
S 28:}

```

The different lines in the template will be explained below.

- *Line 'T 4'*: This annotates the node with the tag `nocreate`. No device nodes will be created, this tag is in effect until a new tag is introduced. No sub-nodes to the device node will be created either. Changes introduced by the template will only be applied to existing nodes.

Further The XPath expression within curly brackets: `/endpoint/ce` is an absolute path. The root of all paths is the service data at 'S 1': `vpn 13vpn volvo`. The XPath evaluation context is set to this root. Hence, 'T 4' will result in a node set consisting of the nodes at: `vpn 13vpn volvo endpoint ce`.

The processing instruction `foreach` will make the template to be applied once for every node in the resulting node-set of the XPath expression `endpoint/ce`. For every iteration the initial context will be set to each node in turn.

- *Line 'T 5'*: The XPath expression within curly brackets: `device` is relative path. This path is relative to the initial context.

In this example, this node set will contain `ce0` (line: 'S 4') and `ce2` (line: 'S 18'), which means that the lines of the template, 'T 4' to 'T 49', will be applied to first the device `ce0` and secondly to the device `ce2`.

As a side effect of the XPath expression at 'T 5', the evaluation context will be changed to its parent node (`ce`). This will make it more convenient to use relative XPath expressions in the succeeding template. Read more about the XPath evaluation context in [the section called “XPath Context in config-templates”](#).

- *Line 'T 6'*: Change the way the template is applied by annotating the node `config` with the tag `merge`. Existing nodes will be changed and if a node does not exist it will be created.
- *Line 'T 8'*: Note then namespace attribute value "urn:ios" of the node `interface`. 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.
- *Line 'T 9'*: Iterate over the nodes in the node-set resulting from the relative XPath expression `link`.

Note how you can use a relative XPath expression here since the evaluation context was changed at line 'T 5'.

- **Line 'T 10':** Select the data at the relative XPath `interface-number`. This selects the value `0 / 2` at '`S 9`' for the `ce0` node, and the value `0 / 1` at '`S 23`' for the `ce2` node.

Note how you can use a relative XPath expression here since the evaluation context was changed at line 'T 5'.

When you have entered the Service configuration, you can check what will be committed by doing a `commit dry-run`. Only some pieces of the output is shown below:

Example 178. Result from the Template Service

```
admin@ncs(config)# commit dry-run outformat xml
result-xml <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
        <name>ce0</name>
        <config>
            <router xmlns="urn:ios">
                <bgp>
                    <as-no>65001</as-no>
                    <neighbor>
                        <id>10.1.1.2</id>
                        <remote-as>100</remote-as>
                        <activate/>
                    ...
                <devices xmlns="http://tail-f.com/ns/ncs">
                    <device>
                        <name>ce2</name>
                        <config>
                            <router xmlns="urn:ios">
                                <bgp>
                                    <as-no>65001</as-no>
                                    <neighbor>
                                        <id>10.2.1.2</id>
                                        <remote-as>100</remote-as>
                                    ...
                            </config>
                        </device>
                    </devices>
                </router>
            </config>
        </device>
    </devices>
</result-xml>
```

Macros in Service Templates

Service templates can define macros - named XML snippets that can be expanded in the template. The example template shown in [Example 176, “Service to Device Model mapping using Templates”](#) can also be written to define and use macros, as shown in [Example 179, “Service Template using Macros”](#)

Example 179. Service Template using Macros

```
M 1:<config-template xmlns="http://tail-f.com/ns/config/1.0"
M 2:  servicepoint="l3vpn-template">
M 3:      <?macro GbEth name='{interface-number}' mask='255.255.255.0' desc?>
M 4:      <GigabitEthernet tags="nocreate">
M 5:          <name>$name</name>
M 6:          <description tags="merge">$desc</description>
M 7:          <ip tags="merge">
M 8:              <address>
M 9:                  <primary>
M 10:                     <address>{ip-address}</address>
M 11:                     <mask>$mask</mask>
M 12:                 </primary>
M 13:             </address>
M 14:         </ip>
```

Macros in Service Templates

```

M 15:      </GigabitEthernet>
M 16:      <?endmacro?>
M 17:
M 18:      <?macro GbEthWithVrf name='{interface-number}' 
M 19:          mask='255.255.255.252' desc forwarding?>
M 20:      <?expand GbEth name=$name desc='$desc' mask=$mask?>
M 21:      <GigabitEthernet tags="nocreate">
M 22:          <name>$name</name>
M 23:          <vrf tags="merge">
M 24:              <forwarding>$forwarding</forwarding>
M 25:          </vrf>
M 26:      </GigabitEthernet>
M 27:      <?endmacro?>
M 28:
M 29:      <devices xmlns="http://tail-f.com/ns/ncs">
M 30:          <?foreach {endpoint/ce}?>
M 31:              <device tags="nocreate">
M 32:                  <name>{device}</name>
M 33:                  <config tags="merge">
M 34:                      <!-- CE template for Cisco IOS routers -->
M 35:                      <interface xmlns="urn:ios">
M 36:                          <?foreach {link}?>
M 37:                              <?expand GbEth descr='Link to PE' mask='255.255.255.252'?>
M 38:                          <?end?>
M 39:                          <?foreach {local}?>
M 40:                              <?expand GbEth descr='Local network'?>
M 41:                          <?end?>
M 42:                      </interface>

    ...

M 57:      </config>
M 58:      </device>
M 59:      <?end?>

    ...

M 185:  </devices>
M 186:</config-template>}
```

Macro definitions and usage should adhere to the following rules.

- A macro should be a valid chunk of XML (or it could be a string without any XML markup). So, a macro cannot contain only start-tags or only end-tags, for example.
- Each macro is defined between the `<?macro?>` and `<?endmacro?>` processing-instructions, immediately following the `<config-template>` tag in the template.
- A macro definition takes a name and an optional list of parameters. The parameters can optionally have a default value.

In the example [Example 179, “Service Template using Macros”](#) a macro is defined at *Line 'M 3'*:

```
<?macro GbEth name='{interface-number}' mask='255.255.255.0' desc?>
```

Here, `GbEth` is the name of the macro. And this macro takes three parameters, `name`, `mask` and `desc`. The parameters `name` and `mask` have default values, and `desc` does not.

- A macro can be expanded in another location in the template using the `<?expand?>` processing-instruction. As shown in *Line 'M 37'* in the example, the `<?expand?>` instruction takes the name of the macro to expand, and an optional list of parameters and their values.

The parameters in the macro definition are replaced with the values given during expansion. If a parameter is not given any value during expansion, the default value is used. If there is no default

value in the definition, a value should be supplied for the parameter during expansion, and it is an error otherwise.

- Macro definitions cannot be nested - that is, a macro definition cannot contain another macro definition. But a macro definition can have <?expand?> instructions to expand another macro within this macro.

The macro expansion and the parameter replacement work on just strings - there is no schema validation or xpath evaluation at this stage. A macro expansion just inserts the macro definition at the expansion site.

- Macros can be defined in multiple files, and macros defined in the same package will be visible to all templates in that package. This means that a template file could have just the definitions of macros, and another file in the same package could use expand those macros.

When reporting errors, if the template uses macros, the line numbers for the macro invocations are also included, so that the actual location of the error can be traced. For example, an error message could be something like `service.xml:19:8 Invalid parameters for processing instruction set.` - this means that there was a macro expansion on line 19 in `service.xml` and an error occurred at line 8 in the file defining that macro.

Pre- and Post-Modification Templates

Pre- and post-modification templates are applied before and after, respectively, applying the service template. The template displayed in [Example 180, “Post Modification Template”](#), demonstrates how to define a post-modification callback using a template for the service template defined in [Example 176, “Service to Device Model mapping using Templates”](#). Pre-modification templates are defined similarly but with the `cbtype` attribute set to `pre-modification` instead of `post-modification`. The `$OPERATION` variable is set internally by NSO in pre- and post-modification templates and contains the service operation, i.e., create, update or delete, that triggered the callback. The `$OPERATION` variable can be used together with template conditional statements (see [the section called “Conditional Statements”](#)) to apply different parts of the template depending on the triggering operation.

For more information on service pre- and post-modification callbacks, refer to [the section called “ Pre and post hooks ”](#).

Example 180. Post Modification Template

```
1: <config-template xmlns="http://tail-f.com/ns/config/1.0"
2:   servicepoint="l3vpn-template"
3:   cbtype="post-modification">
4:   <if {$OPERATION = 'create'}?>
5:     <devices xmlns="http://tail-f.com/ns/ncs">
6:       <device>
7:         <name>{/device}</name>
8:         <config>
9:
10:        ...
11:
12:        </config>
13:      </device>
14:    </devices>
15:    <else?>
16:      <if {$OPERATION = 'update'}?>
17:
18:        ...
19:
20:        <else?>
21:          <!-- $OPERATION = 'delete' -->
```

```

    ...
30:      <?end?>
31:      <?end?>
32:  </config-template>
```



Note The service's data is not available in the pre-or post-modification callbacks when $\$OPERATION = 'delete'$ since the service has been deleted already in the transaction context where the template is applied.



Note Pre- and post-modification templates can only be used in normal services when the create callback is also implemented as a template, i.e. they cannot be used together with create callbacks implemented in Java or Python. Note that this limitation does not apply to nano services.

Templates applied from an API

When a pure mapping of service instance data to device data is not enough and logic and algorithms are needed, a template based service is not sufficient to implement a service. The logic can be placed in Java code and the template can be applied from the Java code.

Just implement the Java code as any other Java service and apply the template. The Template class for Java is defined in: `com.tailf.ncs.template.Template`

Feature Template

A feature template is a config-template used to configure a specific feature. By using feature templates, the service code can be completely device-vendor agnostic. Feature templates is a nice way to decompose a problem into smaller, reusable, chunks.

In the example `$NCS_DIR/examples.ncs/service-provider/mpls-vpn` a l3vpn-acl template is used to create ACL entries for all service specific match rules, and is applied from the Java service code. Below you will find the `l3vpn-acl.xml` as well as parts of the Java logic which shows how the template variables are set and then how the template is applied.

The l3vpn-acl template:

```

<config-template xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device tags="nocreate">
      <name>{$CE}</name>
      <config>
        <ip xmlns="urn:ios" tags="merge">
          <access-list>
            <extended>
              <ext-named-acl>
                <name>{$ACL_NAME}</name>
                <ext-access-nlist-rule>
                  <rule>permit {${$PROTOCOL}} {${$SOURCE_IP_ADDR}}
                    {${$SOURCE_WMASK}} {${$DEST_IP_ADDR}} {${$DEST_WMASK}}
                    range {${$PORT_START}} {${$PORT_END}}</rule>
                </ext-access-nlist-rule>
              </ext-named-acl>
            </extended>
          </access-list>
        </ip>
      </config>
    </device>
  </devices>
</config-template>
```

```

        </ip>
    </config>
</device>
</devices>
</config-template>

```

The setting of the variables and applying of the template from Java code:

```

private TemplateVariables setAclVars(GeneratedValue match,
                                     String namePrefix)
    throws NavuException, UnknownHostException {
    TemplateVariables aclVar = new TemplateVariables();

    aclVar.putQuoted("ACL_NAME", namePrefix + "-" +
                      match.leaf("name") .
                      valueAsString());
    aclVar.putQuoted("PROTOCOL", match.leaf("protocol") .
                      valueAsString());
    aclVar.putQuoted("SOURCE_IP", match.leaf("source-ip") .
                      valueAsString());
    if ("any".equals(match.leaf("source-ip").valueAsString())) {
        aclVar.putQuoted("SOURCE_IP_ADDR", "any");
        aclVar.putQuoted("SOURCE_WMASK", " ");
    }
    else {
        aclVar.putQuoted("SOURCE_IP_ADDR",
                         getIPAddress(match.leaf("source-ip") .
                           valueAsString()));
        aclVar.putQuoted("SOURCE_WMASK",
                         prefixToWildcardMask(getIPPPrefix(
                           match.leaf("source-ip") .
                           valueAsString())));
    }
    if ("any".equals(match.leaf("destination-ip") .
                     valueAsString())) {
        aclVar.putQuoted("DEST_IP_ADDR", "any");
        aclVar.putQuoted("DEST_WMASK", " ");
    }
    else {
        aclVar.putQuoted("DEST_IP_ADDR",
                         getIPAddress(match.leaf("destination-ip") .
                           valueAsString()));
        aclVar.putQuoted("DEST_WMASK",
                         prefixToWildcardMask(getIPPPrefix(
                           match.leaf("destination-ip") .
                           valueAsString())));
    }
    aclVar.putQuoted("PORT_START", match.leaf("port-start") .
                      valueAsString());
    aclVar.putQuoted("PORT_END", match.leaf("port-end") .
                      valueAsString());
    return aclVar;
}

...
TemplateVariables aclVar = setAclVars(match, namePrefix);
aclTemplate.apply(service, aclVar);
...

```

Passing deep structures from an API

There is a well-known mechanism to pass parameters to a template via API which is template variables. One limitation of this mechanism is that a variable can only hold one string value. However sometimes

there is a need to pass not just one value, but a list, map or even more complex data structure from API to be accessible in the template.

One way to handle this is to use smaller templates with variables and invoke them repeatedly passing the parameters from the list one by one, or pair by pair in case of a map. However there are certain disadvantages with this approach. One of them is the performance: every invocation of the template from the API requires a context switch between the user application process and NSO core process, which can be costly. Another disadvantage is that the logic of the service is split between Java/Python code and the template, which makes it harder to write and understand such services.

An approach suggested in this section involves modelling the auxiliary data used by the service as operational data and populating it by the service code in Java or Python. After that the service callback passes control to the template that handles main service logic. Such auxiliary data would then be available to select by means of XPath, just like any other service input data.

There could be different approaches to modelling the auxiliary data. It can reside in the service tree as it is private to the service instance: either integrated in the existing data tree, or as a separate subtree under the service instance. It can also be located outside of the service instance, however it is important to keep in mind that operational data cannot be shared by multiple services because there is no refcounters or backpointers stored on operational data.

After the service is deployed, the auxiliary leafs remain in the database which facilitates debugging because they can be seen via all northbound interfaces. If this is not the intention, they can be hidden with help of `tailf:hidden` statement. Because operational data is also a part of FASTMAP diff, these values will be deleted when the service is deleted and need to be recomputed when the service is re-deployed. This also means that in most cases there should be no need to write any additional code to clean up this data.

One example of a task that is hard to solve in the template itself using native XPath functions is converting a network prefix into a network mask or vice versa. Below is a snippet of data model that is part of a service input data and contains a list of interfaces along with IP addresses to be configured on those interfaces. If the format of the IP address on the input is an IP address with prefix but the device accepts an IP address with network mask instead, then we can use an auxiliary leaf mask to be able to convert prefix to the suitable format before the service template takes control.

```
list interface {
    key name;
    leaf name {
        type string;
    }
    leaf address {
        type tailf:ipv4-address-and-prefix-length;
        description
            "IP address with prefix in the following format, e.g.: 10.2.3.4/24";
    }
    leaf mask {
        config false;
        type inet:ipv4-address;
        description
            "Auxiliary data populated by service code, represents network mask
            corresponding to the prefix in the address field, e.g.: 255.255.255.0";
    }
}
```

The service code would need to populate the mask. It uses NAVU API in this example to do that, but if the list is known to be large it might be useful to use Maapi.setValues() instead to set the leafs in one go.

```
for (NavuListEntry interface : interfaces) {
```

```

String address = interface.leaf("address").valueAsString();
String prefix = address.split("//")[1];
String mask = prefixToNetMask(prefix);
// sharedSet() does not take effect on operational data
// using set() instead
interface.leaf("mask").set(mask);
}
...
// TemplateVariables don't need to contain mask
// it is passed via database
TemplateVariables tv = new TemplateVariables();
template.apply(service, tv);

```

The template would then do the following:

```

<interface>
  <name>{/interface/name}</name>
  <ip-address>{substring-before(address, '/')}</ip-address>
  <ip-mask>{mask}</ip-mask>
</interface>

```

Service and API Templates in multi-NED environment

One of the advantages of using templates is that they are validated against the currently loaded schema which allows to detect a lot of the errors at load time instead of run time. However, such strict validation may become a problem for developing generic, reusable templates that should run in different environment with different sets of NEDs and NED versions loaded. For example, loading fewer NED versions than the template is designed for may result in some elements not recognized, while loading more NED versions may introduce ambiguities.

In order to allow templates to be reusable while at the same time keep as many errors as possible detectable at load time, NSO supports a concept of *supported-ned-ids*. This is a set of ned-ids the package developer may declare in *package-meta-data.xml* file to indicate that the XML (both service and API) templates contained in this package were designed to support. This gives NSO a hint on how to interpret the template.

Namely, if a package declares a list of supported-ned-ids, then the templates in this package are interpreted as if no other ned-ids are loaded in the system. If such template is attempted to be applied to a device with ned-id outside the supported list, then a run-time error is generated because this ned-id was not considered when the template was loaded. This allows us to ignore ambiguities in the data model introduced by additional NEDs that were not considered during template development.

If a package declares a list of supported-ned-ids and the runtime system does not have one or more declared NEDs loaded, then the template engine uses so-called *relaxed* loading mode, which means it ignores any unknown namespaces and `<?if-ned-id?>` clauses containing exclusively unknown ned-ids assuming that these parts of the template are not applicable in the current running system.

Because relaxed loading mode performs less strict validation and potentially prevents some errors from being detected, the package developer should always make sure to test in the system with all supported ned-ids loaded, i.e. when the loading mode is *strict*. The loading mode can be verified by looking at the value of `template-loading-mode` leaf for the corresponding package under `/packages/package` list.

If the package does not declare any supported-ned-ids, then the templates are loaded strictly in the full set of currently loaded ned-ids. This makes the package less reusable between different systems, but may still be desirable in environments where the package is intended to be used in a single or multiple identical runtime system fully under control of the package developer.



CHAPTER 16

Developing Alarm Applications

- [Introduction, page 375](#)
- [Using the Alarms Sink, page 376](#)
- [Using the Alarms Source, page 378](#)
- [Extending the alarm manager, adding user defined alarm types and fields, page 379](#)
- [Mapping alarms to objects, page 381](#)

Introduction

This chapter focus on how to manipulate the NSO alarm table using the dedicated Alarm APIs. Make sure that the concepts in the section called “Alarm Manager Introduction” in *NSO 5.7 User Guide* are well understood before reading this section.

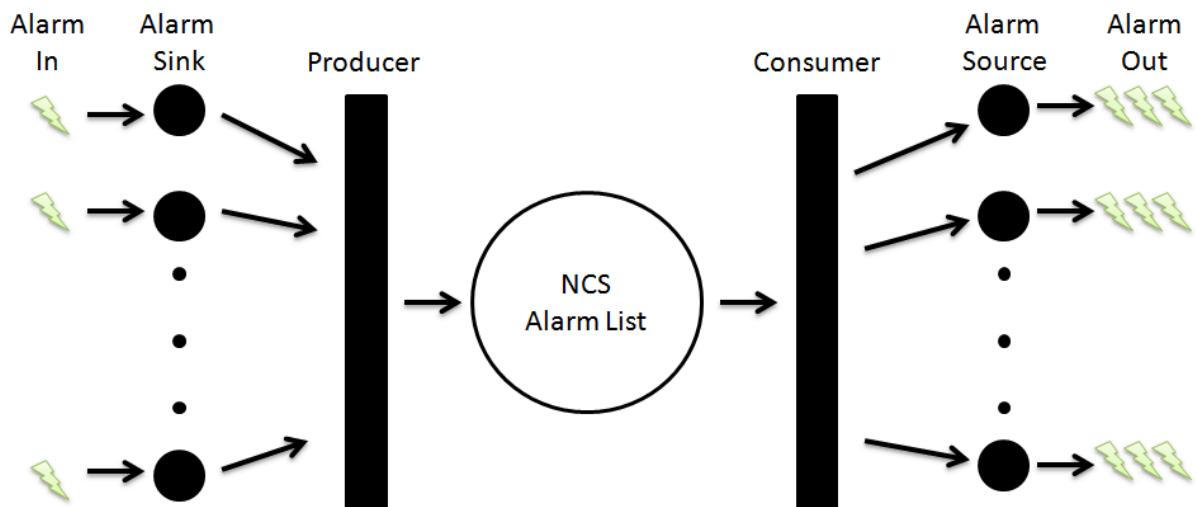
The Alarm API provides a simplified way of managing your alarms for the most common alarm management use cases. The API is divided into a producer and a consumer part.

The producer part provides an alarm sink. Using an alarm sink you can submit your alarms into the system. The alarms are then queued and fed into the NSO alarm list. You can have multiple alarm sinks active at any time.

The consumer part provides an Alarm Source. The alarm source lets you listen to new alarms and alarm changes. As with the producer side you can have multiple alarm sources listening for new and changed alarms in parallel.

The diagram below show a high level view of the flow of alarms in and out of the system. Alarms are received, e.g as SNMP notifications, and fed into the NSO Alarm List. At the other end you subscribe for the alarm changes.

Figure 181. The Alarm Flow



Using the Alarms Sink

The producer part of the Alarm API can be used in the following modes:

- **Centralized Mode:** This is the preferred mode for NSO. In the centralized mode we submit alarms towards a central alarm writer that optimizes the number of sessions towards the CDB. The NSO Java VM will setup the centralized alarm sink at start-up which will be available for all java components run by the NSO Java VM.
 - **Local Mode:** In the local mode we submit alarms directly into the CDB. In this case each Alarm Sink keeps its own CDB session. This mode is the recommended mode for applications run outside of the NSO java VM or java components that have specific need of controlling the CDB session.

The difference between the two modes is manifested by the way you retrieve the AlarmSink instance to use for alarm submission. For submitting an alarm in centralized mode a prerequisite is that a central alarm sink has been set up within your JVM. For components in the NSO java VM this is done for you. For applications outside of the NSO java VM which want to utilize the centralized mode, you need to get a `AlarmSinkCentral` instance. This instance has to be started and the central will then execute in a separate thread. The application needs to maintain this instance and stop it when the application finish.

Example 182. Retrieving and starting an AlarmSinkCentral

```
Socket socket = new Socket("127.0.0.1",Conf.NCS_PORT);
Cdb cdb = new Cdb("MySinkCentral", socket);

AlarmSinkCentral sinkCentral = AlarmSinkCentral.getAlarmSink(1000, cdb);
sinkCentral.start();
```

The centralized alarm sink can then be retrieved using the default constructor in the `AlarmSink` class.

Example 183. Retrieving AlarmSink using centralized mode

```
AlarmSink sink = new AlarmSink();
```

When submitting an alarm using the local mode, you need a CDB socket and a Cdb instance. The local mode alarm sink needs the Cdb instance in order to write alarm info to CDB. The local alarm sink is retrieved using a constructor with a Cdb instance as an argument.

Example 184. Retrieving AlarmSink using local mode

```
Socket socket = new Socket("127.0.0.1",Conf.NCS_PORT);
Cdb cdb = new Cdb(MyLocalModeExample.class.getName(), socket);

AlarmSink sink = AlarmSink(cdb);
```

The `sink.submitAlarm(...)` method provided by the `AlarmSink` instance can be used in both centralized and local mode to submit an alarm.

Example 185. Alarm submit

```
package com.tailf.ncs.alarmman.producer;
...
/**
 * Submits the specified <code>Alarm</code> into the alarm list.
 * If the alarms key
 * "managedDevice, managedObject, alarmType, specificProblem" already
 * exists, the existing alarm will be updated with a
 * new status change entry.
 *
 * Alarm identity:
 *
 * @param managedDevice the managed device which emits the alarm.
 *
 * @param managedObject the managed object emitting the alarm.
 *
 * @param alarmtype the alarm type of the alarm.
 *
 * @param specificProblem is used when the alarmtype cannot uniquely
 * identify the alarm type. Normally, this is not the case,
 * and this leaf is the empty string.
 *
 * Status change within the alarm:
 * @param severity the severity of the alarm.
 * @param alarmText the alarm text
 * @param impactedObjects Objects that might be affected by this alarm
 * @param relatedAlarms Alarms related to this alarm
 * @param rootCauseObjects Objects that are candidates for causing the
 * alarm.
 * @param timeStamp The time the status of the alarm changed,
 * as reported by the device
 * @param customAttributes Custom attributes
 *
 * @return boolean true/false whether the submitting the specified
 * alarm was successful
 *
 * @throws IOException
 * @throws ConfException
 * @throws NavuException
 */
public synchronized boolean
submitAlarm(ManagedObject managedDevice,
           ManagedObject managedObject,
           ConfIdentityRef alarmtype,
           ConfBuf specificProblem,
           PerceivedSeverity severity,
           ConfBuf alarmText,
           List<ManagedObject> impactedObjects,
           List<AlarmId> relatedAlarms,
           List<ManagedObject> rootCauseObjects,
           ConfDatetime timeStamp,
           Attribute ... customAttributes)
```

```

        throws NavuException, ConfException, IOException {
    ...
}

...
}

```

Below follows an example showing how to submit alarms using the centralized mode, which is the normal scenario for components running inside the NSO Java VM. In the example we create an alarm sink and submit an alarm.

Example 186. Submitting an alarm in a centralized environment

```

...
AlarmSink sink = new AlarmSink();
...

// Submit the alarm.

sink.submitAlarm(new ManagedDevice("device0"),
    new ManagedObject("/ncs:devices/device{device0}"),
    new ConfIdentityRef(new MyAlarms().hash(),
        MyAlarms._device_on_fire),
    PerceivedSeverity.INDETERMINATE,
    "Indeterminate Alarm",
    null,
    null,
    null,
    ConfDatetime.getConfDatetime(),
    new AlarmAttribute(new myAlarm(), // A custom alarm attribute
        myAlarm._custom_alarm_attribute_,
        new ConfBuf("this is an alarm attribute")),
    new StatusChangeAttribute(new myAlarm(), // A custom status change attribute
        myAlarm._custom_status_change_attribute_,
        new ConfBuf("this is a status change attribute")));
...

```

Using the Alarms Source

In contrast to the alarm source, the alarm sink only operates in centralized mode. Therefore, before being able to consume alarms using the alarm API you need to set up a central alarm source. If you are executing components in the scope of the NSO Java VM this central alarm source is already set up for you.

You typically set up a central alarm source if you have a stand alone application executing outside the NSO Java VM. Setting up a central alarm source is similar to setting up a central alarm sink. You need to retrieve a `AlarmSourceCentral`. Your application needs to maintain this instance, which implies starting it at initialization and stopping it when the application finishes.

Example 187. Setting up an alarm source central

```

socket = new Socket("127.0.0.1",Conf.NCS_PORT);
cdb = new Cdb("MySourceCentral", socket);

source = AlarmSourceCentral.getAlarmSource(MAX_QUEUE_CAPACITY, cdb);
source.start();

```

The central alarm source subscribes to changes in the alarm list and forwards them to the instantiated alarm sources. The alarms are broadcast to the alarm sources. This means that each alarm source will receive its own copy of the alarm.

The alarm source promotes two ways of receiving alarms:

- Take: Block execution until an alarm is received.
- Poll: Wait for alarm with a timeout. If you do not receive an alarm within the stated time frame the call will return.

Example 188. AlarmSource receiving methods

```
package com.tailf.ncs.alarmman.consumer;
...
public class AlarmSource {
    ...
    /**
     * Waits indefinitely for a new alarm or until the
     * queue is interrupted.
     *
     * @return a new alarm.
     * @throws InterruptedException
     */
    public Alarm takeAlarm() throws InterruptedException{
        ...
    }
    ...
    /**
     * Waits until the next alarm comes or until the time has expired.
     *
     * @param time time to wait.
     * @param unit
     * @return a new alarm or null if timeout expired.
     * @throws InterruptedException
     */
    public Alarm pollAlarm(int time, TimeUnit unit)
        throws InterruptedException{
        ...
    }
}
```

As soon as you create an alarm source object the alarm source object will start receiving alarms. If you do not poll or take any alarms from the alarm source object the queue will fill up until it reaches the maximum number of queued alarms as specified by the alarm source central. The alarm source central will then start to drop the oldest alarms until the alarm source starts the retrieval. This only affects the alarm source that is lagging behind. Any other alarm sources that are active at the same time will receive alarms without discontinuation.

Example 189. Consuming alarms

```
AlarmSource source = new AlarmSource();
Alarm lAlarm = mySource.pollAlarm();
while (lAlarm != null){
    //handle alarm
}
```

Extending the alarm manager, adding user defined alarm types and fields

The NSO alarm manager is extendable. NSO itself has a number of built-in alarms. The user can add user defined alarms. In the website example we have a small YANG module that extends the set of alarm types.

Extending the alarm manager, adding user defined alarm types and fields

We have in the module `my-alarms.yang` the following alarm type extension:

Example 190. Extending alarm-type

```
module my-alarms {
    namespace "http://examples.com/ma";
    prefix ma;

    .....

    import tailf-ncs-alarms {
        prefix al;
    }

    import tailf-common {
        prefix tailf;
    }

    identity website-alarm {
        base al:alarm-type;
    }

    identity webserver-on-fire {
        base website-alarm;
    }
}
```

The `identity` statement in the YANG language is used for this type of constructs. To complete our alarm type extension we also need to populate configuration data related to the new alarm type. A good way to do that is to provide XML data in a CDB initialization file and place this file in the `ncs-cdb` directory:

Example 191. my-alarms.xml

```
<alarms xmlns="http://tail-f.com/ns/ncs-alarms">
    <alarm-model>
        <alarm-type>
            <type
                xmlns:ma="http://examples.com/ma">ma:webserver-on-fire</type>
                <event-type>equipmentAlarm</event-type>
                <has-clear>true</has-clear>
                <kind-of-alarm>root-cause</kind-of-alarm>
                <probable-cause>957</probable-cause>
            </alarm-type>
        </alarm-model>
    </alarms>
```

Another possibility of extension is to add fields to the existing NSO alarms. This can be useful if you want add extra fields for attributes not directly supported by the NSO alarm list.

Below follows an example showing how to extend the alarm and the alarm status.

Example 192. Extending alarm model

```
module my-alarms {
    namespace "http://examples.com/ma";
    prefix ma;

    .....

    augment /al:alarms/al:alarm-list/al:alarm {
        leaf custom-alarm-attribute {
```

```

        type string;
    }
}

augment /al:alarms/al:alarm-list/al:alarm/al:status-change {
    leaf custom-status-change-attribute {
        type string;
    }
}
}

```

Mapping alarms to objects

One of the strengths of the NSO model structure is the correlation capabilities. Whenever NSO FASTMAP creates a new service it creates a back pointer reference to the service that caused the device modification to take place. NSO template based services will generate these pointers by default. For Java based services back pointers are created when the createdShared method is used. These pointers can be retrieved and used as input to the impacted objects parameter of an raised alarm.

The impacted objects of the alarm are the objects that are affected by the alarm i.e. depends on the alarming objects, or the root cause objects. For NSO this typically means services that have created the device configuration. An impacted object should therefore point to a service that may suffer from this alarm.

The root cause object is another important object of the alarm. It describes the object that likely is the original cause of the alarm. Note, that this is not the same thing as the alarming object. The alarming object is the object that raised the alarm, while the root cause object is the primary suspect for causing the alarm. In NSO any object can raise alarms, it may be a service, a device or something else.

Example 193. Finding back pointers for a given device path

```

private List<ManagedObject> findImpactedObjects(String path)
    throws ConfException, IOException
{
    List<ManagedObject> objs = new ArrayList<ManagedObject>();

    int th = -1;
    try {
        //A helper object that can return the topmost tag (not key)
        //and that can reduce the path by one tag at a time (parent)
        ExtConfPath p = new ExtConfPath(path);

        // Start a read transaction towards the running configuration.
        th = maapi.startTrans(Conf.DB_RUNNING, Conf.MODE_READ);

        while(!(p.topTag().equals("config")
            || p.topTag().equals("ncs:config"))){

            //Check for back pointer
            ConfAttributeValue[] vals = this.maapi.getAttrs(th,
                new ConfAttributeType[]{ConfAttributeType.BACKPOINTER},
                p.toString());

            for(ConfAttributeValue v : vals){
                ConfList refs = (ConfList)v.getAttributeValue();
                for (ConfObject co : refs.elements()){
                    ManagedObject mo = new ManagedObject((ConfObjectRef)co);
                    objs.add(mo);
                }
            }
        }
    }
}

```

```
        }

        p = p.parent();
    }
}
catch (IOException ioe){
    LOGGER.warn("Could not access Maapi, "
                +" aborting mapping attempt of impacted objects");
}
catch (ConfException ce){
    ce.printStackTrace();
    LOGGER.warn("Failed to retrieve Attributes via Maapi");
}
finally {
    maapi.finishTrans(th);
}
return objs;
}
```



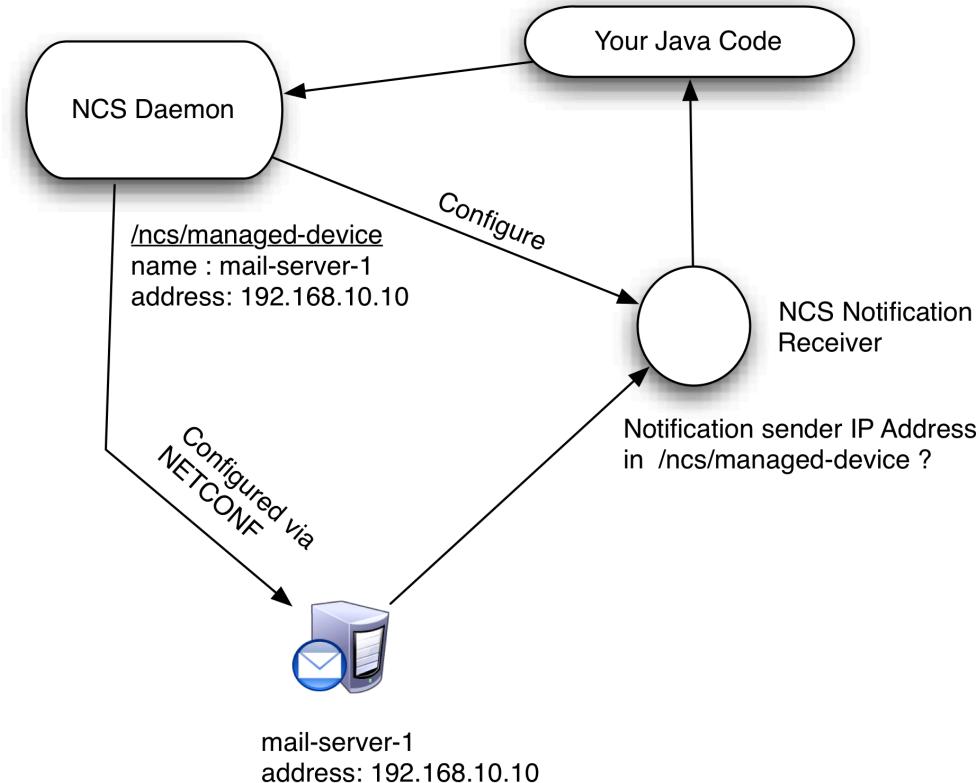
CHAPTER 17

SNMP Notification Receiver

- [Introduction, page 383](#)
- [Configuring NSO to Receive SNMP Notifications, page 384](#)
- [Built-in Filters, page 384](#)
- [Notification Handlers, page 385](#)

Introduction

NSO can act as an SNMP notification receiver (v1, v2c, v3) for its managed devices. The application can register notification handlers and react on the notifications, for example by mapping SNMP notifications to NSO alarms.



SNMP NED Compile Steps

The notification receiver is started in the Java VM by application code, as described below. The application code registers the handlers, which are invoked when a notification is received from a managed device. The NSO operator can enable and disable the notification receiver as needed. The notification receiver is configured in the `/snmp-notification-receiver` subtree.

By default nothing happens with SNMP Notifications. You need to register a function to listen to traps and do something useful with the traps. First of all SNMP var-binds are typically sparse in information and in many cases you want to do enrichment of the information and map the notification to some meaningful state. Sometimes a notification indicates an alarm state change, sometimes they indicate that the configuration of the device has changed. The action based on the two above examples are very different, in the first case you want to interpret the notification for meaningful alarm information and submit a call to the NSO Alarm Manager. In the second case you probably want to initiate a `check-sync`, `compare-config`, `sync` action sequence.

Configuring NSO to Receive SNMP Notifications

The NSO operator must enable the SNMP notification receiver, and configure the addresses NSO will use to listen for notifications. The primary parameters for the Notification receiver are shown below.

```
+--rw snmp-notification-receiver
  +-rw enabled?      boolean
  +-rw listen
    |  +-rw udp [ip port]
    |  +-rw ip       inet:ip-address
    |  +-rw port     inet:port-number
  +-rw engine-id?    snmp-engine-id
```

The notification reception can be turned on and off using the `enabled` leaf. NSO will listen to notifications at the end-points configured in `listen`. There is no need to manually configure the NSO `engine-id`. NSO will do this automatically using the algorithm described in RFC 3411. However, it can be assigned an `engine-id` manually by setting this leaf.

The managed devices must also be configured to send notifications to the NSO addresses.

NSO silently ignores any notification received from unknown devices. By default, NSO uses the `/devices/device/address` leaf, but this can be overridden by setting `/devices/device/snmp-notification-address`.

```
+--rw device [name]
  |  +-rw name          string
  |  +-rw address        inet:host
  |  +-rw snmp-notification-address?  inet:host
```

Built-in Filters

There are some standard built-in filters for the SNMP notification receiver which perform standard tasks.

Standard filter for suppression of received snmp events which are not of type TRAP, NOTIFICATION or INFORM.

Standard filter for suppression of notifications emanating from ip addresses outside defined set of addresses This filter determines the source ip address first from the `snmpTrapAddress` 1.3.6.1.6.3.18.1.3 varbind if this is set in the PDU, or otherwise from the emanating peer ip address. If the resulting ip address does not match either the `snmp-notification-address` or the `address` leaf of any device in the device-model this notification is discarded.

Standard filter that will acknowledge INFORM notification automatically.

Notification Handlers

NSO uses the Java package SNMP4J to parse the SNMP PDUs.

Notification Handlers are user supplied Java classes that implement the `com.tailf.snmp.snmp4j.NotificationHandler` interface. The `processPDU` method is expected to react on the SNMP4J event, e.g. by mapping the PDU to an NSO alarm. The handlers are registered in the `NotificationReceiver`. The `NotificationReceiver` is the main class that in addition to maintain the handlers also has responsibility to read the NSO SNMP notification configuration, and setup SNMP4J listeners accordingly.

An example of a notification handler can be found at `$NCS_DIR/examples.ncs/snmp-notification-receiver`. This example handler receives notifications and sets an alarm text if the notification is an IF-MIB::linkDown trap.

Example 194.

```
public class ExampleHandler implements NotificationHandler {

    private static Logger LOGGER = LogManager.getLogger(ExampleHandler.class);

    /**
     * This callback method is called when a notification is received from
     * Snmp4j.
     *
     * @param event
     *          a CommandResponderEvent, see Snmp4j javadoc for details
     * @param opaque
     *          any object passed in register()
     */
    public HandlerResponse
        processPdu(EventContext context,
                   CommandResponderEvent event,
                   Object opaque)
    throws Exception {

        String alarmText = "test alarm";

        PDU pdu = event.getPDU();
        for (int i = 0; i < pdu.size(); i++) {
            VariableBinding vb = pdu.get(i);
            LOGGER.info(vb.toString());

            if (vb.getOid().toString().equals("1.3.6.1.6.3.1.1.4.1.0")) {
                String linkStatus = vb.getVariable().toString();
                if ("1.3.6.1.6.3.1.1.5.3".equals(linkStatus)) {
                    alarmText = "IF-MIB::LinkDown";
                }
            }
        }

        String device = context.getDeviceName();
        String managedObject = "/devices/device{" + device + "}";
        ConfidentialityRef alarmType =
            new ConfidentialityRef(new NcsAlarms().hash(),
                                  NcsAlarms._connection_failure);
        PerceivedSeverity severity = PerceivedSeverity.MAJOR;
        ConfDatetime timeStamp = ConfDatetime.getConfDatetime();
    }
}
```

```

        Alarm al = new Alarm(new ManagedDevice(device),
                             new ManagedObject(managedObject),
                             alarmType,
                             severity,
                             false,
                             alarmText,
                             null,
                             null,
                             null,
                             null,
                             timeStamp);

        AlarmSink sink = new AlarmSink();
        sink.submitAlarm(al);

        return HandlerResponse.CONTINUE;
    }
}

```

The instantiation and start of the `NotificationReceiver` as well as registration of notification handlers are all expected to be done in the same application component of some NSO package. The following is an example of such an application component:

Example 195.

```

/**
 * This class starts the Snmp-notification-receiver.
 */
public class App implements ApplicationComponent {

    private ExampleHandler handl = null;
    private NotificationReceiver notifRec = null;

    static {
        LogFactory.setLogFactory(new Log4jLogFactory());
    }

    public void run() {
        try {
            notifRec.start();
            synchronized (notifRec) {
                notifRec.wait();
            }
        } catch (Exception e) {
            NcsMain.reportPackageException(this, e);
        }
    }

    public void finish() throws Exception {
        if (notifRec == null) {
            return;
        }
        synchronized (notifRec) {
            notifRec.notifyAll();
        }
        notifRec.stop();
        NotificationReceiver.destroyNotificationReceiver();
    }

    public void init() throws Exception {
        handl = new ExampleHandler();
    }
}

```

```
    notifRec =
        NotificationReceiver.getNotificationReceiver();
    // register example filter
    notifRec.register(handl, null);
}
}
```




CHAPTER 18

The web server

- [Introduction, page 389](#)
- [Web server capabilities, page 389](#)
- [CGI support, page 390](#)
- [Storing TLS data in database, page 391](#)

Introduction

This document describes an embedded basic web server that can deliver static and Common Gateway Interface (CGI) dynamic content to a web client, commonly a browser. Due to the limitations of this web server, and/or of its configuration capabilities, a proxy server such as Nginx is recommended to address special requirements.

Web server capabilities

The web server can be configured through settings in ncs.conf - see the manual pages of the section called “CONFIGURATION PARAMETERS” in *NSO 5.7 Manual Pages*.

Here is a brief overview of what you can configure on the web server:

- "toggle web server": the web server can be turned on or off
- "toggle transport": enable HTTP and/or HTTPS, set IPs, ports, redirects, certificates, etc.
- "hostname": set the hostname of the web server and decide whether to block requests for other hostnames
- "/": set the docroot from where all static content is served
- "/login": set the docroot from where static content is served for URL paths starting with /login
- "/custom": set the docroot from where static content is served for URL paths starting with /custom
- "/cgi": toggle CGI support and set the docroot from where dynamic content is served for URL paths starting with /cgi
- "non-authenticated paths": by default all URL paths, except those needed for the login page are hidden from non-authenticated users; authentication is done by calling the JSONRPC "login" method
- "allow symlinks": allow symlinks from under the docroot
- "cache": set the cache time window for static content
- "log": several logs are available to configure in terms of file paths - an access log, a full HTTP traffic/trace log and a browser/JavaScript log

- "custom headers": set custom headers across all static and dynamic content, including requests to "/jsonrpc".

In addition to what is configurable, the web server also GZip-compresses responses automatically if the browser handles such responses, either by compressing the response on the fly, or, if requesting a static file, like "/bigfile.txt", by responding with the contents of "/bigfile.txt.gz", if there is such a file.

CGI support

The web server includes CGI functionality, disabled by default. Once you enable it in ncs.conf - see the manual pages of the section called "CONFIGURATION PARAMETERS" in *NSO 5.7 Manual Pages*, you can write CGI scripts, that will be called with the following NSO environment variables prefixed with NCS_ when a user has logged-in via JSON-RPC:

- "JSONRPC_SESSIONID": the JSON-RPC session id (cookie)
- "JSONRPC_START_TIME": the start time of the JSON-RPC session
- "JSONRPC_END_TIME": the end time of the JSON-RPC session
- "JSONRPC_READ": the latest JSON-RPC read transaction
- "JSONRPC_READS": a comma-separated list of JSON-RPC read transactions
- "JSONRPC_WRITE": the latest JSON-RPC write transaction
- "JSONRPC_WRITES": a comma-separated of JSON-RPC write transactions
- "MAAPI_USER": the MAAPI username
- "MAAPI_GROUPS": a comma-separated list of MAAPI groups
- "MAAPI_UID": the MAAPI UID
- "MAAPI_GID": the MAAPI GID
- "MAAPI_SRC_IP": the MAAPI source IP address
- "MAAPI_SRC_PORT": the MAAPI source port
- "MAAPI_USID": the MAAPI USID
- "MAAPI_READ": the latest MAAPI read transaction
- "MAAPI_READS": a comma-separated list of MAAPI read transactions
- "MAAPI_WRITE": the latest MAAPI write transaction
- "MAAPI_WRITES": a comma-separated of MAAPI write transactions

Server or HTTP specific information is also exported as environment variables:

- "SERVER_SOFTWARE":
- "SERVER_NAME":
- "GATEWAY_INTERFACE":
- "SERVER_PROTOCOL":
- "SERVER_PORT":
- "REQUEST_METHOD":
- "REQUEST_URI":
- "DOCUMENT_ROOT":
- "DOCUMENT_ROOT_MOUNT":
- "SCRIPT_FILENAME":
- "SCRIPT_TRANSLATED":
- "PATH_INFO":
- "PATH_TRANSLATED":

- "SCRIPT_NAME":
- "REMOTE_ADDR":
- "REMOTE_HOST":
- "SERVER_ADDR":
- "LOCAL_ADDR":
- "QUERY_STRING":
- "CONTENT_TYPE":
- "CONTENT_LENGTH":
- "HTTP_*": HTTP headers e.g. "Accept" value is exported as HTTP_ACCEPT

Storing TLS data in database

The `tailf-tls.yang` YANG module defines a structure to store TLS data in the database. It is possible to store the private key, the private key's passphrase, the public key certificate, and CA certificates.

In order to enable the web server to fetch TLS data from the database, `ncs.conf` needs to be configured

Example 196. Configuring NSO to read TLS data from database

```
<webui>
  <transport>
    <ssl>
      <enabled>true</enabled>
      <ip>0.0.0.0</ip>
      <port>8889</port>
      <read-from-db>true</read-from-db>
    </ssl>
  </transport>
</webui>
```

Note that the options `key-file`, `cert-file`, and `ca-cert-file`, are ignored when `read-from-db` is set to true. See the `ncs.conf.5` man page for more details.

The database is populated with TLS data by configuring the `/tailf-tls:tls/private-key`, `/tailf-tls:tls/certificate`, and, optionally, `/tailf-tls/ca-certificates`. It is possible to use password protected private keys, then the `passphrase` leaf in the `private-key` container needs to be set to the password of the encrypted private key. Unencrypted private key data can be supplied in both PKCS#8 and PKCS#1 format, while encrypted private key data needs to be supplied in PKCS#1 format.

In the following example a password protected private key, the passphrase, a public key certificate, and two CA certificates are configured with the CLI.

Example 197. Populating the database with TLS data

```
admin@io> configure
Entering configuration mode private
[ok][2019-06-10 19:54:21]

[edit]
admin@io% set tls certificate cert-data
(<unknown>):
[Multiline mode, exit with ctrl-D.]
> -----BEGIN CERTIFICATE-----
```

■ Storing TLS data in database

```

> MIICrzCCAZcCFBh0ETLcNAFCCEcjSrrd5U4/a6vuMA0GCSqGSIB3DQEBCwUAMBQx
> ...
> -----END CERTIFICATE-----
>
[ok][2019-06-10 19:59:36]

[edit]
admin@confd% set tls private-key key-data
(<unknown>):
[Multiline mode, exit with ctrl-D.]
> -----BEGIN RSA PRIVATE KEY-----
> Proc-Type: 4,ENCRYPTED
> DEK-Info: AES-128-CBC,6E816829A93AAD3E0C283A6C8550B255
> ...
> -----END RSA PRIVATE KEY-----
[ok][2019-06-10 20:00:27]

[edit]
admin@confd% set tls private-key passphrase
(<AES encrypted string>): *****
[ok][2019-06-10 20:00:39]

[edit]
admin@confd% set tls ca-certificates ca-cert-1 cert-data
(<unknown>):
[Multiline mode, exit with ctrl-D.]
> -----BEGIN CERTIFICATE-----
> MIIDCTCCAfGgAwIBAgIUbzrNvBdM7p2rxwDBaqF5xN1gfmEwDQYJKoZIhvcNAQEL
> ...
> -----END CERTIFICATE-----
[ok][2019-06-10 20:02:22]

[edit]
admin@confd% set tls ca-certificates ca-cert-2 cert-data
(<unknown>):
[Multiline mode, exit with ctrl-D.]
> -----BEGIN CERTIFICATE-----
> MIIDCTCCAfGgAwIBAgIUZ2GcDzHg44c2g7Q0Xlu3H8/4wnwwDQYJKoZIhvcNAQEL
> ...
> -----END CERTIFICATE-----
[ok][2019-06-10 20:03:07]

[edit]
admin@confd% commit
Commit complete.
[ok][2019-06-10 20:03:11]

[edit]
```

The SHA256 fingerprints of the public key certificate and the CA certificates can be accessed as operational data. The fingerprint is shown as a hex string. The first octet identifies what hashing algorithm is used, *04* is SHA256, and the following octets is the actual fingerprint.

Example 198. Show TLS certificate fingerprints

```

admin@io> show tls
tls certificate fingerprint 04:65:8a:9e:36:2c:a7:42:8d:93:50:af:97:08:ff:e6:1b:c5:43:a8:2c:b5:bf
NAME      FINGERPRINT
-----
cacert-1  04:00:5e:22:f8:4b:b7:3a:47:e7:23:11:80:03:d3:9a:74:8d:09:c0:fa:cc:15:2b:7f:81:1a:e6:80
cacert-2  04:2d:93:9b:37:21:d2:22:74:ad:d9:99:ae:76:b6:6a:f2:3b:e3:4e:07:32:f2:8b:f0:63:ad:21:7d
```

```
[ok][2019-06-10 20:43:31]
```

When the database is populated NSO needs to be reloaded.

```
$ ncs --reload
```

After configuring NSO, populating the database, and reloading, the TLS transport is usable.

```
$ curl -kisu admin:admin https://localhost:8889
HTTP/1.1 302 Found
...
```




CHAPTER 19

Kicker

- [Introduction, page 395](#)
- [Kicker action invocation, page 395](#)
- [Data Kicker Concepts, page 396](#)
- [Notification Kicker Concepts, page 402](#)
- [Reactive FastMap with Kicker, page 405](#)
- [Debugging kickers, page 409](#)

Introduction

Kickers constitutes a declarative notification mechanism for triggering actions on certain stimuli like a database change or a received notification. These different stimuli and their kickers are defined separately as *data-kicker* and *notification-kicker* respectively.

Common to all types of kickers is that they are declarative. Kickers are modeled in YANG and Kicker instances stored as configuration data in CDB.

Immediately after a transaction which defines a new kicker is committed the kicker will be active. The same holds for removal. This also implies that the amount of programming for a kicker is a matter of implementing the action to be invoked.

The *data-kicker* replicates much of the functionality otherwise attained by a CDB subscriber. Without the extra coding in registration and runtime daemon that comes with a CDB subscriber. The *data-kicker* works for all data providers.

The *notification-kicker* reacts on notifications received by NSO using a defined notification subscription under `/ncs:devices/device/notifications/subscription`. This simplifies handling of southbound emitted notifications. Traditionally these were chosen to be stored in CDB as operational data and a separate CDB subscriber was used to act on the received notifications. With the use of *notification-kicker* the CDB subscriber can be removed and there is no longer any need to store the received notification in CDB.

Kicker action invocation

An action as defined by YANG contains an input parameter definition and an output parameter definition. However a kicker that invokes an action treats the input parameters in a specific way.

The kicker mechanism first checks if the input parameters matches those in the `kicker:action-input-params` YANG grouping defined in the `tailf-kicker.yang` file. If so the action will be invoked with the input parameters:

- `id` The id (name) of the invoking kicker.
- `monitor` The path of the current monitor triggering the kicker
- `tid` The transaction id to a synthetic transaction containing the changes that lead to the triggering of the kicker.

The "synthetic" transaction implies that this is a copy of the original transaction that lead to the kicker triggering. It only contains the data tree under the monitor. The original transaction is already committed and this data might no longer reflect the "running" datastore. Its useful in that the action implementation can attach and diff-iterate over this transaction and retrieve the certain changes that lead to the kicker invocation.

If the kicker mechanism finds an action that do not match the above input parameters it will invoke the action with an empty parameter list. This implies that an kicker action must either match the above `kicker:action-input-params` grouping precisely or accept an empty incoming parameter list. Otherwise the action invocation will fail.

Data Kicker Concepts

For a Data Kicker the following principles hold:

- Kicker are triggered by changes in the sub-tree indicated by the `monitor`. potentially triggers a Kicker.
- Actions are invoked during the commit phase. Hence an aborted transaction never trigger kickers.
- No distinction is made between configuration and operational data.
- No distinction is made between CRUD types, i.e. create, delete, update. All changes potentially trigger kickers.
- Kickers may have constraints that suppress invocations. Changes in the sub-tree indicated by `monitor` is a necessary but perhaps not a sufficient condition for the action to be invoked.

Generalized Monitors

For a Data Kicker it is the `monitor` that specifies which subtree under which a change should invoke the kicker. The `monitor` leaf is of type `node-instance-identifier` which means that predicates for keys are optional, i.e. keys may be omitted and then represent all instances for that key.

The resulting evaluation of the monitor defines a node-set. Each node in this node-set will be root context for any further xpath evaluations necessary before invoking the kicker action.

The following example shows the strengths of using xpath to define the kickers. Say that we have a situation described by the following YANG model snippet:

```
module example {
  namespace "http://tail-f.com/ns/test/example";
  prefix example;

  ...

  container sys {
    list ifc {
      key name;
      max-elements 64;
      leaf name {
```

```
    type interfaceName;
}
leaf description {
    type string;
}
leaf enabled {
    type boolean;
    default true;
}
container hw {
    leaf speed {
        type interfaceSpeed;
    }
    leaf duplex {
        type interfaceDuplex;
    }
    leaf mtu {
        type mtuSize;
    }
    leaf mac {
        type string;
    }
}
list ip {
    key address;
    max-elements 1024;
    leaf address {
        type inet:ipv4-address;
    }
    leaf prefix-length {
        type prefixLengthIPv4;
        mandatory true;
    }
    leaf broadcast {
        type inet:ipv4-address;
    }
}
tailf:action local_me {
    tailf:actionpoint kick-me-point;
    input {
    }
    output {
    }
}
tailf:action kick_me {
    tailf:actionpoint kick-me-point;
    input {
    }
    output {
    }
}
tailf:action iter_me {
    tailf:actionpoint kick-me-point;
    input {
        uses kicker:action-input-params;
    }
    output {
    }
}
```

```

        }
    }
}
```

Then we can define a kicker for monitoring a specific element in the list and calling the correlated local_me action:

```
admin@ncs(config)# kickers data-kicker e1 \
> monitor /sys/ifc[name='port-0'] \
>kick-node /sys/ifc[name='port-0']\
> action-name local_me

admin(config-data-kicker-e1)# commit
Commit complete
admin(config-data-kicker-e1)# top
admin@ncs(config)# show full-configuration kickers
kickers data-kicker e1
  monitor      /sys/ifc[name='port-0']
  kick-node   /sys/ifc[name='port-0']
  action-name local_me
!
```

On the other hand we can define a kicker for monitoring all elements of the list and call the correlated local_me action for each element:

```
admin@ncs(config)# kickers data-kicker e2 \
> monitor /sys/ifc \
>kick-node . \
> action-name local_me

admin(config-data-kicker-e2)# commit
Commit complete
admin(config-data-kicker-e2)# top
admin@ncs(config)# show full-configuration kickers
kickers data-kicker e2
  monitor      /sys/ifc
  kick-node   .
  action-name local_me
!
```

Here the "." in the kick-node refer to the current node in the node-set defined by the monitor.

Kicker Constraints/Filters

A Data Kicker may be constrained by adding conditions that suppress invocations. The leaf trigger-expression contains a boolean XPath expression that is evaluated twice, before and after the change-set of the commit has been applied to the database(s).

The Xpath expression has to evaluated twice in order to detect the *change* caused by the transaction.

The two boolean results together with the leaf trigger-type controls if the Kicker should be triggered or not:

enter-and-leave	false -> true (i.e. positive flank) or true -> false (negative flank)
enter	false -> true

```
admin(config)# kickers data-kicker k1 monitor /sys/ifc \
> trigger-expr "hw/mtu > 800" \
```

```
> trigger-type enter \
> kick-node /sys \
> action-name kick_me
admin(config-data-kicker-k1)# commit
Commit complete
admin(config-data-kicker-k1)# top
admin@ncs%
admin@ncs% show kickers
kickers data-kicker k1
  monitor      /sys/ifc
  trigger-expr "hw/mtu > 800"
  trigger-type enter
  kick-node    /sys
  action-name   kick_me
!
```

Start by changing the MTU to 800:

```
admin(config)# sys ifc port-0 hw mtu 800
admin(config-ifc-port-0)# commit | debug kicker
2017-02-15T16:35:36.039 kicker: k1 at /kicker_example:sys/kicker_example:ifc[kicker_example:invoking 'kick_me' trigger-expr false -> false
Commit complete.
```

Since the trigger-expression evaluates to false, the kicker is not triggered. Let's try again:

```
admin(config)# sys ifc port-0 hw mtu 801
admin(config-ifc-port-0)# commit | debug kicker
2017-02-15T16:35:36.039 kicker: k1 at /kicker_example:sys/kicker_example:ifc[kicker_example:invoking 'kick-me' trigger-expr false -> true
Commit complete.
```

Variable Bindings

A Data Kicker may be provided with a list of variables (named values). Each variable binding consists of a name and a XPath expression. The Xpath expressions are evaluated on-demand, i.e. when used in either of monitor or trigger-expression nodes.

```
admin@ncs(config)# set kickers data-kicker k3 monitor $PATH/c
                           kick-node /x/y[id='n1']
                           action-name kick-me
                           variable PATH value "/a/b[k1=3][k2='3']"
admin@ncs(config)#

```

In the example above PATH is defined and referred to by the monitor expression by using the expression \$PATH.



Note A monitor expression is not evaluated by the XPath engine. Hence no trace of the evaluation can be found in the the Xpath log.

Monitor expressions are expanded and installed in an internal data-structure at kicker creation/compile time. XPath may be used while defining kickers by referring to a named XPath expression.

A Simple Data Kicker Example

This example is part of the examples.ncs/web-server-farm/web-site-service example. It consists of an action and a README_KICKER file. For all kickers defined in this example the same action

A Simple Data Kicker Example

is used. This action is defined in the website-service package. The following is the yang snippet for the action definition from the `website.yang` file:

```
module web-site {
    namespace "http://examples.com/web-site";
    prefix wse;

    ...

    augment /ncs:services {

        ...

        container actions {
            tailf:action diffcheck {
                tailf:actionpoint diffcheck;
                input {
                    uses kicker:action-input-params;
                }
                output {
                }
            }
        }
    }
}
```

The implementation of the action can be found in the `WebSiteServiceRFS.java` class file. Since it takes the `kicker:action-input-params` as input, the "Tid" for the synthetic transaction is available. This transaction is attached and diff-iterated. The result of the diff-iteration is printed in the `ncs-java-vm.log`:

```
class WebSiteServiceRFS {

    ...

    @ActionCallback(callPoint="diffcheck", callType=ActionCBType.ACTION)
    public ConfXMLParam[] diffcheck(DpActionTrans trans, ConfTag name,
                                    ConfObject[] kp, ConfXMLParam[] params)
    throws DpCallbackException {
        try {

            System.out.println("-----");
            System.out.println(params[0]);
            System.out.println(params[1]);
            System.out.println(params[2]);

            ConfUInt32 val = (ConfUInt32) params[2].getValue();
            int tid = (int)val.longValue();

            Socket s3 = new Socket("127.0.0.1", Conf.NCS_PORT);
            Maapi maapi3 = new Maapi(s3);
            maapi3.attach(tid, -1);

            maapi3.diffIterate(tid, new MaapiDiffIterate() {
                // Override the Default iterate function in the TestCase class
                public DiffIterateResultFlag iterate(ConfObject[] kp,
                                                    DiffIterateOperFlag op,
                                                    ConfObject oldValue,
                                                    ConfObject newValue,
                                                    Object initstate) {
                    System.out.println("path = " + new ConfPath(kp));
                    System.out.println("op = " + op);
                }
            });
        }
    }
}
```

```
        System.out.println("newValue = " + newValue);
        return DiffIterateResultFlag.ITER_RECURSE;
    }

});

maapi3.detach(tid);
s3.close();

return new ConfXMLParam[ ]{};
}

} catch (Exception e) {
    throw new DpCallbackException("diffcheck failed", e);
}
}
```

We are now ready to start the website-service example and define our data-kicker. Do the following:

```
$ make all
$ ncs-netsim start
$ ncs
$ ncs_cli -C -u admin

admin@ncs# devices sync-from
sync-result {
    device lb0
    result true
}
sync-result {
    device www0
    result true
}
sync-result {
    device www1
    result true
}
sync-result {
    device www2
    result true
}
```

The kickers are defined under the hide-group "debug". To be able to show and declare kickers we need first to unhide this hide-group:

```
admin@ncs# config  
admin@ncs(config)# unhide debug
```

We now define a data-kicker for the "profile" list under the by the service augmented container "/services/properties/wsp:web-site":

```
admin@ncs(config)# kickers data-kicker al \
> monitor /services/properties/wsp:web-site/profile \
> kick-node /services/wse:actions action-name diffcheck

admin@ncs(config-data-kicker-al)# commit
admin@ncs(config-data-kicker-al)# top
admin@ncs(config)# show full-configuration kickers data-kicker al
kickers data-kicker al
  monitor      /services/properties/wsp:web-site/profile
```

Notification Kicker Concepts

```
kick-node    /services/wse:actions
action-name diffcheck
!
```

We now commit a change in the profile list and we use the "debug kicker" pipe option to be able to follow the kicker invocation:

```
admin@ncs(config)# services properties web-site profile lean lb lb0
admin@ncs(config-profile-lean)# commit | debug kicker
2017-02-15T16:35:36.039 kicker: a1 at /ncs:services/ncs:properties/wsp:web-site/wsp:profile[wsp
Commit complete.

admin@ncs(config-profile-lean)# top
admin@ncs(config)# exit
```

We can also check the result of the action by looking into the ncs-java-vm.log:

```
admin@ncs# file show logs/ncs-java-vm.log
```

In the end we will find the following printout from the diffcheck action:

```
-----
{[669406386|id], a1}
{[669406386|monitor], /ncs:services/properties/web-site/profile{lean}}
{[669406386|tid], 168}
path = /ncs:services/properties/wsp:web-site/profile{lean}
op = MOP_CREATED
newValue = null
path = /ncs:services/properties/wsp:web-site/profile{lean}/name
op = MOP_VALUE_SET
newValue = lean
path = /ncs:services/properties/wsp:web-site/profile{lean}/lb
op = MOP_VALUE_SET
newValue = lb0
[ok][2017-02-15 17:11:59]
```

Notification Kicker Concepts

For a Notification Kicker the following principles hold:

- Notification Kickers are triggered by the arrival of notifications from any device subscription. These subscriptions are defined under the `/devices/device/notification/subscription` path.
- Storing the received notifications in CDB is optional and not part of the notification kicker functionality.
- The kicker invocations are serialized under a certain subscription i.e. kickers are invoked in the same sequence as notifications are received for the same subscription. This means that invocations are queued up and executed as quickly as the action permits.

Notification selector expression

The notification kicker is defined using a mandatory "selector-expr" which is an XPATH 1.0 expression. When the notification is received a synthetic transaction is started and the notification is written as if it would be stored under the path `/devices/device/notification/received-notifications/data`. Actually storing the notification in CDB is optional. The `selector-expr` is evaluated with the notification node as the current context and '/' as the root context. For example, if the device model defines a notification like this:

```
module device {
```

```

...
notification mynotif {
    leaf message {
        type string;
    }
}
...
}

```

the notification node 'mynotif' will be the current context for the `selector-expr`. There are four predefined variable bindings used when evaluating this expression:

<code>DEVICE</code>	The name of the device emitting the current notification.
<code>SUBSCRIPTION_NAME</code>	The name of the current subscription from which the notification was received. the kicker
<code>NOTIFICATION_NAME</code>	The name of the current notification.
<code>NOTIFICATION_NS</code>	The namespace of the current notification.

The `selector-expr` technique for defining the notification kickers is very flexible. For instance a kicker can be defined:

- To receive all notifications for a device.
- To receive all notifications of a certain type for any device.
- To receive a subset of notifications of a subset of devices by the use of specific subscriptions with the same name in several devices.

In addition to this usage of the predefined variable bindings it is possible to further drill down into the specific notification to trigger on certain leafs in the notification.

Variable Bindings

In addition to the four variable bindings mentioned above, a Notification Kicker may also be provided with a list of variables (named values). Each variable binding consists of a name and a XPath expression. The Xpath expression is evaluated when the `selector-expr` is run.

```

admin@ncs(config)# set kickers notification-kicker k4
selector-expr "$NOTIFICATION_NAME=linkUp and address[ip=$IP]"
kick-node /x/y[id='n1']
action-name kick-me
variable IP value '192.168.128.55'
admin@ncs(config)#

```

In the example above PATH is defined and referred to by the monitor expression by using the expression `$PATH`.



Note A monitor expression is not evaluated by the XPath engine. Hence no trace of the evaluation can be found in the the Xpath log.

Monitor expressions are expanded and installed in an internal data-structure at kicker creation/compile time. XPath may be used while defining kickers by referring to a named XPath expression.

A Simple Notification Kicker Example

In this example we use the same action and setup as in the Data kicker example above. The procedure for starting is also the same.

A Simple Notification Kicker Example

The website-service example has devices that has notifications generated on the stream "interface". We start with defining the notification kicker for a certain SUBSCRIPTION_NAME = "mysub". This subscription does not exist for the moment and the kicker will therefore not be triggered:

```
admin@ncs# config

admin@ncs(config)# kickers notification-kicker n1 \
> selector-expr "$SUBSCRIPTION_NAME = 'mysub'" \
> kick-node /services/wse:actions \
> action-name diffcheck

admin@ncs(config-notification-kicker-n1)# commit
admin@ncs(config-notification-kicker-n1)# top

admin@ncs(config)# show full-configuration kickers notification-kicker n1
kickers notification-kicker n1
  selector-expr "$SUBSCRIPTION_NAME = 'mysub'"
  kick-node      /services/wse:actions
  action-name    diffcheck
!
```

Now we define the "mysub" subscription on a device "www0" and refer to the notification stream "interface". As soon as this definition is committed the kicker will start triggering:

```
admin@ncs(config)# devices device www0 notifications subscription mysub \
> local-user admin stream interface
admin@ncs(config-subscription-mysub)# commit

admin@ncs(config-profile-lean)# top
admin@ncs(config)# exit
```

If we now inspect the ncs-java-vm.log we will see a number of notifications that are received. We also see that the transaction that is diff-iterated contains the notification as data under the path /devices/device/notifications/received-notifications/notification/data. This is a operational data list. However this transaction is synthetic and will not be committed. If the notification will be stored CDB is optional and not depending on the notification kicker functionality:

```
admin@ncs# file show logs/ncs-java-vm.log

-----
{[669406386|id], n1}
{[669406386|monitor], /ncs:devices/device{www0}/notifications.../data/linkUp}
{[669406386|tid], 758}
path = /ncs:devices/device{www0}
op = MOP_MODIFIED
newValue = null
path = /ncs:devices/device{www0}/notifications...
op = MOP_CREATED
newValue = null
path = /ncs:devices/device{www0}/notifications.../event-time
op = MOP_VALUE_SET
newValue = 2017-02-15T16:35:36.039204+00:00
path = /ncs:devices/device{www0}/notifications.../sequence-no
op = MOP_VALUE_SET
newValue = 0
path = /ncs:devices/device{www0}/notifications.../data/notif:linkUp
op = MOP_CREATED
newValue = null
path = /ncs:devices/device{www0}/notifications.../data/notif:linkUp/address{192.168.128.55}
op = MOP_CREATED
newValue = null
```

```

path = /ncs:devices/device{www0}/notifications.../data/notif:linkUp/address{192.168.128.55}/i
op = MOP_VALUE_SET
newValue = 192.168.128.55
path = /ncs:devices/device{www0}/notifications.../data/notif:linkUp/address{192.168.128.55}/m
op = MOP_VALUE_SET
newValue = 255.255.255.0
path = /ncs:devices/device{www0}/notifications.../data/notif:linkUp/ifName
op = MOP_VALUE_SET
newValue = eth2
path = /ncs:devices/device{www0}/notifications.../data/notif:linkUp/linkProperty{0}
op = MOP_CREATED
newValue = null
path = /ncs:devices/device{www0}/notifications.../data/notif:linkUp/linkProperty{0}/extensions
op = MOP_CREATED
newValue = 4668
path = /ncs:devices/device{www0}/notifications.../data/notif:linkUp/linkProperty{0}/extensions
op = MOP_VALUE_SET
newValue = 2
path = /ncs:devices/device{www0}/notifications.../data/notif:linkUp/linkProperty{0}/flags
op = MOP_VALUE_SET
newValue = 42
path = /ncs:devices/device{www0}/notifications.../data/notif:linkUp/linkProperty{0}/newlyAdded
op = MOP_CREATED
newValue = null

```

We end by removing the kicker and the subscription:

```

admin@ncs# config
admin@ncs(config)# no kickers notification-kicker
admin@ncs(config)# no devices device www0 notifications subscription
admin@ncs(config)# commit

```

Reactive FastMap with Kicker

This example illustrates how to write a reactive FastMap application using the Kicker. The idea is to let the service code create a kicker which redeploys a service when something happens.

Use Cases

Kickers are very often used as an implementation technique for Reactive FastMap services. Assume an NFV/ESC based application which:

- 1 Asks ESC to start a VM
- 2 Once the VM is ready, wants to configure the VM

Such an application would create a kicker with a monitor for `/devices/device[name=<vmname>]/ready`. Once the VM is 'ready', the service would be redeployed and it can continue its Reactive FastMap execution further and provision config to the newly started VM. Prior to the kickers, it was common with CDB subscriber code that

- 1 Subscribed to some change
- 2 Read that change, and then redeployed some service which the CDB subscriber code knew was waiting for that change

Now, with kickers we can simply such code by having a CDB subscriber that simply

- 1 Subscribes to some change (for example a device notification listener)
- 2 Writes some operational data field somewhere

The RFM service code is then responsible for setting up a kicker with the monitor pointing to that field written by the CDB subscriber. Thus effectively decoupling the CDB subscriber code from the RFM service code making them independent of each other. Another advantage is that the error handling code when the redeploy fails is unified inside the NSO kicker implementation.

RFM Example

The example can be found in `examples.ncs/getting-started/developing-with-ncs/21-kicker` and uses two NSO packages. The router package introduced in `../0-router-network` and a package called 'ppp-accounting' which is described in this section.

The example is a bit contrived, but since want to exemplify the usage of kickers, it's simplified and artificial.

```
$ ls ./packages
router ppp-accounting
```

To build the three packages, do

```
$ make all
```

To start the ncs-netsim network, follow the instructions in `../0-router-network`, it's the same network here.

```
$ ncs-netsim start
DEVICE ex0 OK STARTED
DEVICE ex1 OK STARTED
DEVICE ex2 OK STARTED
```

All the code for this example resides in `./packages/ppp-accounting`

To run the example we do:

```
$ ncs
```

This will start NSO, and NSO will load the two packages, load the data models defined by the two packages and start the Java code defined by the packages.

The service data model we have here looks like:

```
list ppp-accounting {
    uses ncs:service-data;
    ncs:servicepoint kickerspnt;

    key "interface";
    leaf interface {
        type string;
    }
}

list ppp-accounting-data {
    description "This is helper data, created by the service code for
    /ppp-accounting";

    key "interface";
    leaf interface {
        type string;
    }
    leaf accounting {
        description "populated externally";
```

```

        type string;
    }
}

```

The purpose of the service /ppp-accounting is to set the accounting field in the provided ppp interface on all routers in our example network. The catch here is that the name of the 'accounting' field is not provided as an input parameter to the service, instead it is populated externally and read and used by the service code.

The FastMap code tries to read the field /ppp-accounting-data[interface=<if>]/accounting and if it doesn't exist, the code creates a kicker on that field and returns. If the 'accounting' field exists, it is used and data is written into the /devices tree for our routers.

To run this we do:

```

$ make all
$ ncs-netsim start
$ ncs
$ ncs_cli -u admin
admin connected from 127.0.0.1 using console on mac
admin@ncs> request devices sync-from
sync-result {
    device ex0
    result true
}
sync-result {
    device ex1
    result true
}
sync-result {
    device ex2
    result true
}
[ok][2016-12-13 16:18:45]
admin@ncs> configure
Entering configuration mode private
[ok][2016-12-13 16:19:06]
[edit]
admin@ncs% set ppp-accounting ppp0
[ok][2016-12-13 16:20:01]

[edit]
admin@ncs% commit
Commit complete.
[ok][2016-12-13 16:20:04]

[edit]
admin@ncs% request ppp-accounting ppp0 get-modifications
cli {
    local-node {
        data
    }
}

```

We created the service, and verified that it didn't do anything. Looking at the code in packages/ppp-accounting/src/java/src/com/example/kicker/KickerServiceRFS.java we can see though that the code created a kicker.

Let's take a look at that:

```

admin@ncs% show kickers
-----^

```

```
syntax error: element does not exist
[error][2016-12-13 16:22:53]
```

The kicker data is hidden, and we cannot directly view it in the CLI. The `src/ncs/yang/tailf-kicker.yang` file says:

```
container kickers {
    tailf:info "Kicker specific configuration";
    tailf:hidden debug;

    list data-kicker {
        key id;
        ....
```

To view the kickers data we must do two things:

- 1 Provide an entry in the ncs.conf file
- 2 Unhide in the CLI

See [the section called “Unhide Kickers”](#) for details.

And now the kicker container is visible:

```
admin@ncs%$ show kickers
data-kicker ncs-internal-side-effects {
    monitor      /ncs:side-effect-queue;
    kick-node   /ncs:side-effect-queue;
    action-name invoke;
}
data-kicker ppp-accounting-ppp0 {
    monitor      /ppp-accounting-data[interface='ppp0']/accounting;
    kick-node   /ppp-accounting[interface='ppp0'];
    action-name reactive-re-deploy;
}
```

There we can see our newly created kicker.

To trigger this kicker, which will then execute the redeploy on the `/ppp-accounting[interface='ppp0']` service, all we need to do is to assign some data to the field that is monitored by the kicker.

```
admin@ncs%$ set ppp-accounting-data ppp0 accounting radius
[ok][2016-12-13 16:26:43]

[edit]
admin@ncs%$ commit
Commit complete.
[ok][2016-12-13 16:26:46]

[edit]

admin@ncs%$ request ppp-accounting ppp0 get-modifications
cli {
    local-node {
        data devices {
            device ex0 {
                config {
                    r:sys {
                        interfaces {
                            serial ppp0 {
                                ppp {
```

```

-
+           accounting acme;
+           accounting radius;
}
}
}
}
}
device ex1 {
    config {
        r:sys {
            interfaces {
                serial ppp0 {
                    ppp {
-
+                   accounting acme;
+                   accounting radius;

```

**Note**

Looking at the RFM java code we see that the /ppp-accounting-data help entry is created by a so called PRE_MODIFICATION hook. This is a common trick in RFM applications. We don't want that data to be part of the FastMap diffset. Usually the help entry is also used to contain various 'config false' fields pertaining to the service instance. If that data was part of FastMap diffset, the data would disappear with every redeploy turn, thus we use the PRE_MODIFICATION trick.

Debugging kickers

Kicker CLI Debug target

In order to find out why a Kicker kicked when it shouldn't or more commonly and annoying, why it didn't kick when it should, use the CLI pipe **debug kicker**.

Evaluation of potential Kicker invocations are reported in the CLI together with XPath evaluation results:

```

admin@ncs(config)# set sys ifc port-0 hw mtu 8000
admin@ncs(config)# commit | debug kicker
2017-02-15T16:35:36.039 kicker: k1 at /kicker_example:sys/kicker_example:ifc[kicker_example:
not invoking 'kick-me' trigger-expr false -> false
Commit complete.
admin@ncs(config)#

```

Unhide Kickers

The top level container **kickers** is by default invisible due to a hidden attribute. In order to make kickers visible in the CLI, two steps are required. First the following XML snippet must be added to **ncs.conf**:

```

<hide-group>
    <name>debug</name>
</hide-group>

```

Now the unhide command may be used in the CLI session:

```

admin@ncs(config)# unhide debug
admin@ncs(config)#

```

XPath log

Detailed information from the XPath evaluator can be enabled and made available in the xpath log. Add the following snippet to ncs.conf.

```
<xpathTraceLog>
  <enabled>true</enabled>
  <filename>./xpath.trace</filename>
</xpathTraceLog>
```

Devel Log

Error information is written to the development log. The development log is meant to be used as support while developing the application. It is enabled in ncs.conf:

Example 199. Enabling the developer log

```
<developer-log>
  <enabled>true</enabled>
  <file>
    <name>./logs-devel.log</name>
    <enabled>true</enabled>
  </file>
</developer-log>
<developer-log-level>trace</developer-log-level>
```



CHAPTER 20

Scheduler

- [Introduction, page 411](#)
- [Scheduling Periodic Work, page 411](#)
- [Scheduling Non-recurring Work, page 412](#)
- [Scheduling in a HA Cluster, page 412](#)
- [Troubleshooting, page 413](#)

Introduction

NSO includes a native time-based job scheduler suitable for scheduling background work. Tasks can be scheduled to run at particular times or periodically at fixed times, dates, or intervals. It can typically be used to automate system maintenance or administration-though tasks.

Scheduling Periodic Work

A standard Vixie Cron expression is used to represent the periodicity in which the task should run. When the task is triggered, the configured action is invoked on the configured action node instance. The action is run as the user that configured the task. To schedule a task to run sync-from on 2 AM on the 1st every month we do:

```
admin(config)# scheduler task sync schedule "0 2 1 * *" \
action-name sync-from action-node /devices
```



Note If the task was added through an XML init file the task will run with the `system` user, which implies that AAA rules will *not* be applied at all. Thus the task action will not be able to initiate device communication.

If the action node instance is given as an XPath 1.0 expression, the expression is evaluated with the root as the context node, and the expression must return a node set. The action is then invoked on each node in this node set.

Optionally action parameters can be configured in XML format to be passed to the action during invocation.

```
admin(config-task-sync)# action-params "<device>ce0</device><device>ce1</device>"
```

```
admin(config)# commit
```

Once the task has been configured you could view the next run times of the task:

```
admin(config)# scheduler task sync get-next-run-times display 3
next-run-time [ 2017-11-01 02:00:00+00:00 2017-12-01 02:00:00+00:00 2018-01-01 02:00:00+00:00 ]
```

You could also see if the task is running or not:

```
admin# show scheduler task sync is-running
is-running false
```

Schedule Expression

A standard Vixie Cron expression is a string comprising five fields separated by white space that represents a set of times. The following rules can be used to create an expression.

Table 200. Expression rules

Field	Allowed values	Allowed special characters
Minutes	0-59	* , - /
Hours	0-23	* , - /
Day of month	1-31	* , - /
Month	1-12 or JAN-DEC	* , - /
Day of week	0-6 or SUN-SAT	* , - /

The following list describes the legal special characters and how you can use them in a Cron expression.

- *Star* (*). Selects all values within a field. For example, * in the minute field selects every minute.
- *Comma* (,). Commas are used to specify additional values. For example, using MON,WED,FRI in the day of week field.
- *Hyphen* (-). Hyphens define ranges. For example 1-5 in the day of week field indicates every day between Monday and Friday, inclusive.
- *Forward slash* (/). Slashes can be combined with ranges to specify increments. For example, * /5 in the minutes field indicates every 5 minutes.

Scheduling Non-recurring Work

The scheduler can also be used to configure non-recurring tasks that will run at a particular time.

```
admin(config)# scheduler task my-compliance-report time 2017-11-01T02:00:00+01:00 \
action-name check-compliance action-node /reports
```

A non-recurring task will by default be removed when it has finished executing. It will be up to the action to raise an alarm if an error would occur. The task can also be kept in the task list by setting the keep leaf.

Scheduling in a HA Cluster

In a HA cluster a scheduled task will by default be run on the primary HA node. By configuring the ha-mode leaf a task can be scheduled to run on nodes with a particular HA mode, for example scheduling a read-only action on the secondary nodes. More specifically a task can be configured with the ha-node-id to only run on a certain node. These settings will not have any effect on a standalone node.

```
admin(config)# scheduler task my-compliance-report schedule "0 2 1 * *" \
ha-mode slave ha-node-id secondary-node1 \
action-name check-compliance action-node /reports
```

**Note**

The scheduler is disabled when HA is enabled and when HA mode is NONE. See the section called “Mode of operation” in *NSO 5.7 Administration Guide* for more details.

Troubleshooting

History log

In order to find out whether a scheduled task has run successfully or not, the easiest way is to view the history log of the scheduler. It will display the latest runs of the scheduled task.

```
admin# show scheduler task sync history | notab
history history-entry 2017-11-01T02:00:00.55003+00:00 0
  duration 0.15
  succeeded true
history history-entry 2017-12-01T02:00:00.549939+00:00 0
  duration 0.09
  succeeded true
history history-entry 2017-01-01T02:00:00.550128+00:00 0
  duration 0.01
  succeeded false
  info      "Resource device ce0 doesn't exist"
```

XPath log

Detailed information from the XPath evaluator can be enabled and made available in the xpath log. Add the following snippet to ncs.conf.

```
<xpathTraceLog>
  <enabled>true</enabled>
  <filename>./xpath.trace</filename>
</xpathTraceLog>
```

Devel Log

Error information is written to the development log. The development log is meant to be used as support while developing the application. It is enabled in ncs.conf:

```
<developer-log>
  <enabled>true</enabled>
  <file>
    <name>./logs-devel.log</name>
    <enabled>true</enabled>
  </file>
</developer-log>
<developer-log-level>trace</developer-log-level>
```

Suspending the Scheduler

While investigating a failure with a scheduled task or performing maintenance on the system, like upgrading, it might be useful to suspend the scheduler temporarily.

```
admin# scheduler suspend
```

When ready the scheduler can be resumed.

```
admin# scheduler resume
```

Suspending the Scheduler



CHAPTER 21

Progress Trace

- [Introduction, page 415](#)
- [Configuring Progress Trace, page 416](#)
- [Report Progress Events from User Code, page 417](#)

Introduction

Progress tracing in NSO provides developers with useful information for debugging, diagnostics and profiling. This information can be used both during development cycles and after release of the software. The system overhead for progress tracing are *usually* negligible.

When a transaction or action is applied, NSO emits progress events. These events can be displayed and recorded in a number of different ways. The easiest way is to pipe an action to details in the CLI.

```
admin@ncs% commit | details
Possible completions:
  debug verbose very-verbose
admin@ncs% commit | details
```

As seen by the details output, all events are recorded with a timestamp and in some cases with the duration. All phases of the transaction, service and device communication are printed.

```
2021-05-25T17:28:12.267 applying transaction...
entering validate phase for running usid=41 tid=1761 trace-id=d7f06482-41ad-4151-938d-7a8bc7b3
2021-05-25T17:28:12.267 grabbing transaction lock... ok (0.000 s)
2021-05-25T17:28:12.268 creating rollback file... ok (0.004 s)
2021-05-25T17:28:12.272 run transforms and transaction hooks...
2021-05-25T17:28:12.273 run pre-transform validation... ok (0.000 s)
2021-05-25T17:28:12.275 service /ordserv[name='o2']: run service... ok (0.035 s)
2021-05-25T17:28:12.311 run transforms and transaction hooks: ok (0.038 s)
2021-05-25T17:28:12.311 mark inactive... ok (0.000 s)
2021-05-25T17:28:12.311 pre validate... ok (0.000 s)
2021-05-25T17:28:12.311 run validation over the changeset... ok (0.000 s)
2021-05-25T17:28:12.312 run dependency-triggered validation... ok (0.000 s)
2021-05-25T17:28:12.312 check configuration policies... ok (0.000 s)
leaving validate phase for running usid=41 tid=1761 trace-id=d7f06482-41ad-4151-938d-7a8bc7b3
entering write-start phase for running usid=41 tid=1761 trace-id=d7f06482-41ad-4151-938d-7a8bc7b3
2021-05-25T17:28:12.312 cdb: write-start
2021-05-25T17:28:12.313 check data kickers... ok (0.000 s)
leaving write-start phase for running usid=41 tid=1761 trace-id=d7f06482-41ad-4151-938d-7a8bc7b3
entering prepare phase for running usid=41 tid=1761 trace-id=d7f06482-41ad-4151-938d-7a8bc7b3
2021-05-25T17:28:12.314 cdb: prepare
2021-05-25T17:28:12.314 ncs-internal-device-mgr: prepare
```

```

leaving prepare phase for running usid=41 tid=1761 trace-id=d7f06482-41ad-4151-938d-7a8bc7b3ce33
entering commit phase for running usid=41 tid=1761 trace-id=d7f06482-41ad-4151-938d-7a8bc7b3ce33
2021-05-25T17:28:12.317 cdb: commit
2021-05-25T17:28:12.318 ncs-internal-service-mux: commit
2021-05-25T17:28:12.318 ncs-internal-device-mgr: commit
2021-05-25T17:28:12.320 releasing transaction lock
leaving commit phase for running usid=41 tid=1761 trace-id=d7f06482-41ad-4151-938d-7a8bc7b3ce33
2021-05-25T17:28:12.320 applying transaction: ok (0.053 s)

```

Some actions (usually those involving device communication) also produces progress data.

```

admin@ncs% request devices device ce0 sync-from dry-run | details very-verbose
2021-05-25T17:31:31.222 device ce0: sync-from...
2021-05-25T17:31:31.222 device ce0: taking device lock... ok (0.000 s)
2021-05-25T17:31:31.227 device ce0: connect... ok (0.013 s)
2021-05-25T17:31:31.240 device ce0: show... ok (0.001 s)
2021-05-25T17:31:31.242 device ce0: get-trans-id... ok (0.000 s)
2021-05-25T17:31:31.242 device ce0: close... ok (0.000 s)
2021-05-25T17:31:31.248 device ce0: releasing device lock
2021-05-25T17:31:31.249 device ce0: sync-from: ok (0.026 s)

```

Configuring Progress Trace

The pipe details in the CLI is useful during development cycles of for example a service, but not as useful when tracing calls from other northbound interfaces or events in a released running system. Then it's better to configure a progress trace to be outputted to a file or operational data which can be retrieved through a northbound interface.

Unhide Progress Trace

The top level container `progress` is by default invisible due to a hidden attribute. In order to make `progress` visible in the CLI, two steps are required. First the following XML snippet must be added to `ncs.conf`:

```

<hide-group>
  <name>debug</name>
</hide-group>

```

Now the `unhide` command may be used in the CLI session:

```
admin@ncs% unhide debug
```

Log to File

Progress data can be outputted to a given file. This is useful when the data is to be analyzed in some third party software like a spreadsheet application.

```
admin@ncs% set progress trace test destination file event.csv format csv
```

The file can be formatted as a comma-separated values file defined by RFC 4180 or in a pretty printed log file with each event on a single line.

The location of the file is the directory of `/ncs-config/logs/progress-trace/dir` in `ncs.conf`.

Log as Operational Data

When the data is to be retrieved through a northbound interface it is more useful to output the progress events as operational data.

```
admin@ncs% set progress trace test destination oper-data
```

This will log non-persistent operational data to the /progress:progress/trace/event list. As this list might grow rapidly there is a maximum size of it (defaults to 1000 entries). When the maximum size is reached, the oldest list entry is purged.

```
admin@ncs% set progress trace test max-size 2000
```

Using the /progress:progress/trace/purge action the event list can be purged.

```
admin# request progress trace test purge
```

Log as Notification Events

Progress events can be subscribed to as Notifications events. See [the section called “NOTIF API”](#) for further details.

Verbosity

The *verbosity* parameter is used to control the level of output. The following levels are available:

- *normal* - Informational messages that highlight the progress of the system at a coarse-grained level. Used mainly to give a high level overview. This is the default and the lowest verbosity level.
- *verbose* - Detailed informational messages from the system. The various service and device phases and their duration will be traced. This is useful to get an overview over where time is spent in the system.
- *very-verbose* - Very detailed informational messages from the system and its internal operations.
- *debug* - The highest verbosity level. Fine-grained informational messages usable for debugging the system and its internal operations. Internal system transactions as well as data kicker evaluation and CDB subscribers will be traced. Setting this level could result in a large number of events being generated.

Additional debug tracing can be turned on for various parts. These are consciously left out of the normal debug level due to the high amount of output and should only be turned on during development.

Using Filters

By default all transaction and action events with the given verbosity level will be logged. To get a more selective choice of events, filters can be used.

```
admin@ncs% show progress trace filter
Possible completions:
  all-devices  - Only log events for devices.
  all-services - Only log events for services.
  context      - Only log events for the specified context.
  device       - Only log events for the specified device(s).
  device-group - Only log events for devices in this group.
  local-user   - Only log events for the specified local user.
  service-type - Only log events for the specified service type.
```

The context filter can be used to only log events that originate through a specific northbound interface. The context is either one of *netconf*, *cli*, *webui*, *snmp*, *rest*, *system* or it can be any other context string defined through the use of MAAPI.

```
admin@ncs% set progress trace test filter context netconf
```

Report Progress Events from User Code

API methods to report progress events exists for Java, Python and C. There also exist specific methods to report progress events for services.

Report Progress Events from User Code

```
...
Maapi maapi = service.context().getMaapi();
int tHandle = service.context().getMaapiHandle();
ConfPath servicePath = new ConfPath(service.getKeyPath());
maapi.reportServiceProgress(tHandle, Maapi.Verbosity.VERBOSE,
    "service test", servicePath);
```



CHAPTER 22

Nano Services for Staged Provisioning

Typical NSO services perform the necessary configuration by using the `create()` callback, within a transaction tracking the changes. This approach greatly simplifies service implementation, but it also introduces some limitations. For example, all provisioning is done at once, which may not be possible or desired in all cases. In particular, network functions implemented by containers or virtual machines often require provisioning in multiple steps.

Another limitation is that the service mapping code must not produce any side effects. Side effects are not tracked by the transaction and therefore cannot be automatically reverted. For example, imagine that there is an API call to allocate an IP address from an external system as part of the `create()` code. The same code runs for every service change or a service re-deploy, even during a **commit dry-run**, unless you take special precautions. So, a new IP address would be allocated every time, resulting in a lot of waste, or worse, provisioning failures.

Nano services help you overcome these limitations. They implement a service as several smaller (nano) steps or stages, by using a technique called reactive FASTMAP (RFM), and provide a framework to safely execute actions with side effects. Reactive FASTMAP can also be implemented directly, using the CDB subscribers, but nano services offer a more streamlined and robust approach for staged provisioning.

The chapter starts by gradually introducing the nano service concepts on a typical use-case. To aid readers working with nano services for the first time, some of the finer points are omitted in this part and discussed later on, in the section called “[Implementation Reference](#)”. The latter is designed as a reference to aid you during implementation, so it focuses on recapitulating the workings of nano services at the expense of examples. The rest of the chapter covers individual features with associated use-cases and the complete working examples, which you may find in the `examples.ncs` folder.

- [Basic Concepts, page 420](#)
- [Benefits and Use Cases, page 425](#)
- [Backtracking and Staged Delete, page 427](#)
- [Managing Side Effects, page 429](#)
- [Multiple and Dynamic Plan Components, page 431](#)
- [Netsim Router Provisioning Example, page 434](#)
- [Zombie Services, page 436](#)
- [Using Notifications to Track the Plan and its Status, page 437](#)
- [Developing and Updating a Nano Service, page 440](#)
- [Implementation Reference, page 442](#)

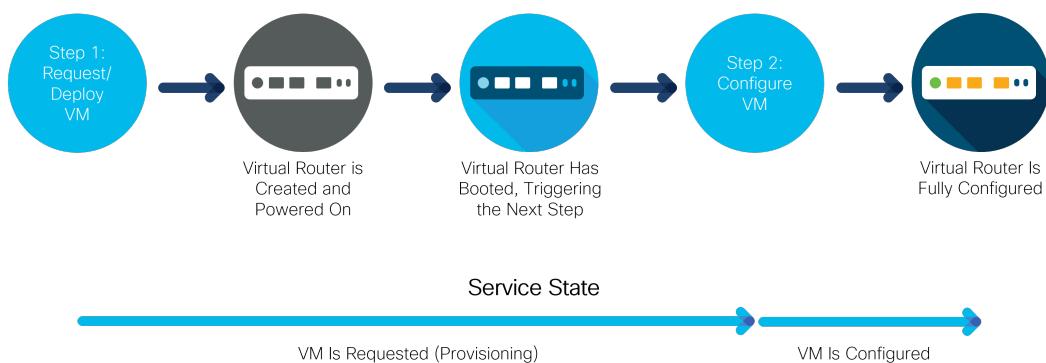
- Graceful Link Migration Example, page 455

Basic Concepts

Services ideally perform the configuration all at once, with all the benefits of a transaction, such as automatic rollback and cleanup on errors. For nano services, this is not possible in the general case. Instead, a nano service performs as much configuration as possible at the moment and leaves the rest for later. When an event occurs that allows more work to be done, the nano service instance restarts provisioning, by using a re-deploy action called `reactive-re-deploy`. It allows the service to perform additional configuration that was not possible before. The process of automatic re-deploy, called reactive FASTMAP, is repeated until the service is fully provisioned.

This is most evident with, for example, virtual machine (VM) provisioning, during virtual network function (VNF) orchestration. Consider a service that deploys and configures a router in a VM. When the service is first instantiated, it starts provisioning a router VM. However, it will likely take some time before the router has booted up and is ready to accept a new configuration. In turn, the service cannot configure the router just yet. The service must wait for the router to become ready. That is the event that triggers a re-deploy and the service can finish configuring the router, as the following figure illustrates:

Figure 201. Virtual router provisioning steps



While each step of provisioning happens inside a transaction and is still atomic, the whole service is not. Instead of a simple fully-provisioned or not-provisioned-at-all status, a nano service can be in a number of other *states*, depending on how far in the provisioning process it is.

The figure shows that the router VM goes through multiple states internally, however, only two states are important for the service. These two are shown as arrows, in the lower part of the figure. When a new service is configured, it requests a new VM deployment. Having completed this first step, it enters the “VM is requested but still provisioning” state. In the following step, the VM is configured and so enters the second state, where the router VM is deployed and fully configured. The states obviously follow individual provisioning steps and are used to report progress. What is more, each state tracks if an error occurred during provisioning.

For these reasons, service states are central to the design of a nano service. A list of different states, their order, and transitions between them is called a *plan outline* and governs the service behavior.

Plan Outline

The following YANG snippet, also part of the examples.ncs/development-guide/nano-services/basic-vrouter example, shows a plan outline with the two VM-provisioning states presented above:

```
module vrouter {
    prefix vr;
```

```

identity vm-requested {
    base ncs:plan-state;
}

identity vm-configured {
    base ncs:plan-state;
}

ncs:plan-outline vrouter-plan {
    description "Plan for configuring a VM-based router";

    ncs:component-type "ncs:self" {
        ncs:state "vr:vm-requested";
        ncs:state "vr:vm-configured";
    }
}
}

```

The first part contains a definition of states as identities, deriving from the ncs:plan-state base. These identities are then used with the ncs:plan-outline, inside an ncs:component-type statement. The YANG code defines a single ncs:self component, that tracks the progress of the service as a whole, but additional components can be used, as described later. Also note that it is customary to use past tense for state names, for example “configured-vm” or “vm-configured” instead of “configure-vm” and “configuring-vm.”

At present, the plan contains the two states but no logic. If you wish to do any provisioning for a state, the state must declare a special nano create callback, otherwise, it just acts as a checkpoint. The nano create callback is similar to an ordinary create service callback, allowing service code or templates to perform configuration. To add a callback for a state, extend the definition in the plan outline:

```

ncs:state "vr:vm-requested" {
    ncs:create {
        ncs:nano-callback;
    }
}

```

The service automatically enters each state one by one when a new service instance is configured. However, for the “vm-configured” state, the service should wait until the router VM has had the time to boot and is ready to accept a new configuration. An ncs:pre-condition statement in YANG provides this functionality. Until the condition becomes fulfilled, the service will not advance to that state.

The following YANG code instructs the nano service to check the value of the vm-up-and-running leaf, before entering and performing the configuration for a state.

```

ncs:state "vr:vm-configured" {
    ncs:create {
        ncs:nano-callback;
        ncs:pre-condition {
            ncs:monitor "$SERVICE" {
                ncs:trigger-expr "vm-up-and-running = 'true'";
            }
        }
    }
}

```

Per-State Configuration

The main reason for defining multiple nano service states is to specify what part of the overall configuration belongs in each state. For the VM-router example, that entails splitting the configuration into

a part for deploying a VM on a virtual infrastructure and a part for configuring it. In this case, a router VM is requested simply by adding an entry to a list of VM requests, while making the API calls is left to an external component, such as the VNF Manager.

If a state defines a nano callback, you can register a configuration template to it. The XML template file is very similar to an ordinary service template but requires the additional `componenttype` and `state` attributes in the `config-template` root element. These attributes identify which component and state in the plan outline the template belongs to, for example:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0"
    servicepoint="vrouter-servicepoint"
    componenttype="ncs:self"
    state="vr:vm-configured">
    <devices xmlns="http://tail-f.com/ns/ncs">
        <!-- ... -->
    </devices>
</config-template>
```

Likewise, you can implement a callback in the service code. The registration requires you to specify the component and state, as the following Python example demonstrates:

```
class NanoApp(ncs.application.Application):
    def setup(self):
        self.register_nano_service('vrouter-servicepoint', # Service point
                                   'ncs:self', # Component
                                   'vr:vm-requested', # State
                                   NanoServiceCallbacks)
```

The selected `NanoServiceCallbacks` class then receives callbacks in the `cb_nano_create()` function:

```
class NanoServiceCallbacks(ncs.application.NanoService):
    @ncs.application.NanoService.create
    def cb_nano_create(self, tctx, root, service, plan, component, state,
                      proplist, component_proplist):
        ...
```

The `component` and `state` parameters allow the function to distinguish calls for different callbacks when registered for more than one.

For most flexibility, each state defines a separate callback, allowing you to implement some with a template and others with code, all as part of the same service. You may even use Java instead of Python, as explained in the section called “[Nano Service Callbacks](#)”.

Link Plan Outline to Service

The set of states used in the plan outline describe the stages that a service instance goes through during provisioning. Naturally, these are service-specific, which presents a problem if you just want to tell whether a service instance is still provisioning or has already finished. It requires the knowledge of which state is the last, final one, making it hard to check in a generic way.

That is why each service component must have the built-in `ncs:init` state as the first state and `ncs:ready` as the last state. Using the two built-in states allows for interoperability with other services and tools. The following is a complete four-state plan outline for the VM-based router service, with the two states added:

```
ncs:plan-outline vrouter-plan {
    description "Plan for configuring a VM-based router";
```

```

ncs:component-type "ncs:self" {
    ncs:state "ncs:init";
    ncs:state "vr:vm-requested" {
        ncs:create {
            ncs:nano-callback;
        }
    }
    ncs:state "vr:vm-configured" {
        ncs:create {
            ncs:nano-callback;
            ncs:pre-condition {
                ncs:monitor "$SERVICE" {
                    ncs:trigger-expr "vm-up-and-running = 'true'";
                }
            }
        }
    }
    ncs:state "ncs:ready";
}
}

```

For the service to use it, the plan outline must be linked to a service point with the help of a *behavior tree*. The main purpose of a behavior tree is to allow a service to dynamically instantiate components, based on service parameters. Dynamic instantiation is not always required and the behavior tree for a basic, static, single-component scenario boils down to the following:

```

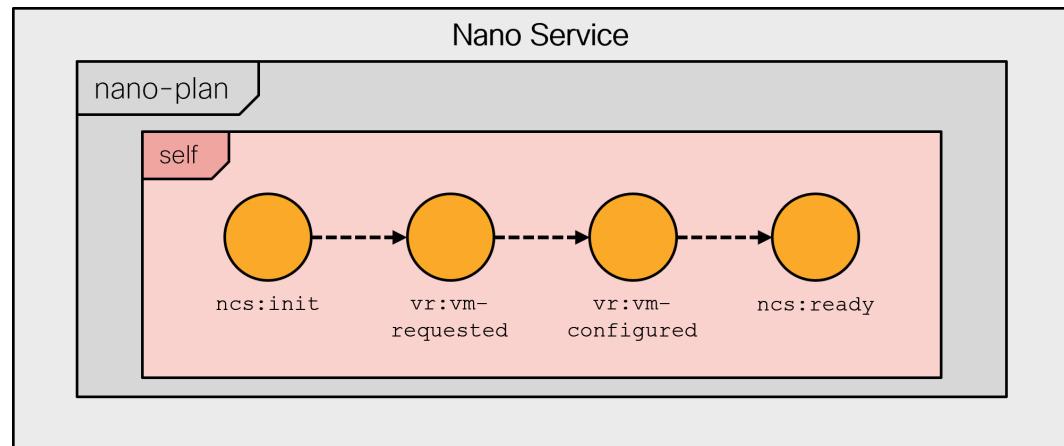
ncs:service-behavior-tree vrouter-servicepoint {
    description "A static, single component behavior tree";
    ncs:plan-outline-ref "vr:vrouter-plan";
    ncs:selector {
        ncs:create-component "'self'" {
            ncs:component-type-ref "ncs:self";
        }
    }
}

```

This behavior tree always creates a single “self” component for the service. The service point is provided as an argument to the ncs:service-behavior-tree statement, while the ncs:plan-outline-ref statement provides the name for the plan outline to use.

The following figure visualizes the resulting service plan and its states.

Figure 202. Virtual router provisioning plan



Along with the behavior tree, a nano service also relies on the ncs:nano-plan-data grouping in its service model. It is responsible for storing state and other provisioning details for each service instance. Other than that, the nano service model follows the standard YANG definition of a service:

```
list vrouter {
    description "Trivial VM-based router nano service";

    uses ncs:nano-plan-data;
    uses ncs:service-data;
    ncs:servicepoint vrouter-servicepoint;

    key name;
    leaf name {
        type string;
    }

    leaf vm-up-and-running {
        type boolean;
        config false;
    }
}
```

This model includes the operational `vm-up-and-running` leaf, that the example plan outline depends on. In practice, however, a plan outline is more likely to reference values provided by another part of the system, such as the actual, externally provided, state of the provisioned VM.

Service Instantiation

A nano service does not directly use its service point for configuration. Instead, the service point invokes a behavior tree to generate a plan, and the service starts executing according to this plan. As it reaches a certain state, it performs the relevant configuration for that state.

For example, when you create a new instance of the VM-router service, the `vm-up-and-running` leaf is not set, so only the first part of the service runs. Inspecting the service instance plan reveals the following:

```
admin@ncs# show vrouter vr-01 plan
      BACK
TYPE  NAME  TRACK  GOAL   STATE      STATUS      WHEN      ref  ACTION
-----+
self  self  false  -     init       reached    2021-09-16T14:04:38  -  POST
                           vm-requested  reached    2021-09-16T14:04:38  -  -
                           vm-configured not-reached -          -          -          -
                           ready      not-reached -          -          -          -
```

Since neither the “init” nor the “vm-requested” states have any pre-conditions, they are reached right away. In fact, NSO can optimize it into a single transaction (this behavior can be disabled if you use forced commits, discussed later on).

But the process has stopped at the “vm-configured” state, denoted by the `not-reached` status in the output. It is waiting for the pre-condition to become fulfilled with the help of a *kicker*. The job of the kicker is to watch the value and perform an action, the reactive re-deploy, when the conditions are satisfied. The kickers are managed by the nano service subsystem: when an unsatisfied precondition is encountered, a kicker is configured, and when the precondition becomes satisfied, the kicker is removed.

You may also verify, through the `get-modifications` action, that only the first part, the creation of the VM, was performed:

```
admin@ncs# vrouting vr-01 get-modifications
```

```

cli {
    local-node {
        data +vm-instance vr-01 {
            +      type csr-small;
            +
        }
    }
}

```

At the same time, a kicker was installed under the `kickers` container but you may need to use the `unhide debug` command to inspect it. More information on kickers in general is available in [Chapter 19, Kicker](#).

At a later point in time, the router VM becomes ready, and the `vm-up-and-running` leaf is set to a `true` value. The installed kicker notices the change and automatically calls the `reactive-re-deploy` action on the service instance. In turn, the service gets fully deployed.

```
admin@ncs# show vrouter vr-01 plan
```

TYPE	NAME	TRACK	GOAL	STATE	STATUS	WHEN	POST ACTION	
							ref	STATUS
self	self	false	-	init	reached	2021-09-16T14:26:30	-	-
				vm-requested	reached	2021-09-16T14:26:30	-	-
				vm-configured	reached	2021-09-16T14:28:40	-	-
				ready	reached	2021-09-16T14:28:40	-	-

The get-modifications output confirms this fact. It contains the additional IP address configuration, performed as part of the “vm-configured” step:

```
admin@ncs# vrouter vr-01 get-modifications
cli {
    local-node {
        data +vm-instance vr-01 {
            +      type      csr-small;
            +      address 198.51.100.1;
            +
        }
    }
}
```

The “ready” state has no additional pre-conditions, allowing NSO to reach it along with the “vm-configured” state. This effectively breaks the provisioning process into two steps. To break it down further, simply add more states with corresponding pre-conditions and create logic.

Other than staged provisioning, nano services act the same as other services, allowing you to use the service check-sync and similar actions, for example. But please note the un-deploy and re-deploy actions may behave differently than expected, as they deal with provisioning. Chiefly, a re-deploy reevaluates the pre-conditions, possibly generating a different configuration if a pre-condition depends on operational values that have changed. The un-deploy action, on the other hand, removes all of the recorded modifications, along with the generated plan.

Benefits and Use Cases

Every service in NSO has a YANG definition of the service parameters, a service point name, and an implementation of the service point `create()` callback. Normally, when a service is committed, the FASTMAP algorithm removes all previous data changes internally, and presents the service data to the `create()` callback as if this was the initial create. When the `create()` callback returns, the FASTMAP algorithm compares the result and calculates a reverse diff-set from the data changes. This

reverse diff-set contains the operations that are needed to restore the configuration data to the state as it was before the service was created. The reverse diff-set is required, for instance, if the service is deleted or modified.

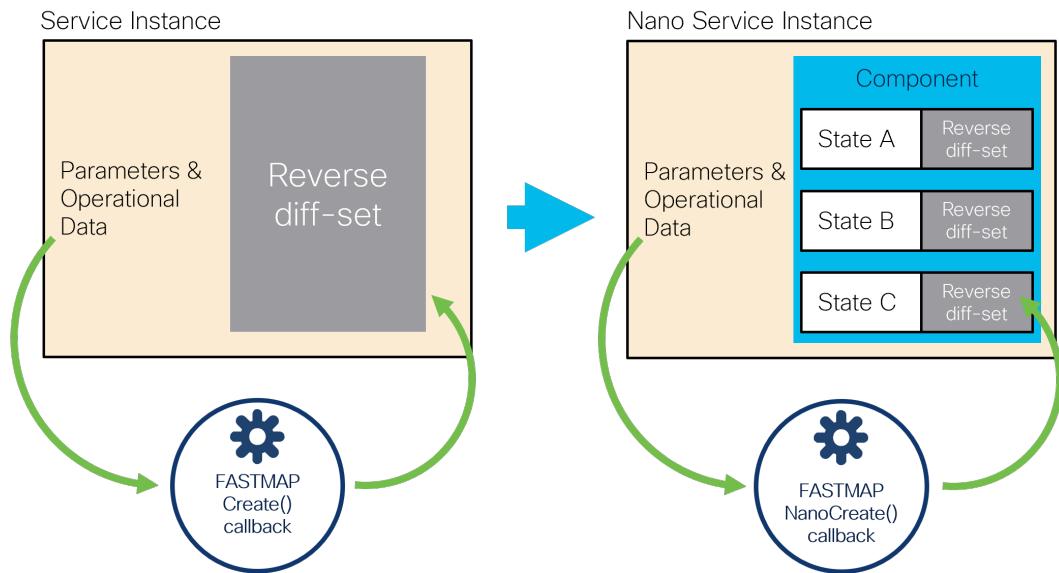
This fundamental principle is what makes the implementation of services and the `create()` callback simple. In turn, a lot of the NSO functionality relies on this mechanism.

However, in the reactive FASTMAP pattern the `create()` callback is re-entered several times by using the subsequent `reactive-re-deploy` calls. Storing all changes in a single reverse diff-set then becomes an impediment. For instance, if a staged delete is necessary, there is no way to single out which changes has each RFM step performed.

A nano service abandons the single reverse diff-set by introducing `nano-plan-data` and a new `NanoCreate()` callback. The `nano-plan-data` YANG grouping represents an executable plan that the system can follow to provision the service. It has additional storage for reverse diff-set and pre-conditions per state, for each component of the plan.

This is illustrated in the following figure:

Figure 203. Per-state FASTMAP with nano services



You can still use the service `get-modifications` action to visualize all data changes performed by the service as an aggregate. In addition, each state also has its own `get-modifications` action that visualizes the data-changes for that particular state. It allows you to more easily identify the state and, by extension, the code that produced those changes.

Before nano services became available, RFM services could only be implemented by creating a CDB subscriber. With the subscriber approach, the service can still leverage the `plan-data` grouping, which `nano-plan-data` is based on, to report the progress of the service under the resulting `plan` container. But the `create()` callback becomes responsible for creating the plan components, their states, and setting the status of the individual states as the service creation progresses.

Moreover, implementing a staged delete with a subscriber often requires keeping the configuration data outside of the service. The code is then distributed between the service `create()` callback and the

correlated CDB subscriber. This all results in several sources which potentially contain errors that are complicated to track down. Nano services, on the other hand, do not require any use of CDB subscribers or other mechanisms outside of the service code itself to support the full service life cycle.

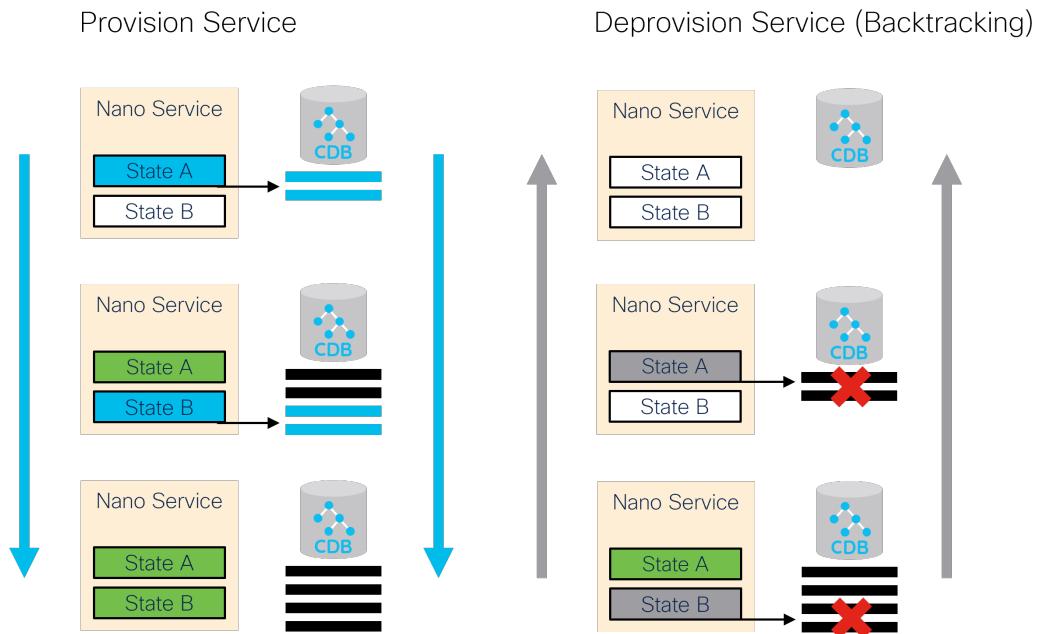
Backtracking and Staged Delete

Resource deprovisioning is an important part of the service life cycle. The FASTMAP algorithm ensures that no longer needed configuration changes in NSO are removed automatically but that may be insufficient by itself. For example, consider the case of a VM-based router, such as the one described earlier. Perhaps provisioning of the router also involves assigning a license from a central system to the VM and that license must be returned when the VM is decommissioned. If releasing the license must be done by the VM itself, simply destroying it will not work.

Another example is management of a web server VM for a web application. Here, each VM is part of a larger pool of servers behind a load balancer that routes client requests to these servers. During deprovisioning, simply stopping the VM interrupts the currently processing requests and results in client timeouts. This can be avoided with a graceful shutdown, which stops the load balancer from sending new connections to the server and waits for the current ones to finish, before removing the VM.

Both examples require two distinct steps for deprovisioning. Can nano services be of help in this case? Certainly. In addition to the state-by-state provisioning of the defined components, the nano service system in NSO is responsible for *back-tracking* during their removal. This process traverses all reached states in the reverse order, removing the changes previously done for each state one by one.

Figure 204. Staged delete with backtracking



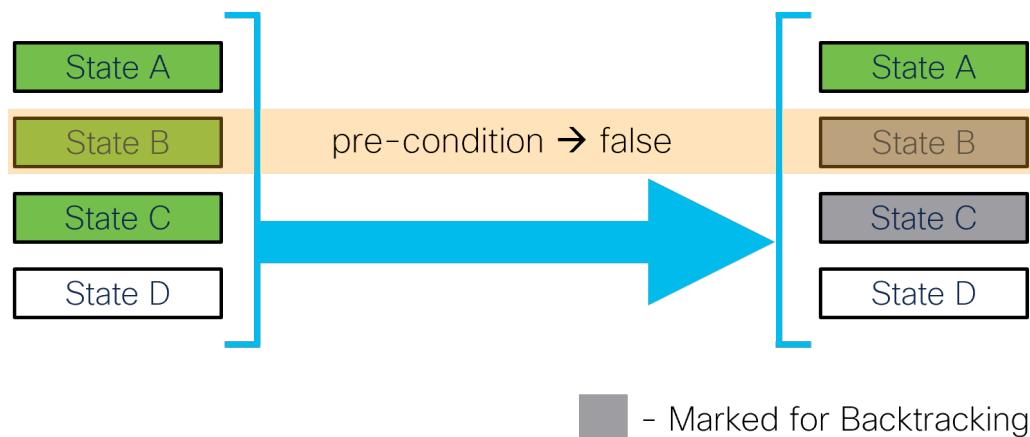
In doing so, the back-tracking process checks for a *delete pre-condition* of a state. A delete pre-condition is similar to the create pre-condition, but only relevant when back-tracking. If the condition is not fulfilled, the back-tracking process stops and waits until it becomes satisfied. Behind the scenes, a kicker is configured to restart the process when that happens.

If the state's delete pre-condition is fulfilled, back-tracking first removes the state's create changes recorded by FASTMAP and then invokes the `nano delete()` callback, if defined. The main use of the callback is to override or veto the default status calculation for a back-tracking state. That is why you can't implement the `delete()` callback with a template, for example. Very importantly, `delete()` changes are not kept in a service's reverse diff-set and may stay even after the service is completely removed. In general, you are advised to avoid writing any configuration data because this callback is called under a removal phase of a plan component where new configuration is seldom expected.

Since the “create” configuration is automatically removed, without the need for a separate `delete()` callback, these callbacks are used only in specific cases and are not very common. Regardless, the `delete()` callback may run as part of the **commit dry-run** command, so it must not invoke further actions or cause side effects.

Backtracking is invoked when a component of a nano service is removed, such as when deleting a service. It is also invoked when evaluating a plan and a reached state's create pre-condition is no longer satisfied. In this case, the affected component is temporarily set to back-tracking mode for as long as it contains such nonconforming states. It allows the service to recover and return to a well-defined state.

Figure 205. Backtracking on no longer satisfied pre-condition



To implement the delete pre-condition or the `delete()` callback, you must add the `ncs:delete` statement to the relevant state in the plan outline. Applying it to the web server example above, you might have:

```
ncs:state "vr:vm-requested" {
    ncs:create { ... }
    ncs:delete {
        ncs:pre-condition {
            ncs:monitor "$SERVICE"
            ncs:trigger-expr "requests-in-processing = '0'";
        }
    }
}
ncs:state "vr:vm-configured" {
    ncs:create { ... }
    ncs:delete {
        ncs:nano-callback;
    }
}
```

```
}
```

While, in general, the `delete()` callback should not produce any configuration, the graceful shutdown scenario is one of the few exceptional cases where this may be required. Here, the `delete()` callback allows you to re-configure the load balancer to remove the server from actively accepting new connections, such as marking it “under maintenance.” The delete pre-condition allows you to further delay the VM removal until the ongoing requests are completed.

Similar to the `create()` callback, the `ncs:nano-callback` statement instructs NSO to also process a `delete()` callback. A Python class that you have registered for the nano service must then implement the following method:

```
@NanoService.delete
def cb_nano_delete(self, tctx, root, service, plan, component, state,
                   proplist, component_proplist):
    ...
    ...
```

As explained, there are some uncommon cases where additional configuration with the `delete()` callback is required. However, a more frequent use of the `ncs:delete` statement is in combination with side-effect actions.

Managing Side Effects

In some scenarios, side effects are an integral part of the provisioning process and cannot be avoided. The aforementioned example on license management may require calling a specific device action. Even so, the `create()` or `delete()` callbacks, nano service or otherwise, are a bad fit for such work. Since these callbacks are invoked during the transaction commit, no RPCs or other access outside of the NSO datastore are allowed. If allowed, they would break the core NSO functionality, such as a dry-run, where side-effects are not expected.

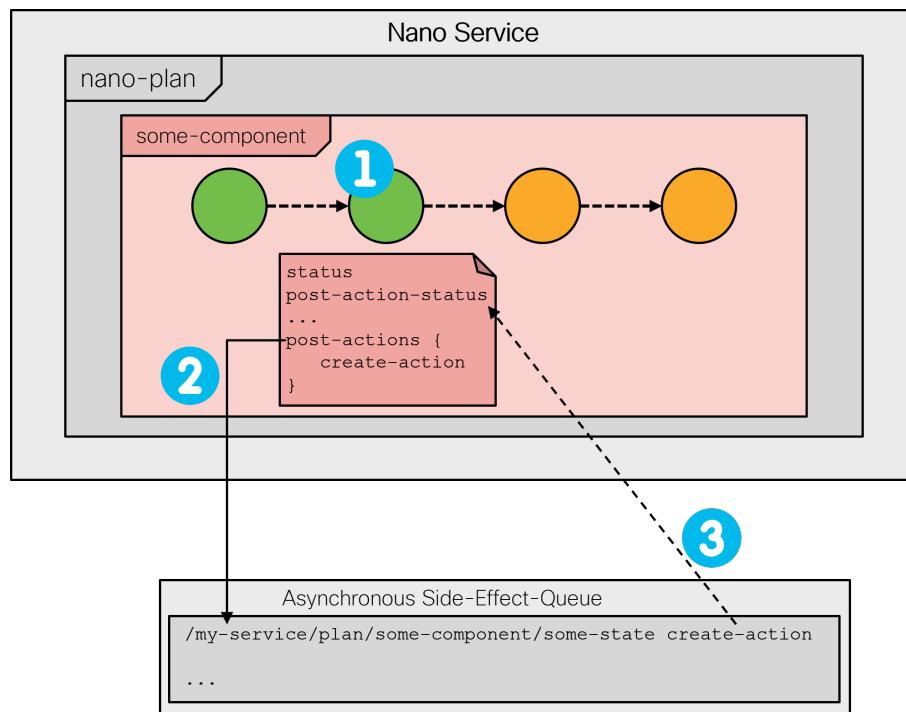
A common solution is to perform these actions outside of the configuration transaction. Nano services provide this functionality through the *post-actions* mechanism, using a `post-action-node` statement for a state. It is a definition of an action that should be invoked after the state has been reached and the commit performed. To ensure the latter, NSO will commit the current transaction before executing the post-action and advancing to the next state.

The service's plan state data also carries a *post-action-status* leaf, which reflects whether the action was executed and if it was successful. The leaf will be set to `not-reached`, `create-reached`, `delete-reached`, or `failed`, depending on the case and result. If the action is still executing, then the leaf will show either a `create-init` or `delete-init` status instead.

Moreover, post actions can be run either asynchronously (default) or synchronously. To run them synchronously, add a `sync` statement to the post-action statement. When a post action is run asynchronously, further states will not wait for the action to finish, unless you define an explicit `post-action-status` precondition. While for a synchronous post action, later states in the same component will be invoked only after the post action is run successfully.

The exception to this setting is when a component switches to a backtracking mode. In that case, the system will not wait for any create post action to complete (synchronous or not) but will start executing backtracking right away. It means a delete callback or a delete post action for a state may run before its synchronous create post action has finished executing.

The *side-effect-queue* and a corresponding kicker are responsible for invoking the actions on behalf of the nano service and reporting the result in the respective state's *post-action-status* leaf. The following figure shows an entry is made in the side-effect-queue (2) after the state is reached (1) and its *post-action-status* is updated (3) once the action finishes executing.

Figure 206. Post-action execution through side-effect-queue

You can use the **show side-effect-queue** command to inspect the queue. The queue will run multiple actions in parallel and keep the failed ones for you to inspect. Please note that High Availability (HA) setups require special consideration: the side effect queue is disabled when High Availability is enabled and the High Availability mode is NONE. See the section called “Mode of operation” in *NSO 5.7 Administration Guide* for more details.

In case of a failure, a post action sets the post-action-status accordingly and, if the action is synchronous, the nano service stops progressing. To retry the failed action, execute a (reactive) re-deploy, which will also restart the nano service if it was stopped.

Using the post-action mechanism, it is possible to define side effects for a nano service in a safe way. A post-action is only executed one time. That is, if the post-action-status is already at the `create-reached` in the create case or `delete-reached` in the delete case, then new calls of the post-actions are suppressed. In dry-run operations, post-actions are never called.

These properties make post actions useful in a number of scenarios. A widely applicable use case is invoking a service self-test as part of initial service provisioning.

Another example, requiring the use of post-actions, is the IP address allocation scenario from the chapter introduction. By its nature, the allocation or assignment call produces a side effect in an external system: it marks the assigned IP address in use. The same is true for releasing the address. Since NSO doesn't know how to reverse these effects on its own, they can't be part of any `create()` callback. Instead, the API calls can be implemented as post-actions.

The following snippet of a plan outline defines a create and delete post-action to handle IP management:

```
ncs:state "ncs:init" {
    ncs:create {
        ncs:post-action-node "$$SERVICE" {
            ncs:action-name "allocate-ip";
            ncs:sync;
```

```

        }
    }
}

ncs:state "vr:ip-allocated" {
    ncs:delete {
        ncs:post-action-node "$SERVICE" {
            ncs:action-name "release-ip";
        }
    }
}

```

Let's see how this plan manifests during provisioning. After the first (init) state is reached and committed, it fires off an allocation action on the service instance, called `allocate-ip`. The job of the `allocate-ip` action is to communicate with the external system, the IP Address Management (IPAM), and allocate an address for the service instance. This process may take a while, however it does not tie up NSO, since it runs outside of the configuration transaction and other configuration sessions can proceed in the meantime.

The `$SERVICE` XPath variable is automatically populated by the system and allows you to easily reference the service instance. There are other automatic variables defined. You can find the complete list inside the `tailf-ncs-plan.yang` submodule, in the `$NCS_DIR/src/ncs/yang/` folder.

Due to the `ncs:sync` statement, service provisioning can continue only after the allocation process (the action) completes. Once that happens, the service resumes processing in the ip-allocated state, with the IP value now available for configuration.

On service deprovisioning, the back-tracking mechanism works backwards through the states. When it is the ip-allocated state's turn to deprovision, NSO reverts any configuration done as part of this state, and then runs the `release-ip` action, defined inside the `ncs:delete` block. Of course, this only happens if the state previously had a reached status. Implemented as a post-action, `release-ip` can safely use the external IPAM API to deallocate the IP address, without impacting other sessions.

The actions, as defined in the example, do not take any parameters. When needed, you may pass additional parameters from the service's `opaque` and `component_proplist` object. These parameters must be set in advance, for example in some previous create callback. For details, please refer to the YANG definition of `post-action-input-params` in the `tailf-ncs-plan.yang` file.

Multiple and Dynamic Plan Components

The discussion on basic concepts briefly mentions the role of a nano behavior tree but it does not fully explore its potential. Let's now consider in which situations you may find a non-trivial behavior tree beneficial.

Suppose you are implementing a service that requires not one but two VMs. While you can always add more states to the main (self) component, these states are processed sequentially. However, you might want to provision the two VMs in parallel, since they take a comparatively long time, and it makes little sense having to wait until the first one is finished before starting with the second one. Nano services provide an elegant solution to this challenge in the form of multiple plan components: provisioning of each VM can be tracked by a separate plan component, allowing the two to advance independently, in parallel.

If the two VMs go through the same states, you can use a single component-type in the plan outline for both. It is the job of the behavior tree to create or *synthesize* actual components for each service instance. Therefore, you could use a behavior tree similar to the following example:

```

ncs:service-behavior-tree multirouter-servicepoint {
    description "A 2-VM behavior tree";
    ncs:plan-outline-ref "vr:multirouter-plan";
    ncs:selector {

```

```

ncs:create-component "'vm1'" {
    ncs:component-type-ref "vr:router-vm";
}
ncs:create-component "'vm2'" {
    ncs:component-type-ref "vr:router-vm";
}
}
}

```

The two ncs:create-component statements instruct NSO to create two components, named vm1 and vm2, of the same vr:router-vm type. Note the required use of single quotes around component names, because the value is evaluated as an XPath expression.

But this behavior tree has a flaw: It is missing the self component to tell the overall provisioning status. With multiple components in place, the self component should naturally reflect the cumulative status of the service. You can define the plan outline in the following way:

```

ncs:plan-outline multirouter-plan {
    description "Plan for configuring VM-based routers";
    ncs:self-as-service-status;

    ncs:component-type "ncs:self" {
        ncs:state "ncs:init";
        ncs:state "ncs:ready";
    }

    ncs:component-type "vr:router-vm" {
        ncs:state "ncs:init";
        // additional states
        ncs:state "ncs:ready";
    }
}

```

With the ncs:self-as-service-status statement present on a plan outline, the ready state of the self component will never have its status set to reached until all other components have the ready state status set to reached and all post actions have been run, too. Likewise, during backtracking, the init state will never be set to “not-reached” until all other components have been fully backtracked and all delete post actions have been run. Additionally, the self ready or init state status will be set to failed if any other state has a failed status or a failed post action, thus signaling that something has failed while executing the service instance.

To make use of the self component, you must also add it to the behavior tree:

```

ncs:service-behavior-tree multirouter-servicepoint {
    description "A 2-VM behavior tree";
    ncs:plan-outline-ref "vr:multirouter-plan";
    ncs:selector {
        ncs:create-component "'self'" {
            ncs:component-type-ref "ncs:self";
        }
        ncs:create-component "'vm1'" {
            ncs:component-type-ref "vr:router-vm";
        }
        ncs:create-component "'vm2'" {
            ncs:component-type-ref "vr:router-vm";
        }
    }
}

```

As you can see, all the ncs:create-component statements are placed inside an ncs:selector block. A selector is a so-called *control flow node*. It selects a group of components and allows you to

decide whether they are created or not, based on a pre-condition. The pre-condition can reference a service parameter, which in turn controls if the relevant components are provisioned for this service instance. The mechanism enables you to dynamically produce just the necessary plan components.

The pre-condition is not very useful on the top selector node, but selectors can also be nested. For example, having a `use-virtual-devices` configuration leaf in the service YANG model, you could modify the behavior tree to the following:

```
ncs:service-behavior-tree multirouter-servicepoint {
    description "A conditional 2-VM behavior tree";
    ncs:plan-outline-ref "vr:multirouter-plan";
    ncs:selector {
        ncs:create-component "'self'" { ... }
        ncs:selector {
            ncs:pre-condition {
                ncs:monitor "$SERVICE" {
                    ncs:trigger-expr "use-virtual-devices = 'true'";
                }
            }
            ncs:create-component "'vm1'" { ... }
            ncs:create-component "'vm2'" { ... }
        }
    }
}
```

The described behavior tree always synthesizes the self component and evaluates the child selector. However, the child selector only synthesizes the two VM components if the service configuration requested so by setting the `use-virtual-devices` to true.

What is more, if the pre-condition value changes, the system re-evaluates the behavior tree and starts the backtracking operation for any removed components.

For even more complex cases, where a variable number of components needs to be synthesized, the `ncs:multiplier` control flow node becomes useful. Its `ncs:foreach` statement selects a set of elements and each element is processed in the following way:

- If the optional `when` statement is not satisfied, the element is skipped.
- All `variable` statements are evaluated as XPath expressions for this element, to produce a unique name for the component and any other element-specific values.
- All `ncs:create-component` and other control flow nodes are processed, creating the necessary components for this element.

The multiplier node is often used to create a component for each item in a list. For example, if the service model contains a list of VMs, with a key name, then the following code creates a component for each of the items:

```
ncs:multiplier {
    ncs:foreach "vms" {
        ncs:variable "$NAME" {
            ncs:value-expr "concat('vm-', name)";
        }
        ncs:create-component "$NAME" { ... }
    }
}
```

In this particular case it might be possible to avoid the variable altogether, by using the expression for the `create-component` statement directly. However, defining a variable also makes it available to service `create()` callbacks.

This is extremely useful, since you can access these values, as well as the ones from the service opaque object, directly in the nano service XML templates. The opaque, especially, allows you to separate the logic in code from applying the XML templates.

Netsim Router Provisioning Example

The `examples.ncs/development-guide/nano-services/netsim-vrouter` folder contains a complete implementation of a service that provisions a netsim device instance, onboards it to NSO, and pushes a sample interface configuration to the device. Netsim device creation is neither instantaneous nor side-effect free, and thus requires the use of a nano service. It more closely resembles a real-world use-case for nano services.

To see how the service is used through a prearranged scenario, execute the `make demo` command from the example folder. The scenario provisions and deprovisions multiple netsim devices to show different states and behaviors, characteristic of nano services.

The service, called `vrouter`, defines three component types in the `src/yang/vrouter.yang` file:

- `vr:vrouter`: A “day-0” component that creates and initializes a netsim process as a virtual router device.
- `vr:vrouter-day1`: A “day-1” component for configuring the created device and tracking NETCONF notifications.
- `ncs:self`: The overall status indicator. It summarizes the status of all service components through the `self-as-service-status` mechanism.

As the name implies, the day-0 component must provision before the day-1 component. Since the two provision in sequence, in general, a single component would suffice. But the components are kept separate to illustrate component dependencies.

The behavior tree synthesizes each of the components for a service instance using some service-specific names. In order to do so, the example defines three variables to hold different names:

```
// vrouter name
ncs:variable "NAME" {
    ncs:value-expr "current()/name";
}
// vrouter component name
ncs:variable "D0NAME" {
    ncs:value-expr "concat(current()/name, '-day0')";
}
// vrouter day1 component name
ncs:variable "D1NAME" {
    ncs:value-expr "concat(current()/name, '-day1')";
}
```

The `vr:vrouter` (day-0) component has a number of plan states that it goes through during provisioning:

- `ncs:init`
- `vr:requested`
- `vr:onboarded`
- `ncs:ready`

The init and ready states are required as the first and last state in all components for correct overall state tracking in `ncs:self`. They have no additional logic tied to them.

The vr:requested state represents the first step in virtual router provisioning. While it does not perform any configuration itself (no nano-callback statement), it calls a post-action that does all the work. The following is a snippet of the plan outline for this state:

```
ncs:state "vr:requested" {
    ncs:create {
        // Call a Python action to create and start a netsim vrouter
        ncs:post-action-node "$SERVICE" {
            ncs:action-name "create-vrouter";
            ncs:result-expr "result = 'true'";
            ncs:sync;
        }
    }
}
```

The `create-vrouter` action calls the Python code inside the `python/vrouter/main.py` file, which runs a couple of system commands, such as the **ncs-netsim create-device** and the **ncs-netsim start** commands. These commands do the same thing as you would if you performed the task manually from the shell.

The vr:requested state also has a delete post-action, analogous to create, which stops and removes the netsim device during service deprovisioning or backtracking.

Inspecting the Python code for these post actions will reveal that a semaphore is used to control access to the common netsim resource. It is needed because multiple vrouting instances may run the create and delete action callbacks in parallel. The Python semaphore is shared between the delete and create action processes using a Python multiprocessing manager, as the example configures the NSO Python VM to start the actions in multiprocessing mode. See [the section called “The application component”](#) for details.

In vr:onboarded, the nano Python callback function from the `main.py` file adds the relevant NSO device entry for a newly-created netsim device. It also configures NSO to receive notifications from this device through a NETCONF subscription. When the NSO configuration is complete, the state transitions into the reached status, denoting the onboarding has completed successfully.

The `vr:vrouter` component handles so-called day-0 provisioning. Alongside this component, the `vr:vrouter-day1` component starts provisioning in parallel. During provisioning, it transitions through the following states:

- ncs:init
- vr:configured
- vr:deployed
- ncs:ready

The component reaches the init state right away. However, the vr:configured state has a precondition:

```
ncs:state "vr:configured" {
    ncs:create {
        // Wait for the onboarding to complete
        ncs:pre-condition {
            ncs:monitor "$SERVICE/plan/component[type='vr:vrouter']" +
                "[name=$D0NAME]/state[name='vr:onboarded']" {
                ncs:trigger-expr "post-action-status = 'create-reached'";
            }
        }
        // Invoke a service template to configure the vrouter
        ncs:nano-callback;
    }
}
```

```
}
```

Provisioning can continue only after the first component, `vr:vrouter`, has executed its `vr:onboarded` post-action. The precondition demonstrates how one component can depend on another component reaching some particular state or successfully executing a post-action.

The `vr:onboarded` post-action performs a **sync-from** command for the new device. After that happens, the `vr:configured` state can push the device configuration according to the service parameters, by using an XML template, `templates/vrouter-configured.xml`. The service simply configures an interface with a VLAN ID and a description.

Similarly, the `vr:deployed` state has its own precondition, which makes use of the `ncs:any` statement. It specifies either (any) of the two monitor statements will satisfy the precondition.

One of them checks the last received NETCONF notification contains a `link-status` value of `up` for the configured interface. In other words, it will wait for the interface to become operational.

However, relying solely on notifications in the precondition can be problematic, as the received notifications list in NSO can be cleared and would result in unintentional backtracking on a service re-deploy. For this reason, there is the other monitor statement, checking the device live-status.

Once either of the conditions is satisfied, it marks the end of provisioning. Perhaps the use of notifications in this case feels a little superficial but it illustrates a possible approach to waiting for the steady state, such as routing adjacencies to form and alike.

Altogether, the example shows how to use different nano service mechanisms in a single, complex, multistage service that combines configuration and side-effects. The example also includes a Python script that uses the RESTCONF protocol to configure a service instance and monitor its provisioning status. You are encouraged to configure a service instance yourself and explore the provisioning process in detail, including service removal. Regarding removal, have you noticed how nano services can deprovision in stages, but the service instance is gone from the configuration right away?

Zombie Services

By removing the service instance configuration from NSO, you start a service deprovisioning process. For an ordinary service, a stored reverse diff-set is applied, ensuring that all of the service-induced configuration is removed in the same transaction. For nano services, having a staged, multistep service delete operation, this is not possible. The provisioned states must be backtracked one by one, often across multiple transactions. With the service instance deleted, NSO must track the deprovisioning progress elsewhere.

For this reason, NSO mutates a nano service instance when it is removed. The instance is transformed into a *zombie* service, which represents the original service that still requires deprovisioning. Once the deprovisioning is complete, with all the states backtracked, the zombie is automatically removed.

Zombie service instances are stored with their service data, their plan states, and diff-sets in a `/ncs:zombies/services` list. When a service mutates to a zombie, all plan components are set to back-tracking mode and all service pre-condition kickers are rewritten to reference the zombie service instead. Also, the nano service subsystem now updates the zombie plan states as deprovisioning progresses. You can use the **show zombies service** command to inspect the plan.

Under normal conditions, you should not see any zombies, except for the service instances that are actively deprovisioning. However, if an error occurs, the deprovisioning process will stop with an error status and a zombie will remain. With a zombie present, NSO will not allow creating the same service instance in the configuration tree. The zombie must be removed first.

After addressing the underlying problem, you can restart the deprovisioning process with the `re-deploy` or the `reactive-re-deploy` actions. The difference between the two is which user the action uses. The `re-deploy` uses the current user that initiated the action whilst the `reactive-re-deploy` action keeps using the same user that last modified the zombie service.

These zombie actions behave a bit differently than their normal service counterparts. In particular, the zombie variants perform the following steps to better serve the deprovisioning process:

- 1 Start a temporary transaction in which the service is reinstated (created). The service plan will have the same status as it had when it mutated.
- 2 Back-track plan components in a normal fashion, that is, removing device changes for states with delete pre-conditions satisfied.
- 3 If all components are completely back-tracked, the zombie is removed from the zombie-list. Otherwise, the service and the current plan states are stored back into the zombie-list, with new kickers waiting to activate the zombie when some delete pre-condition is satisfied.

In addition, zombie services support the `resurrect` action. The action reinstates the zombie back in the configuration tree as a real service, with the current plan status, and reverts plan components back from back-tracking to normal mode. It is an “undo” for a nano service delete.

In some situations, especially during nano service development, a zombie may get stuck because of a misconfigured precondition or similar issues. A `re-deploy` is unlikely to help in that case and you may need to forcefully remove the problematic plan component. The `force-back-track` action performs this job and allows you to backtrack to a specific state, if specified. But beware that using the action avoids calling any post-actions or delete callbacks for the forcefully backtracked states, even though the recorded configuration modifications are reverted. It *can and will* leave your systems in an inconsistent or broken state if you are not careful. For the same reason, you may want to avoid using the `un-deploy` action with nano services, as it behaves the same way.

Using Notifications to Track the Plan and its Status

When a service is provisioned in stages, as nano services are, the success of the initial commit no longer indicates the service is provisioned. Provisioning may take a while and may fail later, requiring you to consult the service plan to observe the service status. This makes it harder to tell when a service finishes provisioning, for example. Fortunately, services provide a set of notifications that indicate important events in the service's life-cycle, including a successful completion. These events enable NETCONF and RESTCONF clients to subscribe to events instead of polling the plan and commit queue status.

The built-in `service-state-changes` NETCONF/RESTCONF stream is used by NSO to generate northbound notifications for services, including nano services. The event stream is enabled by default in `ncs.conf`, however, individual notification events must be explicitly configured to be sent.

The `plan-state-change` Notification

When a service's plan component changes state, the `plan-state-change` notification is generated with the new state of the plan. It includes the `status`, which indicates one of `not-reached`, `reached`, or `failed`. The notification is sent when the state is `created`, `modified`, or `deleted`, depending on the configuration. For reference on the structure and all the fields present in the notification, please see the YANG model in the `tailf-ncs-plan.yang` file.

As a common use-case, suppose the `self-as-service-status` statement has been set on the plan outline. An event with status `reached` for the `self` component `ready` state signifies that all nano service components have reached their ready state and provisioning is complete. A simple example of this

The service-commit-queue-event Notification

scenario is included in the examples.ncs/development-guide/nano-services/netsim-vrouter/demo.py Python script, using RESTCONF.

To enable the plan-state-change notifications to be sent, you must enable them for a specific service in NSO. For example, can load the following configuration into the CDB as an XML initialization file:

```
<services xmlns="http://tail-f.com/ns/ncs">
  <plan-notifications>
    <subscription>
      <name>nano1</name>
      <service-type>/vr:vrouter</service-type>
      <component-type>self</component-type>
      <state>ready</state>
      <operation>modified</operation>
    </subscription>
    <subscription>
      <name>nano2</name>
      <service-type>/vr:vrouter</service-type>
      <component-type>self</component-type>
      <state>ready</state>
      <operation>created</operation>
    </subscription>
  </plan-notifications>
</services>
```

This configuration enables notifications for the self component's ready state when created or modified.

The service-commit-queue-event Notification

When a service is committed through the commit queue, this notification acts as a reference regarding the state of the service. Notifications are sent when the service commit queue item is waiting to run, executing, waiting to be unlocked, completed, failed, or deleted. More details on the service-commit-queue-event notification content can be found in the YANG model inside tailf-ncs-services.yang .

For example, the failed event can be used to detect that a nano service instance deployment failed because a configuration change committed through the commit queue has failed. Measures to resolve the issue can then be taken and the nano service instance can be re-deployed. A simple example of this scenario is included in the examples.ncs/development-guide/nano-services/netsim-vrouter/demo.py Python script where the service is committed through the commit queue, using RESTCONF. By design, the configuration commit to a device fails, resulting in a commit-queue-notification with the failed event status for the commit queue item.

To enable the service-commit-queue-event notifications to be sent, you can load the following example configuration into NSO, as an XML initialization file or some other way:

```
<services xmlns="http://tail-f.com/ns/ncs">
  <commit-queue-notifications>
    <subscription>
      <name>nano1</name>
      <service-type>/vr:vrouter</service-type>
    </subscription>
  </commit-queue-notifications>
</services>
```

Examples of service-state-changes Stream Subscriptions

The following examples demonstrate the usage and sample events for the notification functionality, described in this section, using RESTCONF, NETCONF, and CLI northbound interfaces.

RESTCONF subscription request using curl:

```
$ curl -isu admin:admin -X GET -H "Accept: text/event-stream"
      http://localhost:8080/restconf/streams/service-state-changes/json

data: {
data:   "ietf-restconf:notification": {
data:     "eventTime": "2021-11-16T20:36:06.324322+00:00",
data:     "tailf-ncs:service-commit-queue-event": {
data:       "service": "/vrouter:vrouter[name='vr7']",
data:       "id": 1637135519125,
data:       "status": "completed",
data:       "trace-id": "vr7-1"
data:     }
data:   }
data: }

data: {
data:   "ietf-restconf:notification": {
data:     "eventTime": "2021-11-16T20:36:06.728911+00:00",
data:     "tailf-ncs:plan-state-change": {
data:       "service": "/vrouter:vrouter[name='vr7']",
data:       "component": "self",
data:       "state": "tailf-ncs:ready",
data:       "operation": "modified",
data:       "status": "reached",
data:       "trace-id": "vr7-1"
data:     }
data:   }
data: }
```

See the section called “Streams” in *NSO 5.7 Northbound APIs* for further reference.

NETCONF create subscription using **netconf-console**:

```
$ netconf-console create-subscription=service-state-changes

<?xml version="1.0" encoding="UTF-8"?>
<notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
  <eventTime>2021-11-16T20:36:06.324322+00:00</eventTime>
  <service-commit-queue-event xmlns="http://tail-f.com/ns/ncs">
    <service xmlns:vr="http://com/example/vrouter">/vr:vrouter[vr:name='vr7']</service>
    <id>1637135519125</id>
    <status>completed</status>
    <trace-id>vr7-1</trace-id>
  </service-commit-queue-event>
</notification>
<?xml version="1.0" encoding="UTF-8"?>
<notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
  <eventTime>2021-11-16T20:36:06.728911+00:00</eventTime>
  <plan-state-change xmlns="http://tail-f.com/ns/ncs">
    <service xmlns:vr="http://com/example/vrouter">/vr:vrouter[vr:name='vr7']</service>
    <component>self</component>
    <state>ready</state>
    <operation>modified</operation>
    <status>reached</status>
    <trace-id>vr7-1</trace-id>
  </plan-state-change>
</notification>
```

See the section called “Notification Capability” in *NSO 5.7 Northbound APIs* for further reference.

CLI show received notifications using **ncs_cli**:

```
$ ncs_cli -u admin -C <<<'show notification stream service-state-changes'
```

The trace-id in the Notification

```

notification
eventTime 2021-11-16T20:36:06.324322+00:00
service-commit-queue-event
  service /vrouter[name='vr17']
  id 1637135519125
  status completed
  trace-id vr7-1
!
!
notification
eventTime 2021-11-16T20:36:06.728911+00:00
plan-state-change
  service /vrouter[name='vr7']
  component self
  state ready
  operation modified
  status reached
  trace-id vr7-1
!
!
```

The trace-id in the Notification

You have likely noticed the trace-id field at the end of the example notifications above. The trace id is an optional but very useful parameter when committing the service configuration. It helps you trace the commit in the emitted log messages and the service-state-changes stream notifications. The above notifications, taken from the examples.ncs/development-guide/nano-services/netsim-vrouter example, are emitted after applying a RESTCONF plain patch:

```
$ curl -isu admin:admin -X PATCH
-H "Content-type: application/yang-data+json"
'http://localhost:8080/restconf/data?commit-queue=sync&trace-id=vr7-1'
-d '{ "vrouter:vrouter": [ { "name": "vr7" } ] }'
```

Note that the trace id is specified as part of the URL. If missing, NSO will generate and assign one on its own.

Developing and Updating a Nano Service

At times, especially when you use an iterative development approach or simply due to changing requirements, you might need to update (change) an existing nano service and its implementation. In addition to other service update best practices, such as model upgrades, you must carefully consider the nano-service-specific aspects. The following discussion mostly focuses on migrating an already provisioned service instance to a newer version; however, the same concepts also apply while you are initially developing the service.

In the simple case, updating the model of a nano service and getting the changes to show up in an already created instance is a matter of executing a normal re-deploy. This will synthesize any new components and provision them, along with the new configuration, just like you would expect from a non-nano service.

A major difference occurs if a service instance is deleted and is in a zombie state when the nano service is updated. You should be aware that no synthetization is done for that service instance. The only goal of a deleted service is to revert any changes made by the service instance. Therefore, in that case, the synthetization is not needed. It means that, if you've made changes to callbacks, post actions, or pre-conditions, those changes will not be applied to zombies of the nano service. If a service instance requires the new changes to be applied, you must re-deploy it before it is deleted.

When updating nano services, you also need to be aware that any old callbacks, post actions and any other models that the service depends on, need to be available in the new nano service package until all service instances created before the update have either been updated (through a re-deploy) or fully deleted. Therefore, you must take great care with any updates to a service if there are still zombies left in the system.

Adding Components

Adding new components to the behavior tree will create the new components during the next re-deploy (synthetization) and execute the states in the new components as is normally done.

Removing Components

When removing components from the behavior tree, the components that are removed are set to backtracking and are backtracked fully before they are removed from the plan.

When you remove a component, do so carefully so that any callbacks, post actions or any other model data that the component depends on are not removed until all instances of the old component are removed.

If the identity for a component type is removed, then NSO removes the component from the database when upgrading the package. If this happens, the component is not backtracked and the reverse diffsets are not applied.

Replacing Components

Replacing components in the behavior tree is the same as having unrelated components that are deleted and added in the same update. The deleted components are backtracked as far as possible, and then the added components are created and their states executed in order.

In some cases, this is not the desired behaviour when replacing a component. For example, if you only want to rename a component, backtracking and then adding the component again might make NSO push unnecessary changes to the network or run delete callbacks and post actions that should not be run. To remedy this, you might add the `ncs:deprecates-component` statements to the new component, detailing which components it replaces. NSO then skips the backtracking of the old component and just applies all reverse diffsets of the deprecated component. In the same re-deploy, it then executes the new component as usual. Therefore, if the new component produces the same configuration as the old component, nothing is pushed to the network.

If any of the deprecated components are backtracking, the backtracking will be handled before the component is removed. When there are multiple components that are deprecated in the same update, the components will not be removed, as detailed above, until all of them are done backtracking (if any one of them are backtracking).

Adding and Removing States

When adding or removing states in a component, the component is backtracked before a new component with the new states is added and executed. If the updated component produces the same configuration as the old one (and no preconditions halt the execution), this should lead to no configuration being pushed to the network. So, if changes to the states are done, you need to take care when writing the preconditions and post actions for a component if no new changes should be pushed to the network.

Any changes to the already present states that are kept in the updated component will not have their configuration updated until the new component is created, which happens after the old one has been fully backtracked.

Modifying States

For a component where only the configuration for one or more states have changed, the synthetization process will update the component with the new configuration and make sure that any new callbacks or similar are called during future execution of the component.

Implementation Reference

The text in this section sums up as well as adds additional detail on the way nano services operate, which you will hopefully find beneficial during implementation.

To reiterate, the purpose of a nano service is to break down an RFM service into its isolated steps. It extends the normal `ncs:servicepoint` YANG mechanism and requires the following:

- A YANG definition of the service input parameters, with a service point name and the additional *nano-plan-data* grouping.
- A YANG definition of the plan component types and their states in a *plan outline*.
- A YANG definition of a *behavior tree* for the service. The behavior tree defines how and when to instantiate components in the plan.
- Code or templates for individual state transfers in the plan.

When a nano service is committed, the system evaluates its behavior tree. The result of this evaluation is a set of components that form the current plan for the service. This set of components is compared with the previous plan (before the commit). If there are new components, they are processed one by one.

For each component in the plan, it is executed state by state in the defined order. Before entering a new state, the *create pre-condition* for the state is evaluated if it exists. If a create pre-condition exists and if it is not satisfied, the system stops progressing this component and jumps to the next one. A kicker is then defined for the pre-condition that was not satisfied. Later, when this kicker triggers and the pre-condition is satisfied, it performs a *reactive-re-deploy* and the kicker is removed. This kicker mechanism becomes a self-sustained RFM loop.

If a state's pre-conditions are met, the callback function or template associated with the state is invoked, if it exists. If the callback is successful, the state is marked as *reached*, and the next state is executed.

A component, that is no longer present but was in the previous plan, goes into *back-tracking mode*, during which the goal is to remove all reached states and eventually remove the component from the plan. Removing state data changes is performed in a strict reverse order, beginning with the last reached state and taking into account a *delete pre-condition* if defined.

A nano service is expected to have a `ncs:self` component type, all other component types are optional. Any component type, including `ncs:self`, are expected to have `ncs:init` as its first state and `ncs:ready` as its last state. Any component type, including `ncs:self`, can have any number of specific states in between `ncs:init` and `ncs:ready`.

Back-Tracking

Back-tracking is completely automatic and occurs in the following scenarios:

State pre-condition not satisfied

A reached state's pre-condition is no longer satisfied, and there are subsequent states that are reached and contain reverse diff-sets.

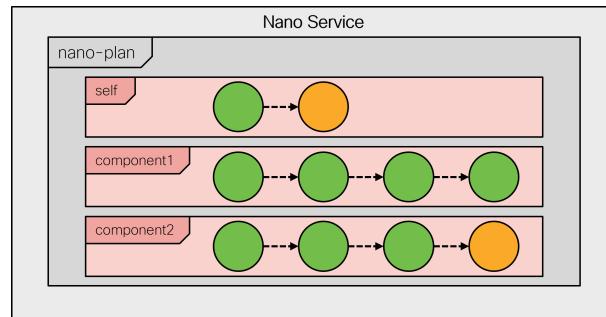
Plan component is removed

When a plan component is removed and has reached states that contain reverse diff-sets.

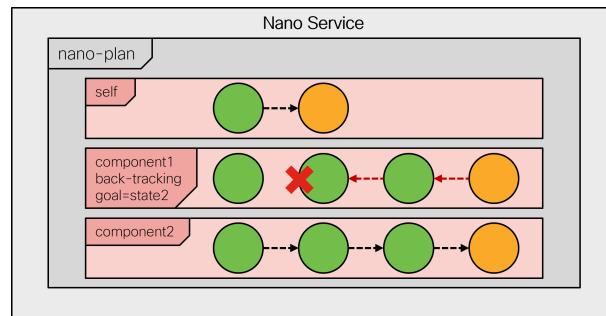
Service is deleted

When a service is deleted, NSO will set all plan components to back-tracking mode before deleting the service.

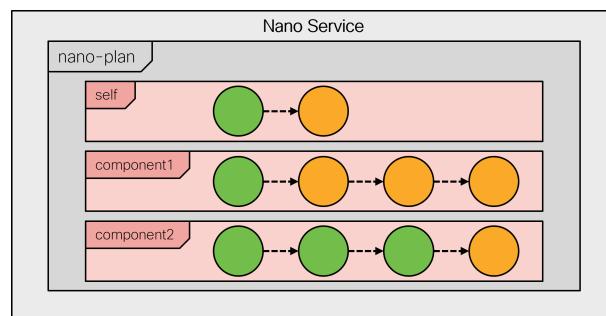
For each RFM loop, NSO traverses each component and state in order. For each non-satisfied create pre-condition, a kicker is started that monitors and triggers when the pre-condition becomes satisfied.



While traversing the states, a create pre-condition that was previously satisfied may become un-satisfied. If there are subsequent reached states that contain reverse diff-sets, then the component must be set to back-tracking mode. The back-tracking mode has as its goal to revert all changes up to the state which originally failed to satisfy its create pre-condition. While back-tracking, the delete pre-condition for each state is evaluated, if it exists. If the delete pre-condition is satisfied, the state's reverse diff-set is applied, and the next state is considered. If the delete pre-condition is not satisfied, a kicker is created to monitor this delete pre-condition. When the kicker triggers, a `reactive-re-deploy` is called and the back-tracking will continue until the goal is reached.

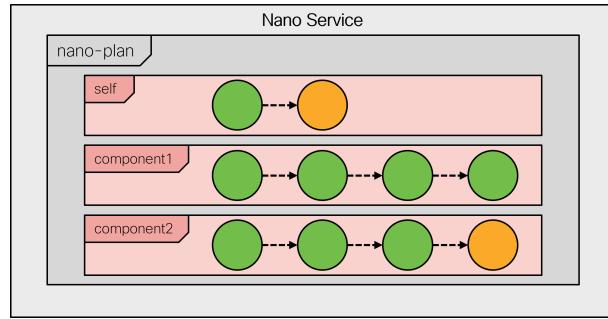


When the back-tracking plan component has reached its goal state, the component is set to normal mode again. The state's create pre-condition is evaluated and if it is satisfied the state is entered or otherwise a kicker is created as described above.

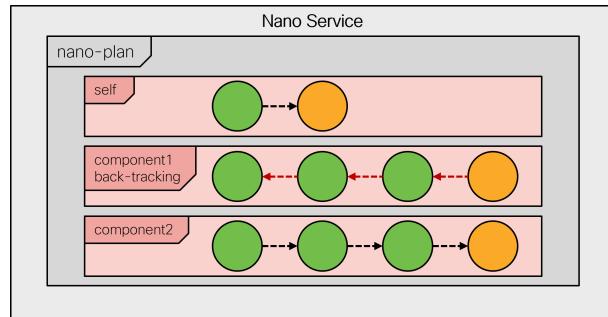


In some circumstances a complete plan component is removed (for example, if the service input parameters are changed). If this happens, the plan component is checked if it contains reached states that contain reverse diff-sets.

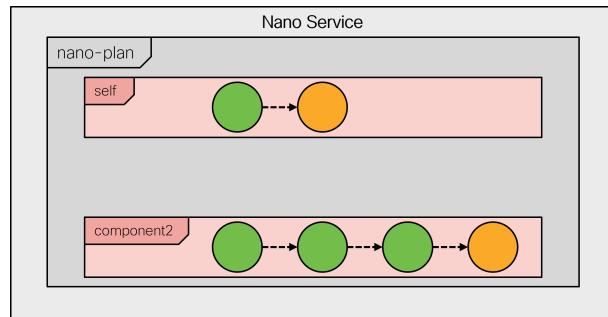
Back-Tracking



If the removed component contains reached states with reverse diff-sets, the deletion of the component is deferred and the component is set to back-tracking mode.

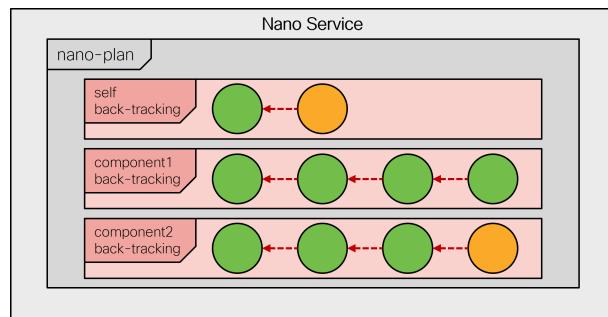


In this case, there is no specified goal state for the back-tracking. This means that when all the states have been reverted, the component is automatically deleted.

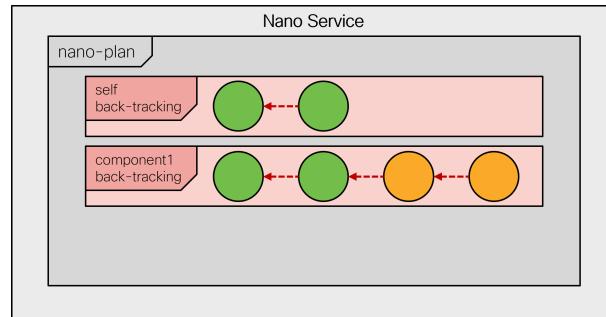


If a service is deleted, all components are set to back-tracking mode. The service becomes a zombie, storing away its plan states so that the service configuration can be removed.

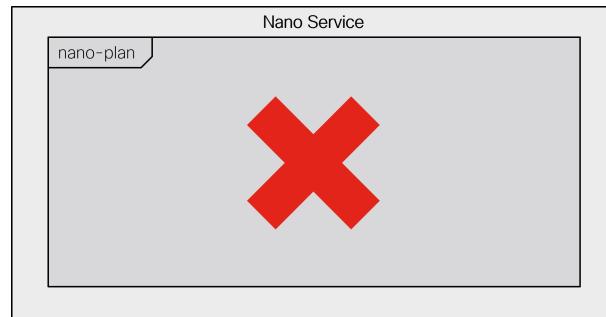
All components of a deleted service are set in backtracking mode.



When a component becomes completely back-tracked, it is removed.



When all components in the plan are deleted, the service is removed.



Behavior Tree

A nano service behavior tree is a data structure defined for each service type. Without a behavior tree defined for the service point, the nano service cannot execute. It is the behavior tree that defines the currently executing nano-plan with its components.



Note

This is in stark contrast to plan-data used for logging purposes where the programmer needs to write the plan and its components in the `create()` callback. For nano services, it is not allowed to define the nano plan in any other way than by a behavior tree.

The purpose of a behavior tree is to have a declarative way to specify how the service's input parameters are mapped to a set of component instances.

A behavior tree is a directed tree in which the nodes are classified as control flow nodes and execution nodes. For each pair of connected nodes, the outgoing node is called parent and the incoming node is called child. A control flow node has zero or one parent and at least one child, and the execution nodes have one parent and no children.

There is exactly one special control flow node called the *root*, which is the only control flow node without a parent.

This definition implies that all interior nodes are control flow nodes, and all leaves are execution nodes. When creating, modifying, or deleting a nano service, NSO evaluates the behavior tree to render the current nano plan for the service. This process is called *synthesizing* the plan.

The control flow nodes have a different behavior, but in the end, they all synthesize its children in zero or more instances. When a control flow node is synthesized, the system executes its rules for synthesizing

the node's children. Synthesizing an execution node adds the corresponding plan component instance to the nano service's plan.

All control flow and execution nodes may define pre-conditions, which must be satisfied to synthesize the node. If a pre-condition is not satisfied, a kicker is started to monitor the pre-condition.

All control flow and execution nodes may define an *observe monitor* which results in a kicker being started for the monitor when the node is synthesized.

If an invocation of an RFM loop (for example, a re-deploy) synthesizes the behavior tree and a pre-condition for a child is no longer satisfied, the sub-tree with its plan-components is removed (that is, the plan-components are set to back-tracking mode).

The following control flow nodes are defined:

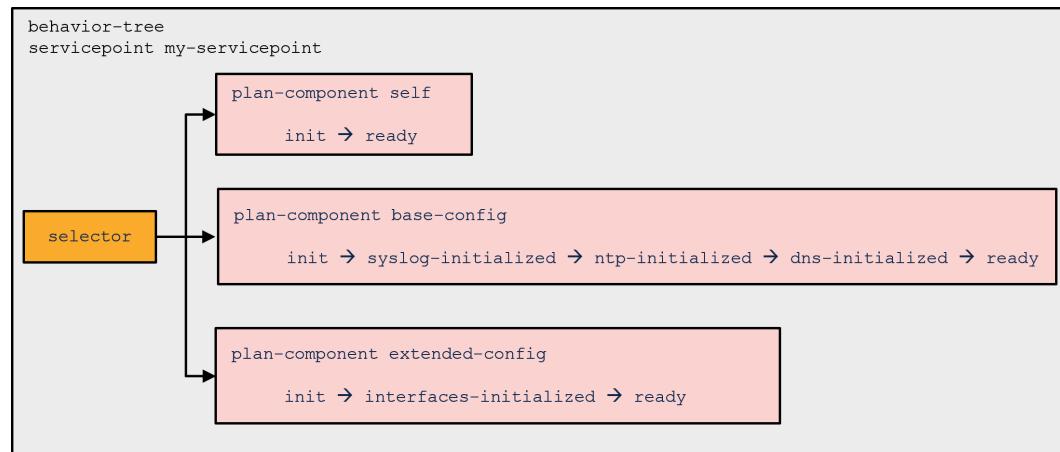
Selector A *selector* node has a set of children which are synthesized as described above.

Multiplier A *multiplier* has a *foreach* mechanism that produces a list of elements. For each resulting element, the children are synthesized as described above. This can be used, for example, to create several plan-components of the same type.

There is just one type of execution node:

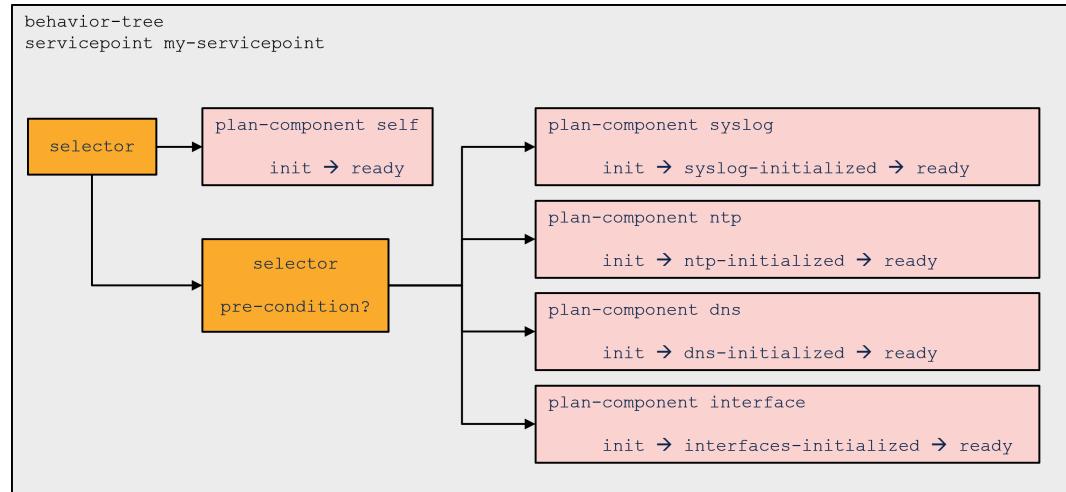
Create component The *create-component* execution node creates an instance of the component type that it refers to in the plan.

It is recommended to keep the behavior tree as flat as possible. The most trivial case is when the behavior tree creates a static nano-plan, that is, all the plan-components are defined and never removed. The following is an example of such a behavior tree:



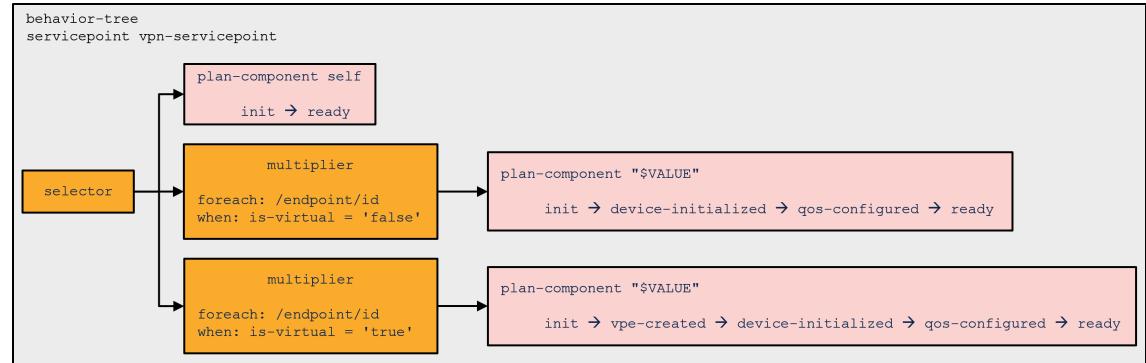
Having a selector on root implies that all plan-components are created if they don't have any pre-conditions, or for which the pre-conditions are satisfied.

An example of a more elaborated behavior tree is the following:



This behavior tree has a selector node as the root. It will always synthesize the "self" plan component and then evaluate then pre-condition for the selector child. If that pre-condition is satisfied, it then creates four other plan-components.

The multiplier control flow node is used when a plan component of a certain type should be cloned into several copies depending on some service input parameters. For this reason, the multiplier node defines a `foreach`, a `when`, and a `variable`. The `foreach` is evaluated and for each node in the nodeset that satisfies the `when`, the `variable` is evaluated as the outcome. The value is used for parameter substitution to a unique name for a duplicated plan component.



The value is also added to the nano service opaque which enables the individual state nano service `create()` callbacks to retrieve the value.

Variables might also have “when” expressions, which are used to decide if the variable should be added to the list of variables or not.

Nano Service Pre-Condition

Pre-conditions are what drives the execution of a nano service. A pre-condition is a prerequisite for a state to be executed or a component to be synthesized. If the pre-condition is not satisfied, it is then turned into a kicker which in turn re-deploys the nano service once the condition is fulfilled.

When working with pre-conditions, you need to be aware that they work a bit differently when used as a kicker to re-deploy the service and when they are used in the execution of the service. When the pre-condition is used in the re-deploy kicker, it then works as explained in the kicker documentation (that is, the trigger expression is evaluated before and after the change-set of the commit when the monitored nodeset is changed). When used during the execution of a nano service, you can only evaluate it on the

current state of the database, which means that it only checks that the monitor returns a nodeset of one or more nodes and that trigger expression (if there is one) is fulfilled for any of the nodes in the nodeset.

Support for pre-conditions checking, if a node has been deleted, is handled a bit differently due to the difference in how the pre-condition is evaluated. Kickers always trigger for changed nodes (add, deleted, or modified) and can check that the node was deleted in the commit that triggered the kicker. While in the nano service evaluation, you only have the current state of the database and the monitor expression will not return any nodes for evaluation of the trigger expression, consequently evaluating the pre-condition to false. To support deletes in both cases, you can create a pre-condition with a monitor expression and a child node `ncs:trigger-on-delete` which then both create a kicker that checks for deletion of the monitored node and also does the right thing in the nano service evaluation of the pre-condition. For example, you could have the following component:

```
ncs:component "self" {
    ncs:state "init" {
        ncs:delete {
            ncs:pre-condition {
                ncs:monitor "/devices/device[name='test']" {
                    ncs:trigger-on-delete;
                }
            }
        }
        ncs:state "ready";
    }
}
```

The component would only trigger the init states delete pre-condition when the device named test is deleted.

It is possible to add multiple monitors to a pre-condition by using the `ncs:all` or `ncs:any` extensions. Both extensions take one or multiple monitors as argument. A pre-condition using the `ncs:all` extension is satisfied if all monitors given as arguments evaluate to true. A pre-condition using the `ncs:any` extension is satisfied if at least one of the monitors given as argument evaluates to true. The following component uses the `ncs:all` and `ncs:any` extensions for its `self` state's create and delete pre-condition, respectively:

```
ncs:component "self" {
    ncs:state "init" {
        ncs:create {
            ncs:pre-condition {
                ncs:all {
                    ncs:monitor $SERVICE/syslog {
                        ncs:trigger-expr: "current() = true"
                    }
                    ncs:monitor $SERVICE/dns {
                        ncs:trigger-expr: "current() = true"
                    }
                }
            }
        }
        ncs:delete {
            ncs:pre-condition {
                ncs:any {
                    ncs:monitor $SERVICE/syslog {
                        ncs:trigger-expr: "current() = false"
                    }
                    ncs:monitor $SERVICE/dns {
                        ncs:trigger-expr: "current() = false"
                    }
                }
            }
        }
    }
}
```

```
        }
    }
}
}

ncs:state "ready";
```

Nano Service Opaque and Component Properties

The service opaque is a name-value list that can optionally be created/modified in some of the service callbacks, and then travels the chain of callbacks (pre-modification, create, post-modification). It is returned by the callbacks and stored persistently in the service private data. Hence, the next service invocation has access to the current opaque and can make subsequent read/write operations to the same object. The object is usually called opaque in Java and proplist in Python callbacks.

The nano services handle the opaque in a similar fashion, where a callback for every state has access to and can modify the opaque. However, the behavior tree can also define variables, which you can use in preconditions or to set component names. These variables are also available in the callbacks, as component properties. The mechanism is similar but separate from the opaque. While the opaque is a single service-instance-wide object set only from the service code, component variables are set in and scoped according to the behavior tree. That is, component properties contain only the behavior tree variables which are in scope when a component is synthesized.

For example, take the following behavior tree snippet:

```

ncs:selector {
    ncs:variable "VAR1" {
        ncs:value-expr "'value1'";
    }
    ncs:create-component "'self'" {
        ncs:component-type-ref "ncs:self";
    }
}
ncs:selector {
    ncs:variable "VAR2" {
        ncs:value-expr "'value2'";
    }
    ncs:create-component "'component1'" {
        ncs:component-type-ref "t:my-component";
    }
}

```

The callbacks for states in the “self” component only see the VAR1 variable, while those in “component1” see both VAR1 and VAR2 as component properties.

Additionally, both the service opaque and component variables (properties) are used to look up substitutions in nano service XML templates and in the behavior tree. If used in the behavior tree, the same rules apply for the opaque as for component variables. So, a value needs to contain single quotes if you wish to use it verbatim in preconditions and similar constructs, for example:

```
proplist.append( ('VARX', "'some value'") )
```

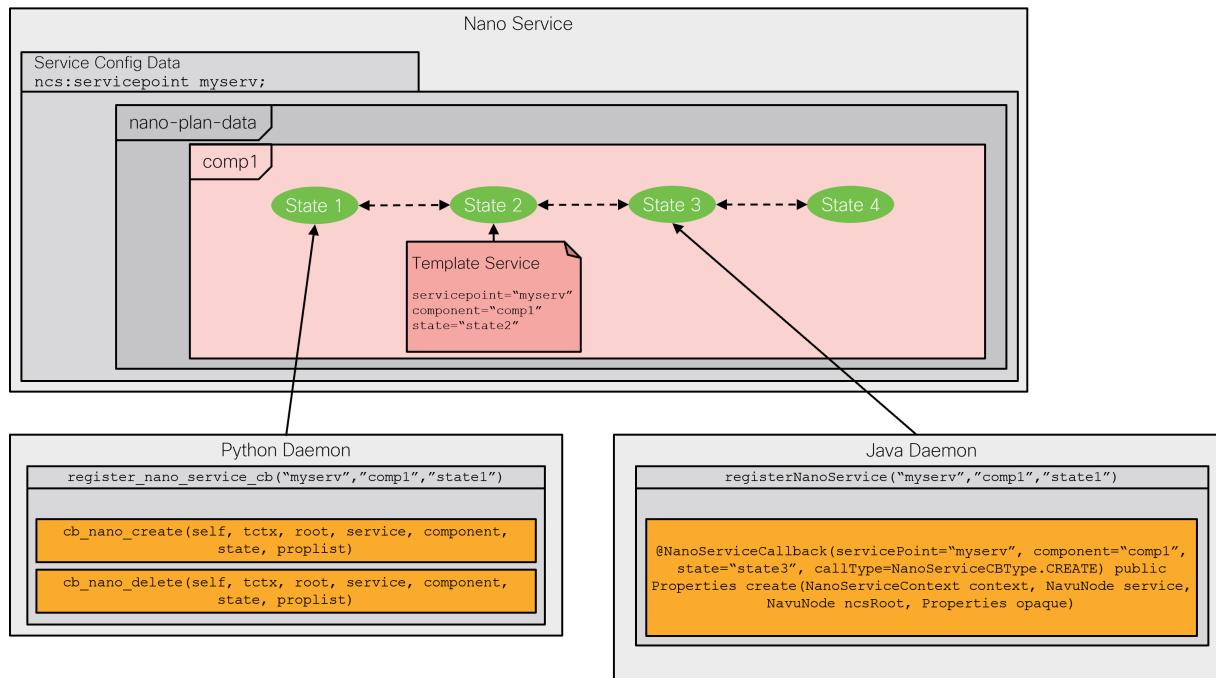
Using this scheme at an early state, such as the “self” component’s “ncs:init”, you can have a callback that sets name-value pairs for all other states that are then implemented solely with templates and preconditions.

Nano Service Callbacks

The nano service can have several callback registrations, one for each plan component state. But note that some states may have no callbacks at all. The state may simply act as a checkpoint, that some condition is satisfied, using pre-condition statements. A component's `ncs:ready` state is a good example of this.

The drawback with this flexible callback registration is that there must be a way for the NSO Service Manager to know if all expected nano service callbacks have been registered. For this reason, all nano service plan component states that require callbacks are marked with this information. When the plan is executed and the callback markings in the plan mismatch with the actual registrations, this results in an error.

All callback registrations in NSO require a *daemon* to be instantiated, such as a Python or Java process. For nano services, it is allowed to have many *daemons* where each *daemon* is responsible for a subset of the plan state callback registrations. The neat thing here is that it becomes possible to mix different callback types (Template/Python/Java) for different plan states.



The mixed callback feature caters to the case where most of the callbacks are templates and only some are Java or Python. This works well because nano services try to resolve the template parameters using the nano service opaque when applying a template. This is a unique functionality for nano services that makes Java or Python apply-template callbacks unnecessary.

You can implement nano service callbacks as Templates as well as Python, Java, Erlang, and C code. The following examples cover the implementation of Template, Python and Java.

A plan state template, if defined, replaces the need of a `create()` callback. In this case, there are no `delete()` callbacks and the status definitions must in this case be handled by the states delete pre-condition. The template must in addition to the `servicepoint` attribute have a *component type* and a *state* attribute to be registered on the plan state:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0"
    servicepoint="my-servicepoint"
```

```

        componenttype="my:some-component"
        state="my:some-state"
    <devices xmlns="http://tail-f.com/ns/ncs">
        <!-- ... -->
    </devices>
</config-template>

```

Specific to nano services, you can use parameters, such as \$SOMEPARAM in the template. The system searches for the parameter value in the service opaque and in the component properties. If it is not defined, applying the template will fail.

A Python `create()` callback is very similar to its ordinary service counterpart. The difference is that it has additional arguments. `plan` refers to the synthesized plan, while `component` and `state` specify the component and state for which it is invoked. The `proplist` argument is the nano service opaque (same naming as for ordinary services) and `component_proplist` contains component variables, along with their values.

```

class NanoServiceCallbacks(ncs.application.NanoService):

    @ncs.application.NanoService.create
    def cb_nano_create(self, tctx, root, service, plan, component, state,
                       proplist, component_proplist):
        ...

    @ncs.application.NanoService.delete
    def cb_nano_delete(self, tctx, root, service, plan, component, state,
                       proplist, component_proplist):
        ...

```

In the majority of cases, you should not need to manage the status of nano states yourself. However, should you need to override the default behavior, you can set the status explicitly, in the callback, using code similar to the following :

```
plan.component[component].state[state].status = 'failed'
```

The Python nano service callback needs a registration call for the specific servicepoint, componentType, and state that it should be invoked for.

```

class Main(ncs.application.Application):

    def setup(self):
        ...
        self.register_nano_service('my-servicepoint',
                                  'my:some-component',
                                  'my:some-state',
                                  NanoServiceCallbacks)

```

For Java, annotations are used to define the callbacks for the component states. The registration of these callbacks is performed by the ncs-java-vm. The `NanoServiceContext` argument contains methods for retrieving the component and state for the invoked callback as well as methods for setting the resulting plan state status.

```

public class myRFS {

    @NanoServiceCallback(servicePoint="my-servicepoint",
                        componentType="my:some-component",
                        state="my:some-state",
                        callType=NanoServiceCBType.CREATE)
    public Properties createSomeComponentSomeState(
        NanoServiceContext context,
        NavuNode service,

```

```

        NavuNode ncsRoot,
        Properties opaque,
        Properties componentProperties)
    throws DpCallbackException {
    // ...
}

@NanoServiceCallback(servicePoint="my-servicepoint",
                     componentType="my:some-component",
                     state="my:some-state",
                     callType=NanoServiceCBType.DELETE)
public Properties deleteSomeComponentSomeState(
    NanoServiceContext context,
    NavuNode service,
    NavuNode ncsRoot,
    Properties opaque,
    Properties componentProperties)
throws DpCallbackException {
    // ...
}

```

Several componentType and state callbacks can be defined in the same Java class and are then registered by the same daemon.

Generic Service Callbacks

In some scenarios, there is a need to be able to register a callback for a certain state in several components with different component types. For this reason, it is possible to register a callback with a wildcard, using “*” as the component type. The invoked state sends the actual component name to the callback, allowing the callback to still distinguish component types if required.

In Python, the component type is provided as an argument to the callback (`component`) and a generic callback is registered with an asterisk for a component, such as:

```
self.register_nano_service('my-servicepoint', '*', state, ServiceCallbacks)
```

In Java, you can perform the registration in the method annotation, as before. To retrieve the calling component type, use the `NanoServiceContext.getComponent()` method. For example:

```

@NanoServiceCallback(servicePoint="my-servicepoint",
                     componentType="*", state="my:some-state",
                     callType=NanoServiceCBType.CREATE)
public Properties genericNanoCreate(NanoServiceContext context,
                                    NavuNode service,
                                    NavuNode ncsRoot,
                                    Properties opaque,
                                    Properties componentProperties)
throws DpCallbackException {

    String currentComponent = context.getComponent();
    // ...
}

```

The generic callback can then act for the registered state in any component type.

Nano Service Pre/Post Modifications

The ordinary service pre/post modification callbacks still exist for nano services. They are registered as for an ordinary service and are invoked before the behavior tree synthetization, and after the last component/state invocation.

Registration of the ordinary `create()` or `pre_lock_create()` does not fail for a nano service. But they are never invoked.

Forced Commits

When implementing a nano service, you might end up in a situation where a commit is needed between states in a component to make sure that something has happened before the service can continue executing. One example of such behaviour is if the service is dependent on the notifications from a device. In such a case, you can setup a notification kicker in the first state and then trigger a forced commit before any later states can proceed, therefore making sure that all future notifications are seen by the later states of the component.

To force a commit in between two states of a component, add the `ncs:force-commit` tag in a `ncs:create` or `ncs:delete` tag. See the following example:

```
ncs:component "self" {
    ncs:state "init" {
        ncs:create {
            ncs:force-commit;
        }
    }
    ncs:state "ready" {
        ncs:delete {
            ncs:force-commit;
        }
    }
}
```

Plan Location

When defining a nano service, it is assumed that the plan and the plan history data are stored under the service path, as `ncs:plan-data` and `ncs:plan-history` are added to the service definition. When the service instance is deleted, the plan is moved to the zombie instead, since the instance has been removed and the plan cannot be stored under it anymore. When writing other services or when working with a nano service in general, you need to be aware that the plan for a service might be in one of these two places depending on if the service instance has been deleted or not.

To make it easier to work with a service, you can define a custom location for the plan and its history. In the `ncs:service-behaviour-tree`, you can specify that the plan should be stored outside of the service by setting the `ncs:plan-location` tag to a custom location. The location where the plan should be stored must be either a list or a container and include the `ncs:plan-data` tag and optionally the `ncs:plan-history` tag. The plan data is then created in this location, no matter if the service instance has been deleted (turned into a zombie) or not, making it easy to base decisions on the state of the service as all plan queries can query the same plan.

You can use XPath with the `ncs:plan-location` statement. The XPath is evaluated based on the nano service context. When the list or container, which contains the plan, is nested under another list, the outer list instance must exist before creating the nano service. At the same time, the outer list instance of the plan location must also remain intact for further service's life-cycle management, such as redeployment, deletion etc. Otherwise, an error will be returned, logged, and any service interaction (create, re-deploy, delete, etc.) won't succeed.

Example 207. Nano services custom plan location example

```
list custom {
    description "Custom plan location example service.";
```

```

key name;
leaf name {
    tailf:info "Unique service id";
    tailf:cli-allow-range;
    type string;
}

uses ncs:service-data;
ncs:servicepoint custom-plan-servicepoint;
}

list custom-plan {
    description "Custom plan location example plan.';

    key name;
    leaf name {
        tailf:info "Unique service id";
        tailf:cli-allow-range;
        type string;
    }

    uses ncs:nano-plan-data;
    uses ncs:nano-plan-history;
}

ncs:plan-outline custom-plan {
    description
        "Custom plan location example outline";

    ncs:component-type "ncs:self" {
        ncs:state "ncs:init";
        ncs:state "ncs:ready";
    }
}
}

ncs:service-behavior-tree custom-plan-location-servicepoint {
    description
        "Custom plan location example service behaviour three.";

    ncs:plan-outline-ref custom:custom-plan;
    ncs:plan-location "/custom-plan";

    ncs:selector {
        ncs:create-component "'self'" {
            ncs:component-type-ref "ncs:self";
        }
    }
}
}

```

Nano Services and Commit Queue

The commit queue feature, described in the section called “Commit Queue” in *NSO 5.7 User Guide*, allows for increased overall throughput of NSO by committing configuration changes into an outbound queue item instead of directly to affected devices. Nano services are aware of the commit queue and will make use of it, however, this interaction requires additional consideration.

When the commit queue is enabled and there are outstanding commit queue items, the network is lagging behind the CDB. The CDB is forward looking and shows the desired state of the network. Hence, the nano plan shows the desired state as well, since changes to reach this state may not have been pushed to the devices yet.

To keep the convergence of the nano service in sync with the commit queue, nano services behave more asynchronously:

- A nano service does not make any progression while the service has an outstanding commit queue item. The outstanding item is listed under `plan/commit-queue` for the service, in normal or in zombie mode.
- On completion of the commit queue item, the nano plan comes in sync with the network. The outstanding commit queue item is removed from the list above and the system issues a **reactive-re-deploy** action to resume the progression of the nano service.
- Post-actions are delayed, while there is an outstanding commit queue item.
- Deleting a nano service always (even without a commit queue) creates a zombie and schedules its re-deploy to perform backtracking. Again, the re-deploy and, consequently, removal will not take place while there is an outstanding commit queue item.

The reason for such behavior is that commit queue items can fail. In case of a failure, the CDB and the network have diverged. In turn, the nano plan may have diverged and not reflect the actual network state if the failed commit queue item contained changes related to the nano service.

What is worse, the network may be left in an inconsistent state. To counter that, NSO supports multiple recovery options for the commit queue. Since NSO release 5.7, using the `rollback-on-error` is the recommended option, as it undoes all the changes that are part of the same transaction. If the transaction includes the initial service instance creation, the instance is removed as well. That is usually not desired for nano services. A nano service will avoid such removal by only committing the service intent (the instance configuration) in the initial transaction. In this case, the service avoids potential rollback, as it does not perform any device configuration in the same transaction but progresses solely through (reactive) re-deploy.

While error recovery helps keeping the network consistent, the end result remains that the requested change was not deployed. If a commit queue item with nano service related changes fails, that signifies a failure for the nano service and NSO does the following:

- Service progression stops.
- The nano plan is marked as failed by creating the `failed` leaf under the plan.
- The scheduled post actions are canceled. Canceled post actions stay in the `side-effect-queue` with status `canceled` and are not going to be executed.

After such an event, manual intervention is required. If not using the `rollback-on-error` option or rollback transaction fails, please consult the section called “Commit Queue” in *NSO 5.7 User Guide* for the correct procedure to follow. Once the cause of the commit queue failure is resolved, you can manually resume the service progression by invoking the **reactive-re-deploy** action on a nano service or a zombie.

Graceful Link Migration Example

You can find another nano service example under `examples.ncs/getting-started/developing-with-ncs/20-nano-services`. The example illustrates a situation with a simple VPN link that should be set up between two devices. The link is considered established only after it is tested and a `test-passed` leaf is set to `true`. If the VPN link changes, the new endpoints must be set up before removing the old endpoints, to avoid disturbing customer traffic during the operation.

The package named `link` contains the nano service definition. The service has a list containing at most one element, which constitutes the VPN link and is keyed on a-device a-interface b-device b-interface. The list element corresponds to a component-type "link:vlan-link" in the nano service plan.

Example 208. 20-nano-services link example plan

```

identity vlan-link {
    base ncs:plan-component-type;
}

identity dev-setup {
    base ncs:plan-state;
}

ncs:plan-outline link:link-plan {
    description
        "Make before brake vlan plan";

    ncs:component-type "ncs:self" {
        ncs:state "ncs:init";
        ncs:state "ncs:ready";
    }

    ncs:component-type "link:vlan-link" {
        ncs:state "ncs:init";
        ncs:state "link:dev-setup" {
            ncs:create {
                ncs:nano-callback;
            }
        }
        ncs:state "ncs:ready" {
            ncs:create {
                ncs:pre-condition {
                    ncs:monitor "$SERVICE/endpoints" {
                        ncs:trigger-expr "test-passed = 'true'";
                    }
                }
            }
            ncs:delete {
                ncs:pre-condition {
                    ncs:monitor "$SERVICE/plan" {
                        ncs:trigger-expr
                            "component[name != 'self'][back-track = 'false']"
                            "+ "/state[name = 'ncs:ready'][status = 'reached']"
                            "+ " or not(component[back-track = 'false'])";
                    }
                }
            }
        }
    }
}

```

In the plan definition, note that there is only one nano service callback registered for the service. This callback is defined for the "link:dev-setup" state in the "link:vlan-link" component type. In the plan, it is represented as follows:

```

ncs:state "link:dev-setup" {
    ncs:create {
        ncs:nano-callback;
    }
}

```

The callback is a template. You can find it under packages/link/templates as `link-template.xml`.

For the state "ncs:ready" in the "link:vlan-link" component type there are both a create and a delete pre-condition. The create pre-condition for this state is as follows:

```

ncs:create {
    ncs:pre-condition {
        ncs:monitor "$SERVICE/endpoints" {
            ncs:trigger-expr "test-passed = 'true'";
        }
    }
}

```

This pre-condition implies that the components based on this component type are not considered finished until the test-passed leaf is set to a "true" value. The pre-condition implements the requirement that after the initial setup of a link configured by the "link:dev-setup" state, a manual test and setting of the test-passed leaf is performed before the link is considered finished.

The delete pre-condition for the same state is as follows:

```

ncs:delete {
    ncs:pre-condition {
        ncs:monitor "$SERVICE/plan" {
            ncs:trigger-expr
                "component[name != 'self'][back-track = 'false']"
                + "/state[name = 'ncs:ready'][status = 'reached']"
                + " or not(component[back-track = 'false'])";
        }
    }
}

```

This pre-condition implies that before you start deleting (back-tracking) an old component, the new component must have reached the "ncs:ready" state, that is, after being successfully tested. The first part of the pre-condition checks the status of the non-self components. Since there can be at most one link configured in the service instance, the only non-backtracking component, other than self, is the new link component. However, that condition on its own prevents the component to be deleted when deleting the service. So, the second part, after the or statement, checks if all components are back-tracking, which signifies service deletion. This approach illustrates a "create-before-break" scenario where the new link is created first, and only when it is set up, the old one is removed.

Example 209. 20-nano-services link example behavior tree

```

ncs:service-behavior-tree link-servicepoint {
    description
        "Make before brake vlan example";
    ncs:plan-outline-ref "link:link-plan";

    ncs:selector {
        ncs:create-component "'self'" {
            ncs:component-type-ref "ncs:self";
        }
    }

    ncs:multiplier {
        ncs:foreach "endpoints" {
            ncs:variable "VALUE" {
                ncs:value-expr "concat(a-device, '-', a-interface,
                                '-', b-device, '-', b-interface)";
            }
        }
        ncs:create-component "$VALUE" {
            ncs:component-type-ref "link:vlan-link";
        }
    }
}

```

The the ncs:service-behavior-tree is registered on the servicepoint "link-servicepoint" that is defined by the nano service. It refers to the plan definition named "link:link-plan". The behavior tree has a selector on top, which chooses to synthesize its children depending on their pre-conditions. In this tree, there are no pre-conditions, so all children will be synthesized.

First, there is a component "self" based on the "ncs:self" component type in the plan that is always synthesized.

Second, there is a "multiplier" control node that chooses a node-set. A variable named VALUE is created with a unique value for each node in that node-set and creates a component of the "link:vlan-link" type for each node in the chosen node-set. The name for each individual component is the value of the variable VALUE.

Since the chosen node-set is the "endpoints" list that can contain at most one element, it produces only one component. However, if the link in the service is changed, that is, the old list entry is deleted and a new one is created, then the multiplier creates a component with a new name.

This forces the old component (which is no longer synthesized) to be back-tracked and the plan definition above handles the "create-before-break" behavior of the back-tracking.

To run the example, do the following:

Build the example:

```
$ cd examples.ncs/getting-started/developing-with-ncs/20-nano-services
$ make all
```

Start the example:

```
$ cd ncs-netsim restart
$ ncs
```

Run the example:

```
$ ncs_cli -C -u admin
admin@ncs(config)# devices sync-from
sync-result {
    device ex0
    result true
}
sync-result {
    device ex1
    result true
}
sync-result {
    device ex2
    result true
}
admin@ncs(config)# config
Entering configuration mode terminal
```

Now you create a service that sets up a VPN link between devices ex1 and ex2, and is completed immediately since the test-passed leaf is set to true.

```
admin@ncs(config)# link t2 unit 17 vlan-id 1
admin@ncs(config-link-t2)# link t2 endpoints ex1 eth0 ex2 eth0 test-passed true
admin@ncs(config-endpoints-ex1/eth0/ex2/eth0)# commit
admin@ncs(config-endpoints-ex1/eth0/ex2/eth0)# top
```

You can inspect the result of the commit:

```

admin@ncs(config)# exit
admin@ncs# link t2 get-modifications
cli devices {
    device ex1 {
        config {
            r:sys {
                interfaces {
                    interface eth0 {
                        unit 17 {
                            vlan-id 1;
                        }
                    }
                }
            }
        }
    }
    device ex2 {
        config {
            r:sys {
                interfaces {
                    interface eth0 {
                        unit 17 {
                            vlan-id 1;
                        }
                    }
                }
            }
        }
    }
}

```

The service sets up the link between the devices. Inspect the plan:

```

admin@ncs# show link t2 plan component * state * status
NAME          STATE      STATUS
-----
self          init       reached
              ready      reached
ex1-eth0-ex2-eth0  init       reached
                  dev-setup  reached
                  ready      reached

```

All components in the plan have reached their ready state.

Now, change the link by changing the interface on one of the devices. To do this, you must remove the old list entry in "endpoints" and create a new one.

```

admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# no link t2 endpoints ex1 eth0 ex2 eth0
admin@ncs(config)# link t2 endpoints ex1 eth0 ex2 eth1

```

Commit dry-run to inspect what happens:

```

admin@ncs(config-endpoints-ex1/eth0/ex2/eth1)# commit dry-run
cli devices {
    device ex1 {
        config {
            r:sys {
                interfaces {
                    interface eth0 {
                }
            }
        }
    }
}

```

```

        }
    }
}
device ex2 {
    config {
        r:sys {
            interfaces {
+                interface eth1 {
+                    unit 17 {
+                        vlan-id 1;
+
+                    }
+
+                }
            }
        }
    }
link t2 {
-    endpoints ex1 eth0 ex2 eth0 {
-        test-passed true;
-
+    endpoints ex1 eth0 ex2 eth1 {
+
+    }
}

```

Upon committing, the service just adds the new interface and does not remove anything at this point. The reason is that the test-passed leaf is not set to "true" for the new component. Commit this change and inspect the plan:

```

admin@ncs(config-endpoints-ex1/eth0/ex2/eth1)# commit
admin@ncs(config-endpoints-ex1/eth0/ex2/eth1)# top
admin@ncs(config)# exit
admin@ncs# show link t2 plan

```

NAME	TYPE	BACK TRACK	GOAL	STATE	STATUS	...
self	self	false	-	init	reached	...
				ready	reached	...
ex1-eth0-ex2-eth1	vlan-link	false	-	init	reached	...
				dev-setup	reached	...
				ready	not-reached	...
ex1-eth0-ex2-eth0	vlan-link	true	-	init	reached	...
				dev-setup	reached	...
				ready	reached	...

Notice that the new component "ex1-eth0-ex2-eth1" has not reached its ready state yet. Therefore, the old component "ex1-eth0-ex2-eth0" still exists in back-track mode but is still waiting for the new component to finish.

If you check what the service has configured at this point, you get the following:

```

admin@ncs# link t2 get-modifications
cli devices {
    device ex1 {
        config {
            r:sys {
                interfaces {
                    interface eth0 {
+
                        unit 17 {
+
                            vlan-id 1;
+
                        }
}
}
}
}
}

```

```

        }
    }
}
device ex2 {
    config {
        r:sys {
            interfaces {
                interface eth0 {
                    unit 17 {
                        vlan-id 1;
                    }
                }
                interface eth1 {
                    unit 17 {
                        vlan-id 1;
                    }
                }
            }
        }
    }
}

```

Both the old and the new link exist at this point. Now, set the test-passed leaf to true to force the new component to reach its ready state.

```
admin@ncs(config)# link t2 endpoints ex1 eth0 ex2 eth1 test-passed true
admin@ncs(config-endpoints-ex1/eth0/ex2/eth1)# commit
```

If you now check the service plan, you see the following:

```
admin@ncs(config-endpoints-ex1/eth0/ex2/eth1)# top
admin@ncs(config)# exit
admin@ncs# show link t2 plan
      BACK
NAME      TYPE   TRACK GOAL STATE STATUS ...
-----
self      self   false -     init  reached ...
                                ready  reached ...
ex1-eth0-ex2-eth1  wlan-link  false -     init  reached ...
                                dev-setup  reached ...
                                ready  reached ...
```

The old component has been completely back-tracked and is removed because the new component is finished. You should also check the service modifications. You should see that the old link endpoint is removed:

```
admin@ncs# link t2 get-modifications
cli devices {
    device ex1 {
        config {
            r:sys {
                interfaces {
                    interface eth0 {
                        unit 17 {
                            vlan-id 1;
                        }
                    }
                }
            }
        }
    }
}
```

```
        }
    }
}
device ex2 {
    config {
        r:sys {
            interfaces {
                interface eth1 {
                    unit 17 {
                        vlan-id 1;
                    }
                }
            }
        }
    }
}
```



CHAPTER 23

Encryption Keys

- [Introduction, page 463](#)
- [Reading encryption keys using an external command, page 463](#)

Introduction

By using the `tailf:des3-cbc-encrypted-string`, `tailf:aes-cfb-128-encrypted-string` or the `tailf:aes-256-cfb-128-encrypted-string` built-in types it is possible to store encrypted values in NSO. The keys used to encrypt these values are configured in `ncs.conf` and default stored in `ncs.crypto_keys`.

Reading encryption keys using an external command

NSO supports reading encryption keys using an external command instead of storing them in `ncs.conf` to allow for use with external key management systems. To use this feature set `/ncs-config/encrypted-strings/external-keys/command` to an executable command that will output the keys following the rules described in the following sections. The command will be executed on startup and when NSO reloads the configuration.

If the external command fails during startup, the startup will abort. If the command fails during a reload the error will be logged and the previously loaded keys will be kept in the system.

The process of providing encryption keys to NSO can be described by the following three steps:

- 1 Read configuration from environment.
- 2 Read encryption keys.
- 3 Write encryption keys or error on standard output.

The value of `/ncs-config/encrypted-strings/external-keys/command-argument` is available in the command as the environment variable `NCS_EXTERNAL_KEYS_ARGUMENT`. The value of this configuration is only used by the configured command.

The external command should return the encryption keys on standard output using the names as shown in the table below. The encryption key values are in hexadecimal format, just as in `ncs.conf`. See the example below for details.

Table 210. Mapping from name to path in configuration.

Name	Configuration path
DES3CBC_KEY1	/ncs-config/encrypted-strings/DES3CBC/key1

Name	Configuration path
DES3CBC_KEY2	/ncs-config/encrypted-strings/DES3CBC/key2
DES3CBC_KEY3	/ncs-config/encrypted-strings/DES3CBC/key3
DES3CBC_IV	/ncs-config/encrypted-strings/DES3CBC/initVector
AESCFB128_KEY	/ncs-config/encrypted-strings/AESCFB128/key
AESCFB128_IV	/ncs-config/encrypted-strings/AESCFB128/initVector
AES256CFB128_KEY	/ncs-config/encrypted-strings/AES256CFB128/key

To signal an error, including ERROR=message is preferred. A non-zero exit code or unsupported line content will also trigger an error. Any form of error will be logged to the development log and no encryption keys will be available in the system.

Example output providing all supported encryption key configuration settings:

```
DES3CBC_KEY1=12785c357764a327
DES3CBC_KEY2=30661368c90bc26d
DES3CBC_KEY3=10604b6b63e09310
DES3CBC_IV=f04ab44ed14c3d76
AESCFB128_KEY=2b57c219e47582481b733c1adb84fc26
AESCFB128_IV=549a40ed57629bf6ea64b568f221b515
AES256CFB128_KEY=3c687d564e250ad987198d179537af563341357493ed2242ef3b16a881dd608c
```

Example error output:

```
ERROR=error message
```

Below follows a complete example of an application written in Python providing encryption keys from a plain text file. The application is included in the example `crypto/external_keys`:

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

import os
import sys

def main():
    key_file = os.getenv('NCS_EXTERNAL_KEYS_ARGUMENT', None)
    if key_file is None:
        error('NCS_EXTERNAL_KEYS_ARGUMENT environment not set')
    if len(key_file) == 0:
        error('NCS_EXTERNAL_KEYS_ARGUMENT is empty')

    try:
        with open(key_file, 'r') as f_obj:
            keys = f_obj.read()
            sys.stdout.write(keys)
    except Exception as ex:
        error('unable to open/read {}: {}'.format(key_file, ex))

def error(msg):
    print('ERROR={}'.format(msg))
    sys.exit(1)

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



CHAPTER 24

External Logging

- [Introduction, page 465](#)
- [Enabling external log processing, page 465](#)
- [Processing logs using an external command, page 466](#)

Introduction

As a development feature NSO supports sending log data as-is to an external command for reading on standard input. As this is a development feature there are a few limitations such as the data sent to the external command is not guaranteed to be processed before the external application is shut down.

Enabling external log processing

General configuration of the external log processing is done in `ncs.conf`. Global and per device settings controlling the external log processing for NED trace logs is stored in CDB.

To enable external log processing set `/ncs-config/logs/external` to `true` and `/ncs-config/logs/command` to the full path of the command that will receive the log data. The same executable will be used for all log types. External configuration example:

```
<external>
  <enabled>true</enabled>
  <command>./path/to/log_filter</command>
</external>
```

To support debugging of the external log command behavior a separate log file is used. This debugging log is configured under `/ncs-config/logs/ext-log`. The example below shows configuration for `./logs/external.log` with the highest log level set:

```
<ext-log>
  <enabled>true</enabled>
  <filename>./logs/external.log</filename>
  <level>7</level>
</ext-log>
```

By default NED trace output is written to file preserving backwards compatibility. To write NED trace logs to file for all but the device `test` which will use external log processing the following configuration can be entered in the CLI:

```
# devices global-settings trace-output file
# devices device example trace-output external
```

When setting both `external` and `file` bits without setting `/ncs-config/logs/external` to `true` a warning message will be logged to `ext-log`. When only setting the `external` bit no logging will be done.

Processing logs using an external command

After enabling external log processing, NSO will start one instance of the external command for each configured log destination. Processing of the log data is done by reading from standard input and processing it as required.

The command line arguments provide information about the log that is being processed and in what format the data is sent.

The example below show how the configured command `./log_processor` would be executed for `NETCONF trace` data configured to log in raw mode:

```
./log_processor 1 log "NETCONF Trace" netconf-trace raw
```

Command line argument position and meaning:

- 1 *version*. Protocol version, always set to *1*. Added for forwards compatibility.
- 2 *action*. Action being performed, always set to *log*. Added for forwards compatibility.
- 3 *name*. Name of the log being processed.
- 4 *log-type*. Type of log data being processed. For all but NETCONF and NED trace logs this is set to *system*. Depending on the type of NED one of *ned-trace-java*, *ned-trace-netconf* and *ned-trace-snmp* is used. NETCONF trace is set to *netconf-trace*.
- 5 *log-mode*. Format of log data being sent. For all but NETCONF and NED trace logs this will be *raw*. NETCONF and NED trace logs can be pretty printed and then format will be *pretty*.