



Development Guide

Release: NSO 6.2.5

Published: May 17, 2010

Last Modified: May 14, 2024

Americas Headquarters

Cisco Systems, Inc.

170 West Tasman Drive

San Jose, CA 95134-1706

USA

<http://www.cisco.com>

Tel: 408 526-4000

800 553-NETS (6387)

Fax: 408 527-0883

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, 2021-2024 Cisco Systems, Inc. All rights reserved.



CONTENTS

CHAPTER 1	Preface	1
CHAPTER 2	Development Environment and Resources	3
	Development NSO Instance	3
	Examples and Showcases	4
	IDE	4
	Automating Instance Setup	5
CHAPTER 3	The Configuration Database and YANG	9
	Key Features of the CDB	9
	Compilation and Loading of YANG Modules	10
	Showcase: Extending the CDB with Packages	10
	Data Modeling Basics	12
	Showcase: Building and Testing a Model	14
	Initialization Files	16
CHAPTER 4	Basic Automation with Python	19
	Setup	19
	Transactions	20
	Read and Write Values	20
	Lists	21
	Showcase: Configuring DNS with Python	22
	A Note on Robustness	25
	The Magic Behind the API	25
CHAPTER 5	Developing a Simple Service	27
	Why services?	27
	The Service Package	28
	Service Parameters	30



Showcase: A Simple DNS Configuration Service	31
Service Templates	35
Showcase: DNS Configuration Service with Templates	37

CHAPTER 6

Applications in NSO	41
NSO Architecture	41
Callbacks as Extension Mechanism	42
Actions	44
Showcase: Implementing Device Count Action	45
Overview of Extension Points	49
Monitoring for Change	50
Running Application Code	50
Application Updates	51

CHAPTER 7

Implementing Services	53
Introduction	53
Service Mapping	54
A Template is All You Need	56
Service Model Captures Inputs	59
Extracting the Service Parameters	62
FASTMAP and Service Life Cycle	66
Templates and Code	68
Templates and Python Code	70
Templates and Java Code	72
Configuring Multiple Devices	73
Supporting Different Device Types	75
Shared Service Settings and Auxiliary Data	78
Service Actions	80
Action Code in Python	81
Action Code in Java	82
Operational Data	83
Writing Operational Data in Python	85
Writing Operational Data in Java	86
Nano Services for Provisioning with Side Effects	87
Service Troubleshooting	88

	Common troubleshooting steps	89
CHAPTER 8	Templates	93
	Structure of a template	94
	Other ways to generate the XML template structure	95
	Values in a template	97
	Conditional statements	98
	Loop statements	99
	Template operations	100
	Operations on ordered lists and leaf-lists	100
	Macros in templates	102
	XPath context in templates	104
	Namespaces and multi-NED support	105
	Passing deep structures from API	106
	Service callpoints and templates	108
	Debugging templates	109
	Example debug template output	110
	Processing instructions reference	112
	XPath functions	115
CHAPTER 9	Services Deep Dive	117
	Common Service Model	117
	Services and Transactions	118
	Service Callbacks	121
	Persistent Opaque Data	122
	Defining Static Service Conflicts	123
	Reference Counting Overlapping Configuration	124
	Stacked Services	125
	Caveats and Best Practices	126
	Service Discovery and Import	128
	Matching configuration	129
	Enumerating instances	130
	Reconciliation	131
	Iterative approach	133
	Partial Sync	135

CHAPTER 10

The NSO Java VM	137
Overview	137
YANG model	139
Java Packages and the Class Loader	139
The NED component type	141
The callback component type	141
The application component type	141
The Resource Manager	142
The Alarm centrals	144
Embedding the NSO Java VM	144
Logging	145
The NSO Java VM timeouts	145
Debugging Startup	146

CHAPTER 11

The NSO Python VM	149
Introduction	149
YANG model	149
Structure of the User provided code	151
The application component	152
The upgrade component	153
Debugging of Python packages	155
Using non-standard Python	155
Configure NSO to use a custom start command (recommended)	156
Changing the path to <i>python3</i> or <i>python</i>	156
Updating the default start command (not recommended)	156
Caveats	156
Using multiprocessing	156

CHAPTER 12

Embedded Erlang applications	157
Introduction	157
Erlang API	157
Application configuration	158
Example	159

CHAPTER 13

The YANG Data Modeling Language	161
The YANG Data Modeling Language	161
YANG in NSO	161
YANG Introduction	162
Modules and Submodules	162
Data Modeling Basics	162
Leaf Nodes	162
Leaf-list Nodes	163
Container Nodes	163
List Nodes	163
Example Module	164
State Data	165
Built-in Types	165
Derived Types (typedef)	166
Reusable Node Groups (grouping)	166
Choices	167
Extending Data Models (augment)	168
RPC Definitions	169
Notification Definitions	169
Working With YANG Modules	169
Integrity Constraints	171
The when statement	172
Using the Tail-f Extensions with YANG	173
Using a YANG annotation file	174
Custom Help Texts and Error Messages	175
Custom Help Texts	175
Custom Help Text in a Typedef	175
Custom Error Messages	176
An Example: Modeling a List of Interfaces	176
Modeling Relationships	179
Ensuring Uniqueness	180
Default Values	181
The Final Interface YANG model	182
More on leafrefs	184
Using Multiple Namespaces	185

CHAPTER 14

Module Names, Namespaces and Revisions	186
Hash Values and the id-value Statement	187
NSO caveats	188
The union type and value conversion	188
User defined types	189

CHAPTER 15

Using CDB	191
Introduction	191
The NSO Data Model	192
Addressing Data Using Keypaths	194
Subscriptions	195
Sessions	196
Loading Initial Data Into CDB	197
Operational Data in CDB	198
Subscriptions	198
Example	199
Operational Data	206
Automatic Schema Upgrades and Downgrades	210
Using Initialization Files for Upgrade	212
New Validation Points	215
Writing an Upgrade Package Component	215
Java API Overview	227
Introduction	227
MAAPI	230
CDB API	233
DP API	236
Transaction and Data Callbacks	237
Service and Action Callbacks	245
Validation Callbacks	246
Transforms	247
Hooks	247
NED API	248
NAVU API	248
ALARM API	256

NOTIF API	258
HA API	260
Java API Conf Package	260
Namespace classes and the loaded schema	263
CHAPTER 16	
Python API Overview	265
Introduction	265
Python API overview	266
Python scripting	267
High-level MAAP API	267
Maagic API	268
Namespaces	269
Reading data	269
Writing data	270
Deleting data	270
Containers	270
Presence containers	270
Choices	270
Lists and List elements	271
Unions	271
Enumeration	271
Leafref	272
Identityref	272
Instance-identifier	272
Leaf-list	272
Binary	273
Bits	273
Empty leaf	274
Maagic examples	274
Action requests	274
PlanComponent	277
Python packages	278
Action handler	278
Service handler	279
ValidationPoint handler	281



CHAPTER 17	NSO Packages	289
	Package Overview	289
	An Example Package	291
	The package-meta-data.xml file	291
	Components	295
	Creating Packages	297
	Creating a NETCONF NED Package	297
	Creating an SNMP NED Package	298
	Creating a CLI NED Package or a Generic NED Package	298
	Creating a Service Package or a Data Provider Package	298
CHAPTER 18	Package Development	299
	Developing a Service Package	299
	ncs-setup	301
	The netsim Part of a NED Package	302
	Plug-and-play scripting	304
	Creating a Service Package	304
	Java Service Implementation	305
	Developing our First Service Application	305
	Tracing Within the NSO Service Manager	308
	Controlling error messages info level from Java	311
	Loading Packages	312
	Debugging the Service and Using Eclipse IDE	312
	Running the NSO Java VM Standalone	314
	Using Eclipse to Debug the Package Java Code	315
	Remote Connecting with Eclipse to the NSO Java VM	316
	Working with ncs-project	316
	Create a new project	316
	Project setup	317

Export	319
NSO Project man pages	320
The project-meta-data.xml file	321

CHAPTER 19 Service Development Using Java 325

Simple VLAN service	325
Overview	325
Setting up the environment	326
Creating a service package	327
The Service Model	328
Managing the NSO Java VM	329
A first look at Java Development	330
Using Eclipse	331
Writing the service code	336
Simple VLAN service with templates	340
Overview	340
The VLAN Feature Template	341
The VLAN Java Logic	342
Steps to Build a Java and Template Solution	343
Layer 3 MPLS VPN service	343
Auxiliary Service Data	344
Topology	344
Creating a Multi-Vendor Service	346
Defining the Mapping	350

CHAPTER 20 NED Development 355

Creating a NED	355
Types of NED Packages	355
Dumb Versus Capable Devices	356
NETCONF NED Development	358
Tools for NETCONF NED Development	359
Using the netconf-console and ncs-make-package Combination	359
Using the NETCONF NED Builder Tool	362
CLI NED Development	370
Writing a data model for a CLI NED	372



Tweaking the basic rendering scheme	374	
Modifying the Java part of the CLI NED	384	
Tail-f CLI NED Annotations	390	
Annotations	391	
Generic NED Development	428	
Getting started with a generic NED	429	
NED commands	431	
The SNMP NED	432	
Introduction	432	
Overview	432	
Compiling and loading MIBs	433	
Configuring NSO to speak SNMP southbound	434	
Annotations for MIB objects	436	
Using the SNMP NED	437	
Statistics	439	
Making the NED handle default values properly	442	
Dry-run considerations	445	
NED identification	446	
Migrating to the <code>juniper-junos_nc-gen</code> NED	447	
Prepare the Example	448	
Adapting the Service to the <code>juniper-junos_nc-gen</code> NED	451	
<hr/>		
CHAPTER 21	NED Upgrades and Migration	457
	Introduction	457
	The Migrate Action	457
<hr/>		
CHAPTER 22	Service Handling of Ambiguous Device Models	459
	Template Services	459
	Java Services	460
	Python Services	461
<hr/>		
CHAPTER 23	Scaling and Performance Optimization	463
	Understanding Your Use Case	463
	Where to Start?	465
	Divide the Work Correctly	465
	Optimizing Device Communication	465

Improving Subscribers	466
Minimizing Concurrency Conflicts	466
Fine-tuning the Concurrency Parameters	467
Enabling Even More Parallelism	467
Limit sync-from	468
Designing for Maximal Transaction Throughput	468
Measuring Transaction Throughput	469
Running the perf-trans Example Using a Single Transaction	470
Concurrent Transactions	471
Design to Minimize Conflicts	472
Design to Minimize Service and Validation Processing Time	473
Use a Data-Kicker Instead of a CDB Subscriber	475
Shorten the Time Used for Writing Configuration to Devices	475
Running the perf-trans Example Using One Transaction per Device	475
Using Commit Queues	477
Enabling Commit Queues for the perf-trans Example	477
Simplify the Per-Device Concurrent Transaction Creation Using a Nano Service	479
Simplify Using a CFS	480
Running the CFS and Nano Service Enabled perf-stack Example	481
Migrating to and Scale Up Using an LSA Setup	483
Running the LSA-enabled perf-lsa Example	484
Scaling RAM and Disk	488
CDB	488
Services and Devices in CDB	488
CDB Stores the YANG Model Schema	489
The Size of CDB	489
Devices, Small and Large	489
Planning Resource Consumption	492
The Size of a Service	492
Implications of a Large CDB	492
Checklists	493
Hardware Sizing	494
Lab Testing and Development	494
Production	494

CHAPTER 24	NSO Concurrency Model	495
	Optimistic Concurrency	495
	Identifying Conflicts	498
	Automatic Retries	499
	Handling Conflicts	500
	Example Retrying Code in Python	501
	Example Retrying Code in Java	502
	Designing for Concurrency	504
	Avoiding Needless Reads	504
	Handling Dependent Services	505
	Searching and Enumerating Lists	505
	Python Assigning to Self	506
CHAPTER 25	Developing Alarm Applications	507
	Introduction	507
	Using the Alarms Sink	508
	Using the Alarms Source	510
	Extending the alarm manager, adding user defined alarm types and fields	511
	Mapping alarms to objects	513
CHAPTER 26	SNMP Notification Receiver	515
	Introduction	515
	Configuring NSO to Receive SNMP Notifications	516
	Built-in Filters	516
	Notification Handlers	517
CHAPTER 27	The web server	521
	Introduction	521
	Web server capabilities	521
	CGI support	522
	Storing TLS data in database	523
	Package upload	525
CHAPTER 28	Kicker	527
	Introduction	527
	Kicker action invocation	527

	Data Kicker Concepts	528
	Generalized Monitors	528
	Kicker Constraints/Filters	530
	Variable Bindings	531
	A Simple Data Kicker Example	532
	Notification Kicker Concepts	534
	Notification selector expression	535
	Variable Bindings	535
	Serializer and priority values	536
	A Simple Notification Kicker Example	536
	Nano Services Reactive FastMap with Kicker	538
	Debugging kickers	538
	Kicker CLI Debug target	538
	Unhide Kickers	538
	XPath log	538
	Devel Log	539
<hr/>		
CHAPTER 29	Scheduler	541
	Introduction	541
	Scheduling Periodic Work	541
	Schedule Expression	542
	Periodic compaction	542
	Scheduling Non-recurring Work	542
	Scheduling in a HA Cluster	542
	Troubleshooting	543
	History log	543
	XPath log	543
	Devel Log	543
	Suspending the Scheduler	543
<hr/>		
CHAPTER 30	Progress Trace	545
	Introduction	545
	Configuring Progress Trace	546
	Unhide Progress Trace	546
	Log to File	546

**CHAPTER 31**

Nano Services for Staged Provisioning	551
Basic Concepts	552
Plan Outline	552
Per-State Configuration	554
Link Plan Outline to Service	554
Service Instantiation	556
Benefits and Use Cases	558
Backtracking and Staged Delete	559
Managing Side Effects	562
Multiple and Dynamic Plan Components	564
Netsim Router Provisioning Example	566
Zombie Services	569
Using Notifications to Track the Plan and its Status	570
The <code>plan-state-change</code> Notification	570
The <code>service-commit-queue-event</code> Notification	570
Examples of <code>service-state-changes</code> Stream Subscriptions	571
The <code>trace-id</code> in the Notification	572
Developing and Updating a Nano Service	573
Adding Components	573
Removing Components	573
Replacing Components	573
Adding and Removing States	574
Modifying States	574
Implementation Reference	574
Back-Tracking	575
Behavior Tree	578
Nano Service Pre-Condition	580

Nano Service Opaque and Component Properties	581
Nano Service Callbacks	582
Generic Service Callbacks	585
Nano Service Pre/Post Modifications	585
Forced Commits	585
Plan Location	586
Nano Services and Commit Queue	587
Graceful Link Migration Example	588
<hr/>	
CHAPTER 32	Encryption Keys 595
Introduction	595
Reading encryption keys using an external command	595
<hr/>	
CHAPTER 33	External Logging 597
Introduction	597
Enabling external log processing	597
Processing logs using an external command	598
<hr/>	
CHAPTER 34	NSO Developer Studio 599
Introduction	599
Contribute	599
NSO Developer Studio - Developer extension	599
System requirements	600
Install the extension	600
Make a new service package (Python only)	600
Edit YANG files	601
Edit Python files	604
Edit XML template files	604
Limitations	605
NSO Developer Studio - Explorer extension	606
System requirements	606
Install the extension	606
Connect to NSO instance	606
Inspect the CDB tree	607



CHAPTER 1

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 2

Development Environment and Resources

This chapter describes some recipes, tools, and other resources that you may find useful throughout the guide. The topics are tailored to novice users and focus on making development with NSO a more enjoyable experience.

- [Development NSO Instance, page 3](#)
- [Examples and Showcases, page 4](#)
- [IDE, page 4](#)
- [Automating Instance Setup, page 5](#)

Development NSO Instance

Many developers prefer their own, dedicated NSO instance to avoid their work clashing with other team members. You can use either a local or remote Linux machine (such as a VM), or a macOS computer for this purpose.

The advantage of running local Linux with a GUI or macOS is that it is easier to set up the Integrated Development Environment (IDE) and other tools when they run on the same system as NSO. However, many IDEs today also allow working remotely, such as through the SSH protocol, making the choice of local versus remote less of a concern.

For development, using the so-called “local install” of NSO has some distinct advantages:

- Does not require elevated privileges to install or run.
- Keeps all NSO files in the same place (user-defined).
- Allows you to quickly switch between projects and NSO versions.

If you work with multiple projects in parallel, local install also allows you to take advantage of Python virtual environments to separate Python packages per project; simply start the NSO instance in an environment you have activated.

The main downside of using a local install is that it differs slightly from a system (production) install, such as in the filesystem paths used and the out-of-the-box configuration.

See the section called “Local Install Steps” in *Getting Started* for installation instructions.

Examples and Showcases

There are a number of examples and showcases in this guide. We encourage you to follow them through. They are also a great reference if you are experimenting with a new feature and have trouble getting it to work; you can inspect and compare with the implementation in the example.

To run the examples, you will need access to an NSO instance. A development instance described in this chapter is the perfect option for running locally. See Chapter 4, *Running NSO Examples* in *Getting Started*.

Cisco also provides online sandbox and containerized environments, such as a [Learning Lab](#) or [NSO Sandbox](#), designed for this purpose. Please refer to the [online documentation](#) for additional options.

IDE

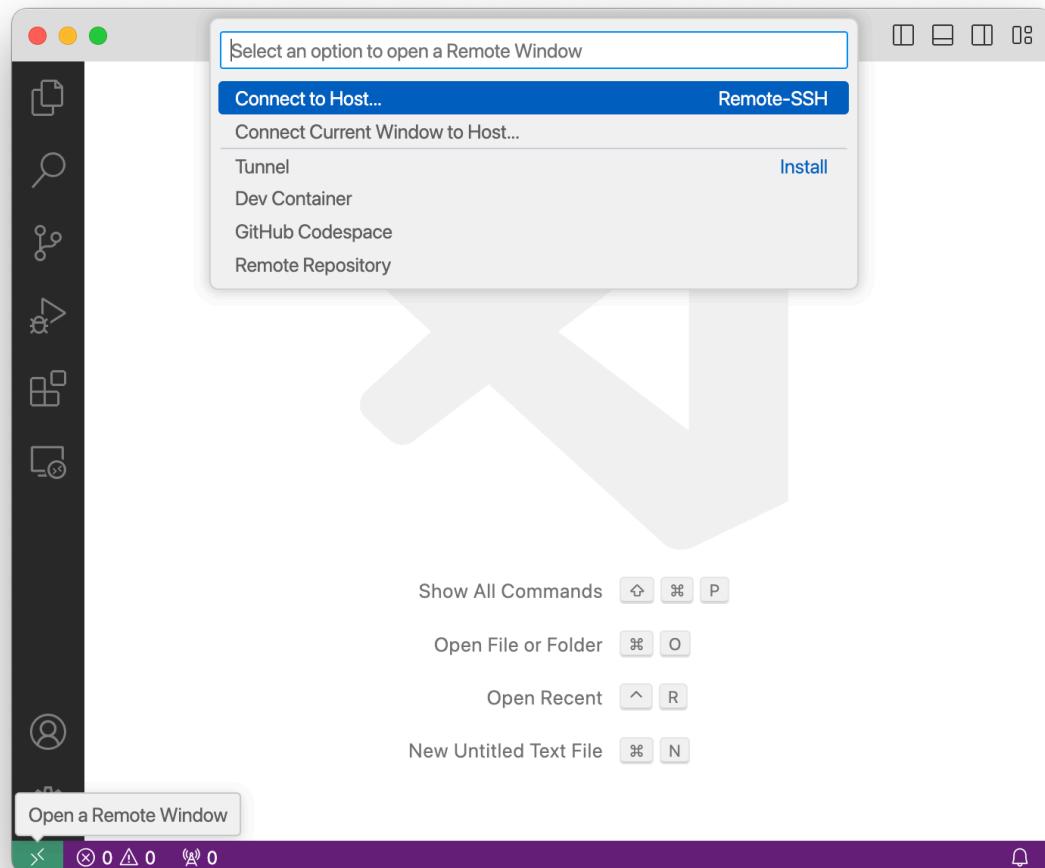
Modern IDEs offer many features on top of advanced file editing support, such as code highlighting, syntax checks, and integrated debugging. While the initial setup takes some effort, the benefits of using an IDE are immense.

[Visual Studio Code](#) (VS Code) is a freely-available and extensible IDE. You can add support for Java, Python, and YANG languages, as well as remote access through SSH via VS Code extensions. Consider installing the following extensions:

- [Python](#) by Microsoft: Adds Python support.
- [Language Support for Java\(TM\)](#) by Red Hat: Adds Java support.
- [NSO Developer Studio](#) by Cisco: Adds NSO-specific features as described in [Chapter 34, NSO Developer Studio](#).
- [Remote - SSH](#) by Microsoft: Adds support for remote development.

The *Remote - SSH* extension is especially useful when you must work with a system through an SSH session. Once you connect to the remote host by clicking the >< button (typically found in the bottom-left corner of the VS Code window), you can open and edit remote files with ease. If you also want language support (syntax highlighting and alike), you may need to install VS Code extensions remotely. That is, install the extensions after you have connected to the remote host, otherwise the extension installation screen might not show the option for installation on the connected host.

Figure 1. Using the Remote - SSH extension in VS Code



You will also benefit greatly from setting up SSH certificate authentication if you are using an SSH session for your work.

Automating Instance Setup

Once you get familiar with NSO development and gain some experience, a single NSO instance is likely to be insufficient; either because you need instances for unit testing, because you need one-off (throwaway) instances for an experiment, or something else entirely.

NSO includes tooling to help you quickly set up new local instances when such a need arises.

The following recipe relies on the **ncs-setup** command, which is available in the local install variant and requires a correctly set up shell environment (e.g. running **source ncsrc**). See the section called “Local Install Steps” in *Getting Started* for details.

A new instance typically needs a few things to be useful:

- Packages

- Initial data
- Devices to manage

In its simplest form, the **ncs-setup** invocation requires only a destination directory. However, you can specify additional packages to use with the **--package** option. Use the option to add as many packages as you need.

Running **ncs-setup** creates the required filesystem structure for an NSO instance. If you wish to include initial configuration data, put the XML-encoded data in the **ncs-cdb** subdirectory and NSO will load it at first start, as described in the section called “Initialization Files”.

NSO also needs to know about the managed devices. In case you are using **ncs-netsim** simulated devices (described in Chapter 14, *Network Simulator* in *User Guide*), you can use the **--netsim-dir** option with **ncs-setup** to add them directly. Otherwise, you may need to create some initial XML files with the relevant device configuration data — much like how you would add a device to NSO manually.

Most of the time, you must also invoke a sync with the device so that it performs correctly with NSO. If you wish to push some initial configuration to the device, you may add the configuration in the form of initial XML data and perform a **sync-to**. Alternatively, you can simply do a **sync-from**. You can use the **ncs_cmd** command for this purpose.

Combining all of this together, consider the following example:

- 1 Start by creating a new directory to hold the files:

```
$ mkdir nso-throwaway
$ cd nso-throwaway
```

- 2 Create and start a few simulated devices with **ncs-netsim**, using **./netsim** as directory:

```
$ ncs-netsim ncs-netsim create-network $NCS_DIR/packages/neds/cisco-ios-cli-3.8 3 c
DEVICE c0 CREATED
DEVICE c1 CREATED
DEVICE c2 CREATED
$ ncs-netsim start
```

- 3 Next, create the running directory with the NED package for the simulated devices and one more package. Also add configuration data to NSO on how to connect to these simulated devices.

```
$ ncs-setup --dest ncs-run --netsim-dir ./netsim \
--package $NCS_DIR/packages/neds/cisco-ios-cli-3.8 \
--package $NCS_DIR/packages/neds/cisco-iosxr-cli-3.0
```

- 4 Now you can add custom initial data as XML files to **ncs-run/ncs-cdb/**. Usually you would use existing files but you can also create them on-the-fly.

```
$ cat >ncs-run/ncs-cdb/my_init.xml <<'EOF'
<config xmlns="http://tail-f.com/ns/config/1.0">
  <session xmlns="http://tail-f.com/ns/aaa/1.1">
    <idle-timeout>0</idle-timeout>
  </session>
</config>
EOF
```

- 5 At this point, you are ready to start NSO:

```
$ cd ncs-run
$ ncs
```

- 6 Finally, request an initial **sync-from**:

```
$ ncs_cmd -u admin -c 'maction /devices/sync-from'
sync-result begin
```

```
device c0
  result true
sync-result end
sync-result begin
  device c1
  result true
sync-result end
sync-result begin
  device c2
  result true
sync-result end
```

- 7 The instance is now ready for work. Once you are finished, you can stop it with **ncs --stop**. Remember to also stop the simulated devices with **ncs-netsim stop** if you no longer need them. Then, delete the containing folder (`nso-throwaway`) to remove all the leftover files and data.



CHAPTER 3

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 9](#)
- [Compilation and Loading of YANG Modules, page 10](#)
- [Showcase: Extending the CDB with Packages, page 10](#)
- [Data Modeling Basics, page 12](#)
- [Showcase: Building and Testing a Model, page 14](#)
- [Initialization Files, page 16](#)

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 ns0
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 *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 *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 2. 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 3. 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 4

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 19](#)
- [Transactions, page 20](#)
- [Read and Write Values, page 20](#)
- [Lists, page 21](#)
- [Showcase: Configuring DNS with Python, page 22](#)
- [A Note on Robustness, page 25](#)
- [The Magic Behind the API, page 25](#)

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 *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 4. 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 5. 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 6. 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

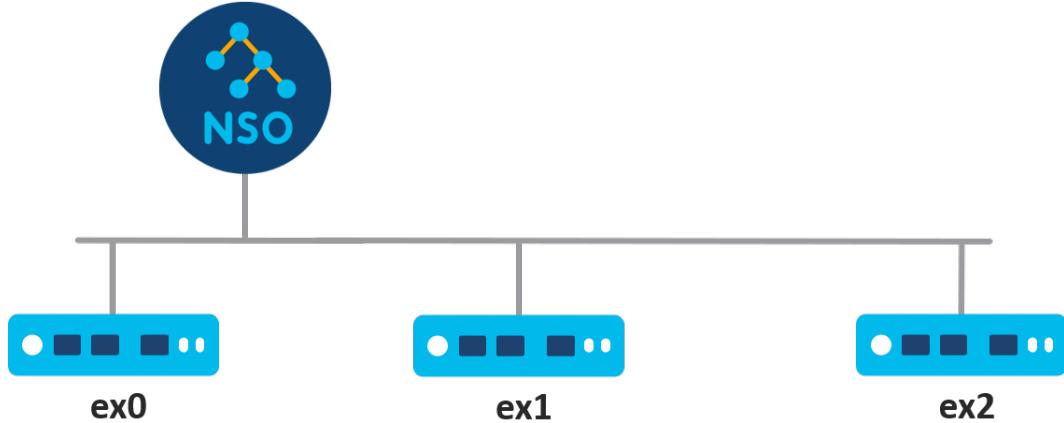
Prerequisites:

- 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 7. 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.

Step 4: Run and verify the script

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.

```
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()
```

Alternatively, you can use the `dry-run` parameter with the `apply_params()` to, for example, preview what will be sent to the device.

```
params = t.get_params()
params.dry_run_native()
result = t.apply_params(True, params)
print(result['device']['ex1'])
t.apply_params(True, t.get_params())
```

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!

A Note on Robustness

The code in this chapter is intentionally kept simple to demonstrate the core concepts and lacks robustness in error handling. In particular, it is missing the retry mechanism in case of concurrency conflicts as described in [the section called “Handling Conflicts”](#).

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 5

Developing a Simple 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.

This chapter introduces the main service building blocks and shows you how to build one yourself.

- [Why services?, page 27](#)
- [The Service Package, page 28](#)
- [Service Parameters, page 30](#)
- [Showcase: A Simple DNS Configuration Service, page 31](#)
- [Service Templates, page 35](#)
- [Showcase: DNS Configuration Service with Templates, page 37](#)

Why services?

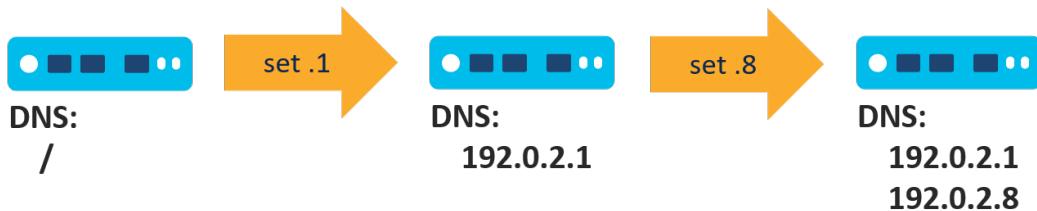
Network automation includes provisioning and deprovisioning configuration, even though the deprovisioning part often doesn't get as much attention. It is nevertheless significant since leftover, residual configuration can cause hard-to-diagnose operational problems. Even more importantly, without proper 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 8. 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.

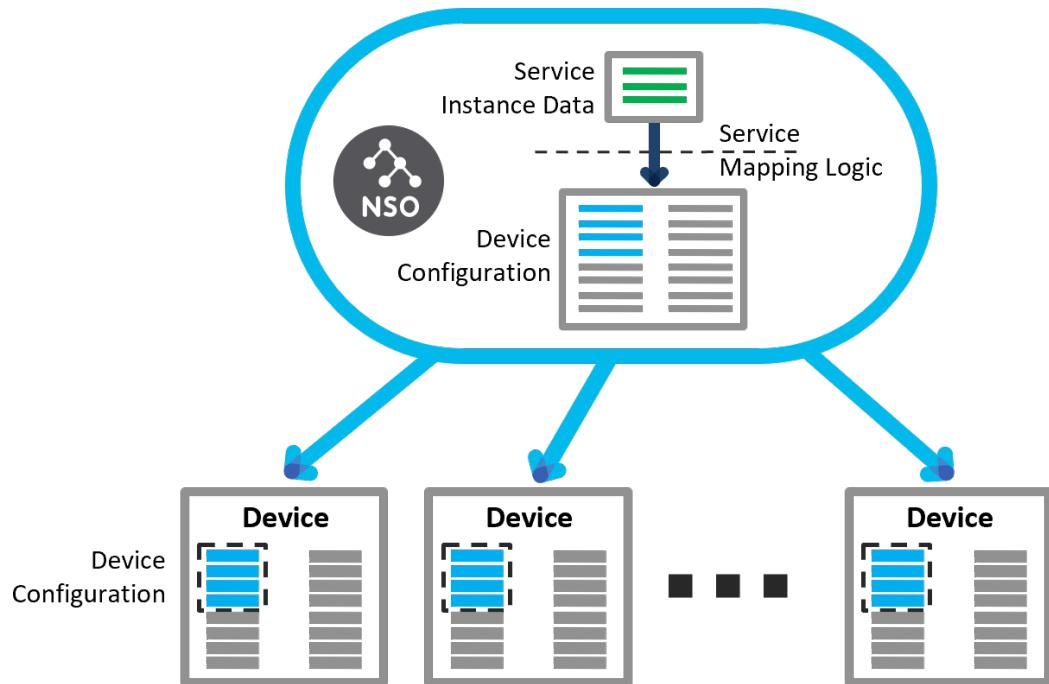
The package contains the two most important parts of the service:

- 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 9. 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 will not 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 is 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 is the **test** folder where you can put automated testing scripts, which will not 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
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. Also note that there is no need to check if the entry for this DNS server already exists in the list.

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
    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
```

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
                            +
                                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/
```

```
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 6

Applications in NSO

Services provide the foundation for managing the configuration of a network. But this is not the only aspect of network automation. A holistic solution must also consider various verification procedures, one-time actions, monitoring, and so on. This is quite different from managing configuration. NSO helps you implement such automation use-cases through a generic application framework.

This chapter explores the concept of services as more general NSO applications. It gives an overview of the mechanisms for orchestrating network automation tasks that require more than just configuration provisioning.

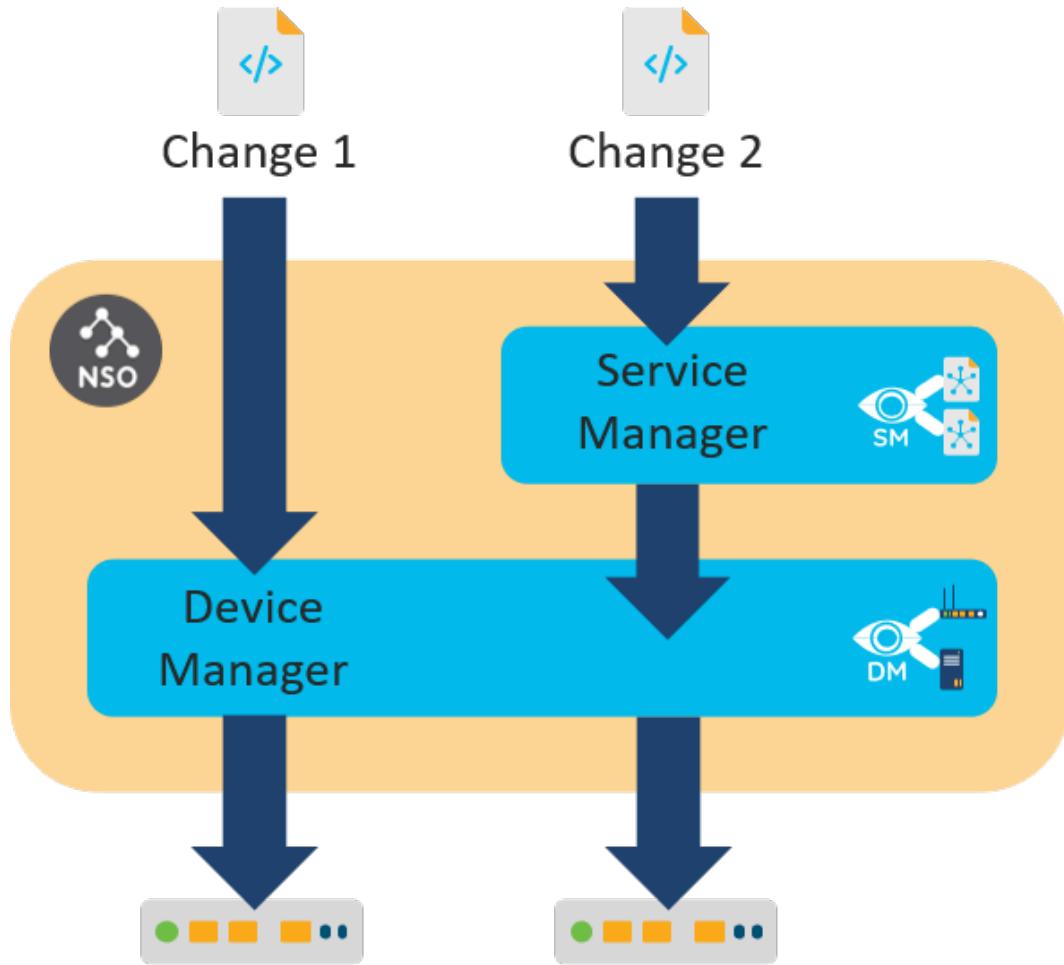
- [NSO Architecture, page 41](#)
- [Callbacks as Extension Mechanism, page 42](#)
- [Actions, page 44](#)
- [Showcase: Implementing Device Count Action, page 45](#)
- [Overview of Extension Points, page 49](#)
- [Monitoring for Change, page 50](#)
- [Running Application Code, page 50](#)
- [Application Updates, page 51](#)

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.

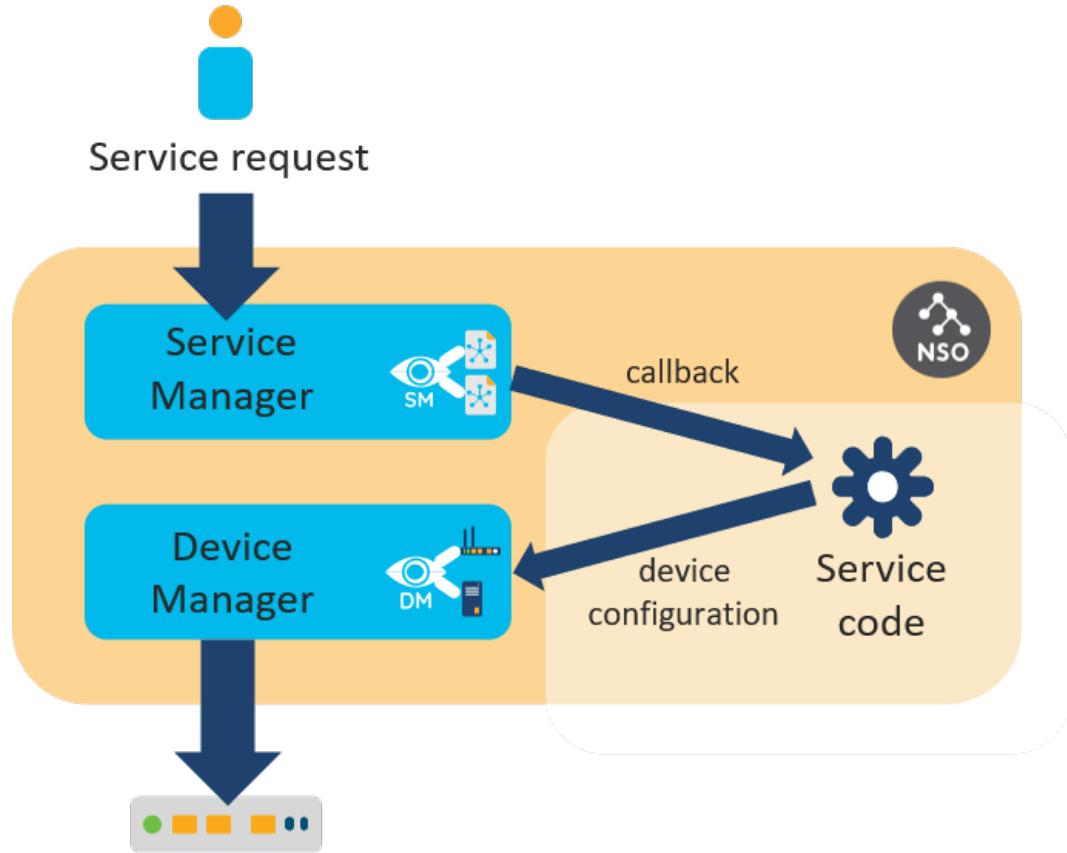
Figure 10. Device and Service Manager

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. For example, an integrated service test action can verify the initial provisioning and simplify troubleshooting if issues arise. A simple test might run the `ping` command to verify connectivity. Or an application could only monitor the network and not produce any configuration at all. That is why NSO actually uses an approach where an application chooses what custom code to execute 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 11. 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. See [the section called “Service Callbacks”](#) for 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. In [the section called “Overview of Extension Points”](#) you will find 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 elements, 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.



Note If, for any reason, your action requires a new, read-write transaction, please also read through [Chapter 24, NSO Concurrency Model](#) to learn about the possible pitfalls.

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 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:

Step 4: Register callback

```

from ipaddress import IPv4Address, IPv4Network
import socket
import ncs
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**.

Overview of Extension Points

NSO supports a number of extension points for custom callbacks.

- **Type:** Service
Supported in: Python, Java, Erlang

YANG extension: ncs:servicepoint

Transforms a list or container into a model for service instances. When configuration of a service instance changes, NSO invokes Service Manager and FASTMAP, which may call service *create* and similar callbacks. See [Chapter 5, *Developing a Simple Service*](#) for an introduction.

- **Type:** Action
Supported in: Python, Java, Erlang

YANG extension: tailf:actionpoint

Defines callbacks when an action or RPC is invoked. See [the section called “Actions”](#) for an introduction.

- **Type:** Validation
Supported in: Python, Java, Erlang

YANG extension: tailf:validate

Defines callbacks for additional validation of data when the provided YANG functionality, such as `must` and `unique` statements, is insufficient. See the respective API documentation for examples; the section called “ValidationPoint handler” (Python), the section called “Validation Callbacks” (Java) and Chapter 12, *Embedded Erlang applications* (Erlang).

- **Type:** Data Provider

Supported in: Java, Python (low-level API with experimental high-level API), Erlang

YANG extension: tailf:callpoint

Defines callbacks for transparently accessing external data (data not stored in the CDB) or callbacks for special processing of data nodes (transforms, set and transaction hooks). Requires careful implementation and understanding of transaction intricacies. Rarely used in NSO.

Each extension point in the list has a corresponding YANG extension that defines to which part of the data model the callbacks apply, as well as the individual name of the call point. The name is required during callback registration and helps distinguish between multiple uses of the extension. Each extension generally specifies multiple callbacks, however, you often need to implement only the main one, e.g. `create` for services or `action` for actions.

In addition, NSO supports some specific callbacks from internal systems, such as the transaction or the authorization engine, but these have very narrow use and are in general *not* recommended.

Monitoring for Change

Services and actions are examples of something that happens directly as a result of a user (or other northbound agent) request. That is, a user takes an active role in starting service instantiation or invoking an action. Contrast this to a change that happens in the network and requires the orchestration system to take some action. In this latter case, the system monitors the notifications that the network generates, such as losing a link, and responds to the new data.

NSO provides out-of-the-box support for automation of not only notifications but also changes to the operational and configuration data, using the concept of *kickers*. With kickers, you can watch for a particular change to occur in the system and invoke a custom action that handles the change.

The kicker system is further described in Chapter 28, *Kicker*.

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 section called “IPC ports” in *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. For example, see [the section called “Service Actions”](#) to compare Python and Java code.

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 11, The NSO Python VM](#), [Chapter 10, The NSO Java VM](#), and [Chapter 12, 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.

Application Updates

As your NSO application evolves, you will create newer versions of your application package, which will replace the existing one. If the application becomes sufficiently complex, you might even split it across multiple packages.

When you replace a package, NSO must redeploy the application code and potentially replace the package-provided part of the YANG schema. For the latter, NSO can perform the data migration for you, as long as the schema is backward compatible. This process is documented in [the section called “Automatic Schema Upgrades and Downgrades”](#) and is automatic when you request reload of the package with `packages reload` or a similar command.

If your schema changes are not backward compatible, you can implement a data migration procedure, which NSO invokes when upgrading the schema. Among other things, this allows you to reuse and migrate the data that is no longer present in the new schema. You can specify the migration procedure as part of the `package-meta-data.xml` file, using a component of the upgrade type. See [the section called “The upgrade component”](#) (Python) and `examples.ncs/getting-started/developing-with-ncs/14-upgrade-service` example (Java) for details.

Note that changing the schema in any way requires you to recompile the `.fxs` files in the package, which is typically done by running `make` in the package's `src` folder.

However, if the schema does not change, you can request that only the application code and templates be redeployed by using the `packages package my-pkg redeploy` command.



CHAPTER 7

Implementing Services

Services are the cornerstone of network automation with NSO. A service is not just a reusable recipe for provisioning network configurations; it allows you to manage the full configuration life-cycle with minimal effort.

The following chapter examines in greater detail how services work, how to design them, and the different ways to implement them. For a quicker introduction and a simple showcase, see [Chapter 5, Developing a Simple Service](#).

- [Introduction, page 53](#)
- [Service Mapping, page 54](#)
- [A Template is All You Need, page 56](#)
- [Service Model Captures Inputs, page 59](#)
- [Extracting the Service Parameters, page 62](#)
- [FASTMAP and Service Life Cycle, page 66](#)
- [Templates and Code, page 68](#)
- [Configuring Multiple Devices, page 73](#)
- [Shared Service Settings and Auxiliary Data, page 78](#)
- [Service Actions, page 80](#)
- [Operational Data, page 83](#)
- [Nano Services for Provisioning with Side Effects , page 87](#)
- [Service Troubleshooting, page 88](#)

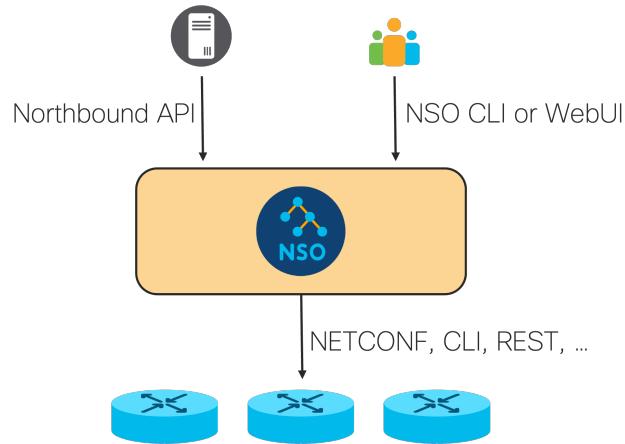
Introduction

In NSO, the term *service* has a special meaning and represents an automation construct that orchestrates create, modify, and delete of a service instance into the resulting native commands to devices in the network. In its simplest form, a service takes some input parameters and maps them to device-specific configurations. It is a recipe, or a set of instructions.

Much like you can bake many cakes using a single cake recipe, you can create many *service instances* using the same service. But unlike cakes, having the recipe produce exactly the same output, is not very useful. That is why service instances define a set of *input parameters*, which the service uses to customize the produced configuration.

A network engineer on the CLI, or an API call from a northbound system, provides the values for input parameters when requesting a new service instance, and NSO uses the “service recipe”, called a *service mapping*, to configure the network.

Figure 12. A high-level view of services in NSO



A similar process takes place when deleting the service instance or modifying the input parameters. The main task of a service is therefore: from a given set of input parameters, calculate the minimal set of device operations to achieve the desired service change. Here, it is very important that the service supports *any* change; create, delete, and update of any service parameter.

Device configuration is usually the primary goal of a service. However, there may be other supporting functions that are expected from the service, such as service-specific actions. The complete *service application*, implementing all the service functionality, is packaged in an NSO *service package*.

The following definitions are used throughout this section:

<i>Service type</i>	Often referred to simply as a service, denotes a specific type of service, such as "L2 VPN", "L3 VPN", "Firewall", or "DNS".
<i>Service instance</i>	A specific instance of a service type, such as "L3 VPN for ACME" or "Firewall for user X".
<i>Service model</i>	The schema definition for a service type, defined in YANG. It specifies the names and format of input parameters for the service.
<i>Service mapping</i>	The instructions that implement a service by mapping the input parameters for a service instance to device configuration.
<i>Device configuration</i>	Network devices are configured to perform network functions. A service instance results in corresponding device configuration changes.
<i>Service application</i>	The code and models implementing the complete service functionality, including service mapping, actions, models for auxiliary data, and so on.

Service Mapping

Developing a service that transforms a service instance request to the relevant device configurations is done differently in NSO than in most other tools on the market. As a service developer, you create a mapping from a YANG service model to the corresponding device YANG model.

This is a declarative, model-to-model mapping. Irrespective of the underlying device type and its native device interface, the mapping is towards a YANG device model and not the native CLI (or any other

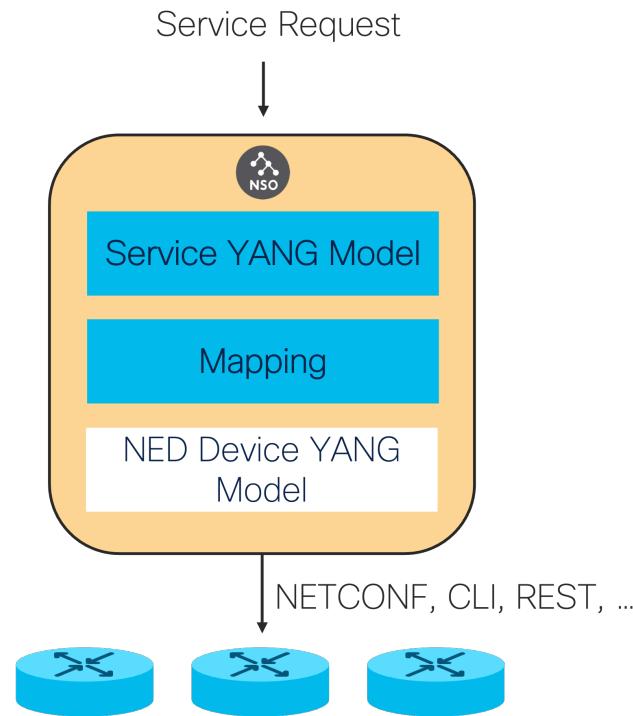
protocol/API). As you write the service mapping, you do not have to worry about the syntax of different CLI commands or in which order these commands are sent to the device. It is all taken care of by the NSO device manager and device NEDs. Implementing a service in NSO is reduced to transforming the 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.

A service application then has two primary artifacts: a YANG service model and a mapping definition to the device YANG, as illustrated in the following figure.

Figure 13. Service model and mapping



To reiterate:

- The mapping is not defined using workflows, or sequences of device commands.
- The mapping is not defined in the native device interface language.

This approach may seem somewhat unorthodox at first but allows NSO to streamline and greatly simplify how you implement services.

A common problem for traditional automation systems is that a set of instructions needs to be defined for every possible service instance change. Take for example a VPN service. During a service life cycle you want to:

- Create the initial VPN

- Add a new site or leg to the VPN
- Remove a site or leg from the VPN
- Modify the parameters of a VPN leg, such as the IP addresses used
- Change the interface used for the VPN on a device
- ...
- Delete the VPN

The possible run-time changes for an existing service instance are numerous. If a developer must define instructions for every possible change, such as 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 renders the minimum resulting change for any possible change in the service instance. It achieves this with the FASTMAP algorithm.

Another challenge in traditional systems is that a lot of code goes into managing error scenarios. The NSO built-in transaction manager takes that burden away from the developer of the service application by providing automatic rollback of incomplete changes.

Another benefit of this approach is that NSO can automatically generate the northbound APIs and database schema from the YANG models, enabling a true 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, and all northbound APIs and User Interfaces are automatically regenerated to reflect the new or updated models.

A Template is All You Need

To demonstrate the simplicity a pure model-to-model service mapping affords, let us consider the most basic approach to providing the mapping: the service XML template. The XML template is an XML-encoded file that tells NSO what configuration to generate when someone requests a new service instance.

The first thing you need is the relevant device configuration (or configurations if multiple devices are involved). Suppose you must configure 192.0.2.1 as a DNS server on the target device. Using the NSO CLI, you first enter the device configuration, then add the DNS server. For a Cisco IOS-based device:

```
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# devices device c1 config
admin@ncs(config-config)# ip name-server 192.0.2.1
admin@ncs(config-config)# top
admin@ncs(config)#
```

Note here that the configuration is not yet committed and you can use the **commit dry-run outformat xml** command to produce the configuration in the XML format. This format is an excellent starting point for creating an XML template.

```
admin@ncs(config)# commit dry-run outformat xml

result-xml {
    local-node {
        data <devices xmlns="http://tail-f.com/ns/ncs">
            <device>
                <name>c1</name>
                <config>
                    <ip xmlns="urn:ios">
                        <name-server>192.0.2.1</name-server>
                    </ip>
```

```

        </config>
    </device>
</devices>
}
}

```

The interesting portion is the part between `<devices>` and `</devices>` tags.

Another way to get the XML output is to list the existing device configuration in NSO by piping it through the **display xml** filter:

```

admin@ncs# show running-config devices device c1 config ip name-server | display xml

<config xmlns="http://tail-f.com/ns/config/1.0">
    <devices xmlns="http://tail-f.com/ns/ncs">
        <device>
            <name>c1</name>
            <config>
                <ip xmlns="urn:ios">
                    <name-server>192.0.2.1</name-server>
                </ip>
            </config>
        </device>
    </devices>
</config>

```

If there is a lot of data, it is easy to save the output to a file using the **save** pipe in the CLI, instead of copying and pasting it by hand:

```

admin@ncs# show running-config devices device c1 config ip name-server | display xml \
| save dns-template.xml

```

The last command saves the configuration for a device in the `dns-template.xml` file using XML format. To use it in a service, you need a *service package*.

You create an empty, skeleton service with the **ncs-make-package** command, such as:

```
ncs-make-package --build --no-test --service-skeleton template dns
```

The command generates the minimal files necessary for a service package, here named `dns`. One of the files is `dns/templates/dns-template.xml`, which is where the configuration in the XML format goes.

```

<config-template xmlns="http://tail-f.com/ns/config/1.0"
    servicepoint="dns">
    <devices xmlns="http://tail-f.com/ns/ncs">
        <!-- ... more statements here ... -->
    </devices>
</config-template>

```

If you look closely, there is one significant difference from the **show running-config** output: the template uses the `config-template` XML root tag, instead of `config`. This tag also has the `servicepoint` attribute. Other than that, you can use the XML formatted configuration from the CLI as-is.

Bringing the two XML documents together, gives the final `dns/templates/dns-template.xml` XML template:

Example 14. Static DNS configuration template

```

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

```

```

<device>
  <name>c1</name>
  <config>
    <ip xmlns="urn:ios">
      <name-server>192.0.2.1</name-server>
    </ip>
  </config>
</device>
</devices>
</config-template>

```

The service is now ready to use in NSO. Start the examples.ncs/implement-a-service/dns-v1 example to set up a live NSO system with such a service and inspect how it works. Try configuring two different instances of the dns service.

```
$ cd $NCS_DIR/examples.ncs/implement-a-service/dns-v1
$ make demo
```

The problem with this service is that it always does the same thing because it always generates exactly the same configuration. It would be much better if the service could configure different devices. The updated version, v1.1, uses a slightly modified template:

```

<config-template xmlns="http://tail-f.com/ns/config/1.0"
  servicepoint="dns">
<devices xmlns="http://tail-f.com/ns/ncs">
  <device>
    <name>{/name}</name>
    <config>
      <ip xmlns="urn:ios">
        <name-server>192.0.2.1</name-server>
      </ip>
    </config>
  </device>
</devices>
</config-template>

```

The changed part is `<name>{/name}</name>`, which now uses the `{ /name }` code instead of a hard-coded c1 value. The curly braces indicate that NSO should evaluate the enclosed expression and use the resulting value in its place. The `/name` expression is an XPath expression, referencing the service YANG model. In the model, name is the name you give each service instance. In this case, the instance name doubles for identifying the target device.

```

admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# dns c2
admin@ncs(config-dns-c2)# commit dry-run

cli {
  local-node {
    data devices {
      device c2 {
        config {
          ip {
            +
            name-server 192.0.2.1;
          }
        }
      }
    }
  }
}

```

```
}
```

In the output, the instance name used was c2 and that is why the service performs DNS configuration for the c2 device.

The template actually allows a decent amount of programmability through XPath and special XML processing instructions. For example:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0"
                  servicepoint="dns">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>{/name}</name>
      <config>
        <ip xmlns="urn:ios">
          <?if {starts-with(/name, 'c1')?}>
            <name-server>192.0.2.1</name-server>
          <?else?>
            <name-server>192.0.2.2</name-server>
          <?end?>
        </ip>
      </config>
    </device>
  </devices>
</config-template>
```

In the preceding printout, the XPath `starts-with()` function is used to check if the device name starts with a specific prefix. Then one set of configuration items is used, and a different one otherwise. For additional available instructions and the complete set of template features, see [Chapter 8, Templates](#).

However, most provisioning tasks require some kind of input to be useful. Fortunately, you can define any number of input parameters in the service model that you can then reference from the template; either to use directly in the configuration or as something to base provisioning decisions on.

Service Model Captures Inputs

The YANG service model specifies the input parameters a service in NSO takes. 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.

Even a service as simple as the DNS configuration service usually needs some parameters, such as the target device. The service model gives each parameter a name and defines validation rules, ensuring the client-provided values fit what the service expects.

Suppose you want to add a parameter for the target device to the simple DNS configuration service. You need to construct an appropriate service model, adding a YANG leaf to capture this input.



Note

This task requires some basic YANG knowledge. Review the section called “[Data Modeling Basics](#)” for a primer on the main building blocks of the YANG language.

The service model is located in the `src/yang/servicename.yang` file in the package. It typically resembles the following structure:

```
list servicename {
  key name;

  uses ncs:service-data;
```

```

ncs:servicepoint "servicename";

leaf name {
    type string;
}

// ... other statements ...
}

```

The list named after the package (*servicename* in the example) is the interesting part.

The uses ncs:service-data and ncs:servicepoint statements differentiate this list from any standard YANG list and make it a service. Each list item in NSO represents a service instance of this type.

The uses ncs:service-data part allows the system to store internal state and provide common service actions, such as **re-deploy** and **get-modifications** for each service instance.

The ncs:servicepoint identifies which part of the system is responsible for the service mapping. For a template-only service, it is the XML template which uses the same service point value in the config-template element.

The name leaf serves as the key of the list and is primarily used to distinguish service instances from each other.

The remaining statements describe the functionality and input parameters that are specific to this service. This is where you add the new leaf for the target device parameter of the dns service:

```

list dns {
    key name;

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

    leaf name {
        type string;
    }

    leaf target-device {
        type string;
    }
}

```

Use the examples.ncs/implement-a-service/dns-v2 example to explore how this model works and try to discover what deficiencies it may have.

```
$ cd $NCS_DIR/examples.ncs/implement-a-service/dns-v2
$ make demo
```

In its current form, the model allows you to specify any value for target-device, including none at all! Obviously, this is not good as it breaks the provisioning of the service. But even more importantly, not validating the input may allow someone to use the service in the way you have not intended and perhaps bring down the network.

You can guard against invalid input with the help of additional YANG statements. For example:

```

leaf target-device {
    mandatory true;
    type string {
        length "2";
        pattern "c[0-2]";
    }
}

```

Now this parameter is mandatory for every service instance and must be one of the string literals: c0, c1, or c2. This format is defined by the regular expression in the `pattern` statement. In this particular case, the `length` restriction is redundant but demonstrates how you can combine multiple restrictions. You can even add multiple `pattern` statements to handle more complex cases.

What if you wanted to make the DNS server address configurable too? You can add another leaf to the service model:

```
leaf dns-server-ip {
    type inet:ipv4-address {
        pattern "192\\\\.0\\\\.2\\\\.\\..*";
    }
}
```

There are three notable things about this leaf:

- There is no `mandatory` statement, meaning the value for this leaf is optional. The XML template will be designed to provide some default value if none is given.
- The type of the leaf is `inet:ipv4-address`, which restricts the value for this leaf to an IP address.
- The `inet:ipv4-address` type is further restricted using a regular expression to only allow IP addresses from the 192.0.2.0/24 range.

YANG is very powerful and allows you to model all kinds of values and restrictions on the data. In addition to the ones defined in the YANG language ([RFC 7950, section 9](#)), predefined types describing common networking concepts, such as those from the `inet` namespace ([RFC 6991](#)), are available to you out of the box. It is much easier to validate the inputs when so many options are supported.

The one missing piece for the service is the XML template. You can take [Example 14, “Static DNS configuration template”](#) as a base and tweak it to reference the defined inputs.

Using the code `{XYZ}` or `{/XYZ}` in the template, instructs NSO to look for the value in the service instance data, in the node with the name `XYZ`. So, you can refer to the target-device input parameter as defined in YANG with the `{/target-device}` code in the XML template.



Note

The code inside the curly brackets actually contains an XPath 1.0 expression with the service instance data as its root, so an absolute path (with a slash) and a relative one (without it) refer to the same node in this case, and you can use either.

The final, improved version of the DNS service template that takes into account the new model, is:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0"
                  servicepoint="dns">
    <devices xmlns="http://tail-f.com/ns/ncs">
        <device>
            <name>{/target-device}</name>
            <config>
                <ip xmlns="urn:ios">
                    <?if {/dns-server-ip}?>
                        <!-- If dns-server-ip is set, use that. -->
                        <name-server>{/dns-server-ip}</name-server>
                    <?else?>
                        <!-- Otherwise, use the default one. -->
                        <name-server>192.0.2.1</name-server>
                    <?end?>
                </ip>
```

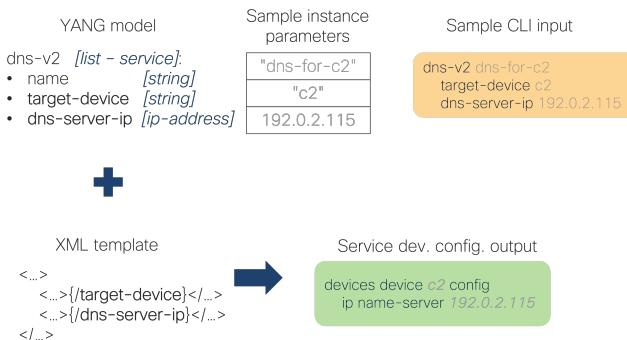
```

</config>
</device>
</devices>
</config-template>

```

The following figure captures the relationship between the YANG model and the XML template that ultimately produces the desired device configuration.

Figure 15. XML template and model relationship



The complete service is available in the `examples.ncs/implement-a-service/dns-v2.1` example. Feel free to investigate on your own how it differs from the initial, no-validation service.

```
$ cd $NCS_DIR/examples.ncs/implement-a-service/dns-v2.1
$ make demo
```

Extracting the Service Parameters

When the service is simple, constructing the YANG model and creating the service mapping (the XML template) is straightforward. Since the two components are mostly independent, you can start your service design with either one.

If you write the YANG model first, you can load it as a service package into NSO (without having any mapping defined) and iterate on it. This way, you can try the model, which is the interface to the service, with network engineers or northbound systems before investing the time to create the mapping. This model-first approach is also sometimes called top-down.

The alternative is to create the mapping first. Especially for developers new to NSO, the template-first, or bottom-up, approach is often easier to implement. With this approach, you templatize the configuration and extract the required service parameters from the template.

Experienced NSO developers naturally combine the two approaches, without much thinking. However, if you have trouble modeling your service at first, consider following the template-first approach demonstrated here.

For the following example, suppose you want the service to configure IP addressing on an ethernet interface. You know what configuration is required to do this manually for a particular ethernet interface. For a Cisco IOS-based device you would use the commands, such as:

```
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# devices device c1 config
admin@ncs(config-config)# interface GigabitEthernet 0/0
admin@ncs(config-if)# ip address 192.168.5.1 255.255.255.0
```

To transform this configuration into a reusable service, complete the following steps:

- Create an XML template with hard-coded values.
- Replace each value specific to this instance with a parameter reference.
- Add each parameter to the YANG model.
- Add parameter validation.
- Consolidate and clean up the YANG model as necessary.

Start by generating the configuration in the XML format, making use of the **display xml** filter. Note that the XML output will not necessarily be a one-to-one mapping of the CLI commands; the XML reflects the device YANG model which can be more complex but the commands on the CLI can hide some of this complexity.

The transformation to a template also requires you to change the root tag, which produces the resulting XML template:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0"
                  servicepoint="iface-servicepoint">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>c1</name>
      <config>
        <interface xmlns="urn:ios">
          <GigabitEthernet>
            <name>0/0</name>
            <ip>
              <address>
                <primary>
                  <address>192.168.5.1</address>
                  <mask>255.255.255.0</mask>
                </primary>
              </address>
            </ip>
          </GigabitEthernet>
        </interface>
      </config>
    </device>
  </devices>
</config-template>
```

However, this template has all the values hard-coded and only configures one specific interface on one specific device.

Now you must replace all the dynamic parts that vary from service instance to service instance with references to the relevant parameters. In this case, it is data specific to each device: which interface and which IP address to use.

Suppose you pick the following names for the variable parameters:

- 1 **device**: The network device to configure.
- 2 **interface**: The network interface on the selected device.
- 3 **ip-address**: The IP address to use on the selected interface.

Generally, you can make up any name for a parameter but it is best to follow the same rules that apply for naming variables in programming languages, such as making the name descriptive but not excessively verbose. It is customary to use hyphen (minus sign) to concatenate words and use all-lowercase (“kebab-case”), which is the convention used in the YANG language standards.

Figure 16. Making a configuration template

The corresponding template then becomes:

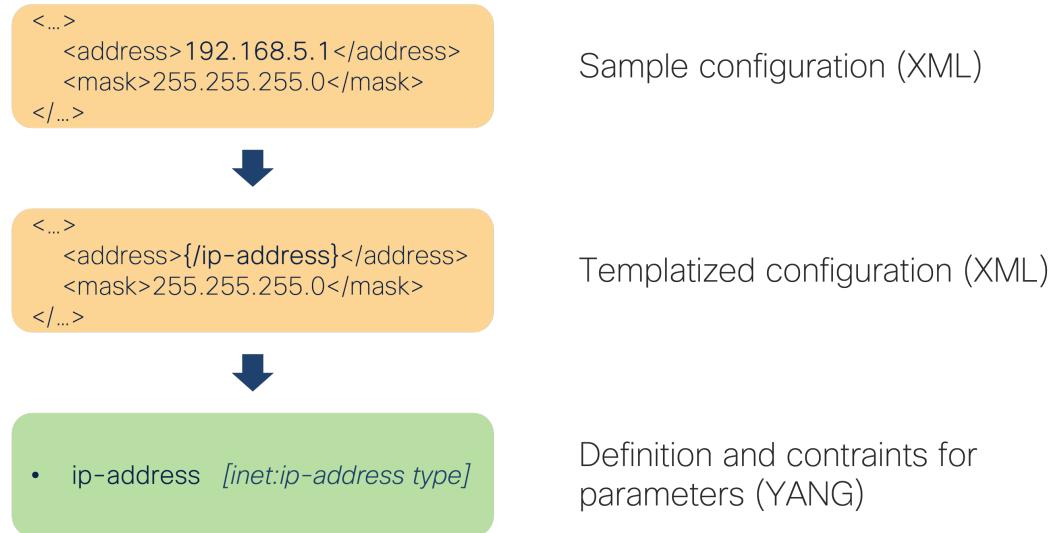
```

<config-template xmlns="http://tail-f.com/ns/config/1.0"
                  servicepoint="iface-servicepoint">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>{/device}</name>
      <config>
        <interface xmlns="urn:ios">
          <GigabitEthernet>
            <name>{/interface}</name>
            <ip>
              <address>
                <primary>
                  <address>{/ip-address}</address>
                  <mask>255.255.255.0</mask>
                </primary>
              </address>
            </ip>
          </GigabitEthernet>
        </interface>
      </config>
    </device>
  </devices>
</config-template>

```

Having completed the template, you can add all the parameters, three in this case, to the service model.

Figure 17. Extracting service model from template in a bottom-up approach



The partially-completed model is now:

```
list iface {
    key name;

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

    leaf name {
        type string;
    }

    leaf device { ... }

    leaf interface { ... }

    leaf ip-address { ... }
}
```

Missing are the data type and other validation statements. At this point, you could fill out the model with generic `type string` statements, akin to the `name` leaf. This is a useful technique to test out the service in early development. But here you can complete the model directly, as it contains only three parameters.

You can use a `leafref` type leaf to refer to a device by its name in the NSO. This type uses dynamic lookup at the specified path to enumerate the available values. For the `device` leaf, it lists every value for a device name that NSO knows about. If there are two devices managed by NSO, named `rtr-sjc-01` and `rtr-sto-01`, either “`rtr-sjc-01`” or “`rtr-sto-01`” are valid values for such a leaf. This is a common way to refer to devices in NSO services.

```
leaf device {
    mandatory true;
    type leafref {
        path "/ncs:devices/ncs:device/ncs:name";
    }
}
```

In a similar fashion restrict the valid values of other two parameters.

```

leaf interface {
    mandatory true;
    type string {
        pattern "[0-9]/[0-9]+";
    }
}

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

```

You would typically create the service package skeleton with the **ncs-make-package** command and update the model in the .yang file. The model in the skeleton might have some additional example leafs that you do not need and should remove to finalize the model. That gives you the final, full service model:

```

list iface {
    key name;

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

    leaf name {
        type string;
    }

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

    leaf interface {
        mandatory true;
        type string {
            pattern "[0-9]/[0-9]+";
        }
    }

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

```

The examples.ncs/implement-a-service/iface-v1 example contains the complete YANG module with this service model in the packages/iface-v1/src/yang/iface.yang file, as well as the corresponding service template in packages/iface-v1/templates/iface-template.xml.

FASTMAP and Service Life Cycle

The YANG model and the mapping (the XML template) are the two main components required to implement a service in NSO. The hidden part of the system that makes such an approach feasible is called FASTMAP.

FASTMAP covers the complete service life cycle: creating, changing and deleting the service. It requires the minimal amount of code for mapping from a service model to a device model.

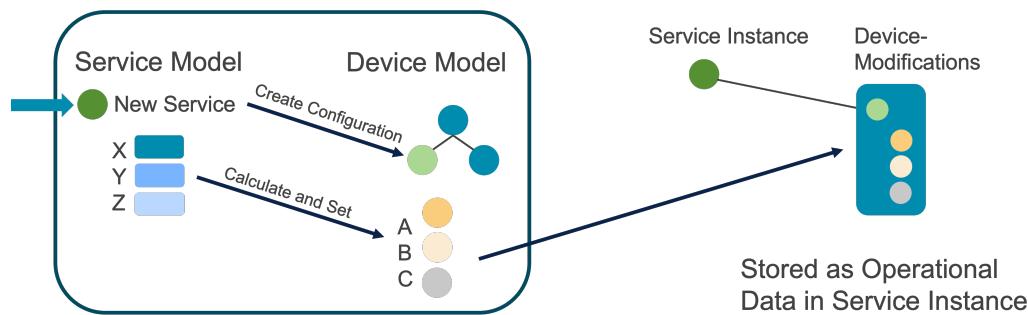
FASTMAP is based on generating changes from an initial create operation. When the service instance is created the reverse of the resulting device configuration is stored together with the service instance. If an NSO user later changes the service instance, NSO first applies (in an isolated transaction) the reverse diff of the service, effectively undoing the previous create operation. Then it runs the logic to create the service again, and finally performs a diff against the current configuration. Only the result of the diff is then sent to the affected devices.

**Important**

It is therefore 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 Opaque Data](#)” 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 on the devices.

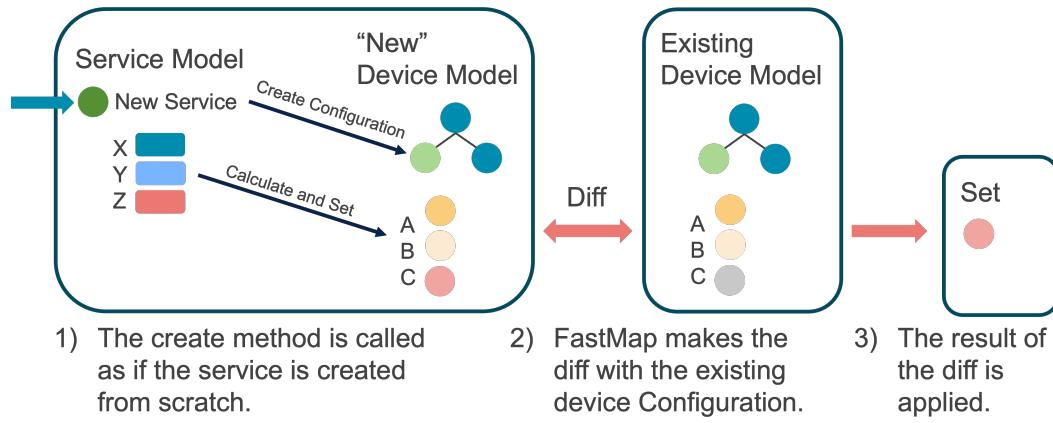
Figure 18. FASTMAP Create a Service



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 must be set on the devices. When the service is instantiated, the previous values of the corresponding device attributes A, B, and C are stored with the service instance in the CDB. This allows NSO to bring the network back to the state before the service was instantiated.

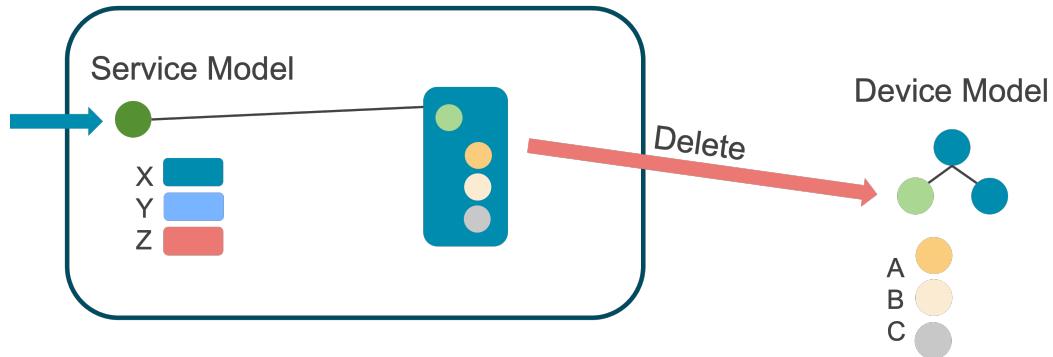
Now let us see what happens if one service attribute is changed. Perhaps the service attribute Z is changed. NSO will execute the mapping as if the service was created from scratch. The resulting device configurations are then compared with the actual configuration and the minimal diff is sent to the devices. Note that this is managed automatically, there is no code to handle the specific "change Z" operation.

Figure 19. FASTMAP Change a Service



When a user deletes a service instance, NSO retrieves the stored device configuration from the moment before the service was created and reverts to it.

Figure 20. FASTMAP Delete a Service



Templates and Code

For a complex service you may realize that the input parameters for a service are not sufficient to render the device configuration. Perhaps the northbound system only provides a subset of the required parameters. For example, the other system wants NSO to pick an IP address and does not pass it as an input parameter. Then additional logic or API calls may be necessary but XML templates provide no such functionality on their own.

The solution is to augment XML templates with custom code. Or, more accurately, create custom provisioning code that leverages XML templates. Alternatively, you can also implement the mapping logic completely in the code and not use templates at all. The latter, forgoing the templates altogether, is less common, since templates have a number of beneficial properties.

Templates separate the way parameters are applied, which depends on the type of target device, from calculating the parameter values. For example, you would use the same code to find the IP address to apply on a device, but the actual configuration might differ whether it is a Cisco IOS (XE) device, an IOS XR, or another vendor entirely.

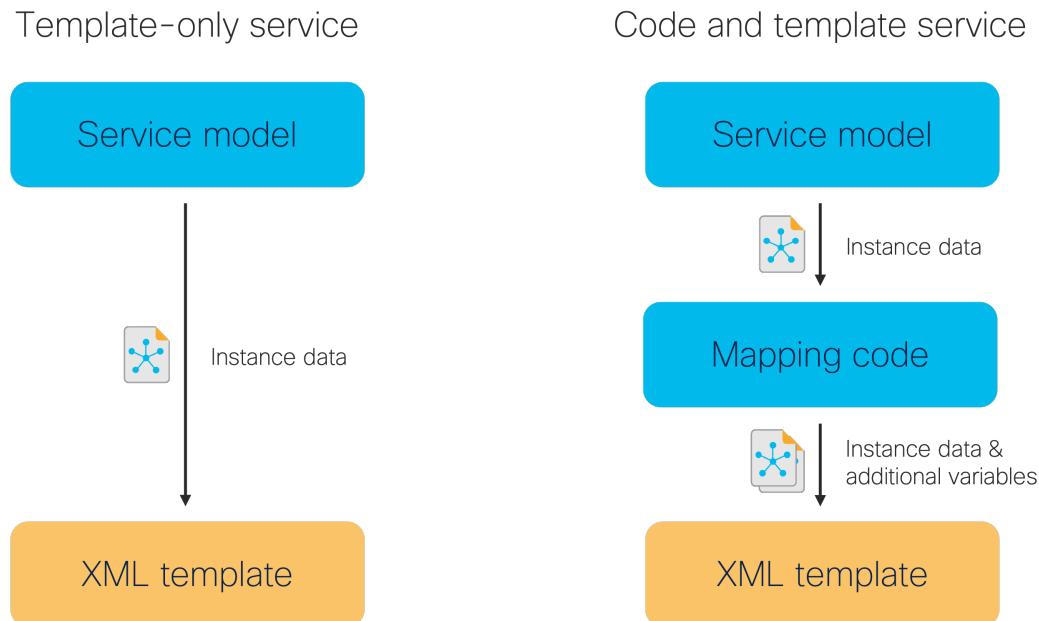
Moreover, if you use templates, NSO can automatically validate the templates being compatible with the used NEDs, which allows you to sidestep whole groups of bugs.

NSO offers multiple programming languages to implement the code. The `--service-skeleton` option of the **ncs-make-package** command influences the selection of the programming language and if the generated code should contain sample calls for applying an XML template.

Suppose you want to extend the template-based ethernet interface addressing service to also allow specifying the netmask. You would like to do this in the more modern, CIDR-based single number format, such as is used in the 192.168.5.1/24 format (the /24 after the address). However, the generated device configuration takes the netmask in the dot-decimal format, such as 255.255.255.0, so the service needs to perform some translation. And that requires custom service code.

Such a service will ultimately contain three parts: the service YANG model, translation code, and the XML template. The model and the template serve the same purpose as before, while custom code provides fine-grained control over how templates are applied and the data available to them.

Figure 21. Code and template service compared to template-only service



Since the service is based on the previous interface addressing service, you can save yourself a lot of work by starting with the existing YANG model and XML template.

The service YANG model needs an additional `cidr-netmask` leaf to hold the user-provided netmask value:

```

list iface {
    key name;

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

    leaf name {
        type string;
    }

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

    leaf interface {
        mandatory true;
        type string {
            pattern "[0-9]/[0-9]+";
        }
    }

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

    leaf cidr-netmask {
    }
}

```

```
    default 24;  
    type uint8 {  
        range "0..32";  
    }  
}
```

This leaf stores a small number (of `uint8` type), with values between 0 and 32. It also specifies a default of 24, which is used when the client does not supply a value for this parameter.

The previous XML template also requires only minor tweaks. A small but important change is the removal of the `servicepoint` attribute on the top element. Since it is gone, NSO does not apply the template directly for each service instance. Instead, your custom code registers itself on this `servicepoint` and is responsible for applying the template.

The reason for it being this way, is that the code will supply the value for the additional variable, here called NETMASK. This is the other change that is necessary in the template: referencing the NETMASK variable for the netmask value:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>{/device}</name>
      <config>
        <interface xmlns="urn:ios">
          <GigabitEthernet>
            <name>{/interface}</name>
            <ip>
              <address>
                <primary>
                  <address>{/ip-address}</address>
                  <mask>{$NETMASK}</mask>
                </primary>
              </address>
            </ip>
          </GigabitEthernet>
        </interface>
      </config>
    </device>
  </devices>
</config-template>
```

Unlike references to other parameters, NETMASK does not represent a data path but a variable. It must start with a dollar character ("\$") to distinguish it from a path. As shown here, variables are often written in all-uppercase, making it easier to quickly tell whether something is a variable or a data path.

Variables get their values from different sources but the most common one is the service code. You implement the service code using a programming language, such as Java or Python.

The following two procedures create an equivalent service that acts identically from a user's perspective. They only differ in the language used; they use the same logic and the same concepts. Still, the final code differs quite a bit due to the nature of each programming language. Generally, you should pick one language and stick with it. If you are unsure which one to pick, you may find Python slightly easier to understand because it is less verbose.

Templates and Python Code

The usual way to start working on a new service is to first create a service skeleton with the **ncs-make-package** command. To use Python code for service logic and XML templates for applying configuration, select the `python-and-template` option. For example:

```
ncs-make-package --no-test --service-skeleton python-and-template iface
```

To use the prepared YANG model and XML template, save them into the `iface/src/yang/iface.yang` and `iface/templates/iface-template.xml` files. This is exactly the same as for the template-only service.

What is different, is the presence of the `python/` directory in the package file structure. It contains one or more Python packages (not to be confused with NSO packages) that provide the service code.

The function of interest is the `cb_create()` function, located in the `main.py` file that the package skeleton created. Its purpose is the same as that of the XML template in the template-only service: generate configuration based on the service instance parameters. This code is also called *the create code*.

The create code usually performs the following tasks:

- Read service instance parameters.
- Prepare configuration variables.
- Apply one or more XML templates.

Reading instance parameters is straightforward with the help of the `service` function parameter, using the Maagic API. For example:

```
def cb_create(self, tctx, root, service, proplist):
    cidr_mask = service.cidr_netmask
```

Note that the hyphen in `cidr-netmask` is replaced with the underscore in `service.cidr_netmask` as documented in [Chapter 16, Python API Overview](#).

The way configuration variables are prepared depends on the type of the service. For the interface addressing service with netmask, the netmask must be converted into dot-decimal format:

```
quad_mask = ipaddress.IPv4Network((0, cidr_mask)).netmask
```

The code makes use of the built-in Python `ipaddress` package for conversion.

Finally, the create code applies a template, with only minimal changes to the skeleton generated sample; the names and values for the `vars.add()` function, which are specific to this service.

```
vars = ncs.template.Variables()
vars.add('NETMASK', quad_mask)
template = ncs.template.Template(service)
template.apply('iface-template', vars)
```

If required, your service code can call `vars.add()` multiple times, to add as many variables as the template expects.

The first argument to the `template.apply()` call is the file name of the XML template, without the `.xml` suffix. It allows you to apply multiple, different templates for a single service instance. Separating the configuration into multiple templates based on functionality, called feature templates, is a great practice with bigger, complex configurations.

The complete create code for the service is:

```
def cb_create(self, tctx, root, service, proplist):
    cidr_mask = service.cidr_netmask

    quad_mask = ipaddress.IPv4Network((0, cidr_mask)).netmask
```

```

vars = ncs.template.Variables()
vars.add('NETMASK', quad_mask)
template = ncs.template.Template(service)
template.apply('iface-template', vars)

```

You can test it out in the `examples.ncs/implement-a-service iface-v2-py` example.

Templates and Java Code

The usual way to start working on a new service is to first create a service skeleton with the **ncs-make-package** command. To use Java code for service logic and XML templates for applying configuration, select the `java-and-template` option. For example:

```
ncs-make-package --no-test --service-skeleton java-and-template iface
```

To use the prepared YANG model and XML template, save them into the `iface/src/yang/iface.yang` and `iface/templates/iface-template.xml` files. This is exactly the same as for the template-only service.

What is different, is the presence of the `src/java` directory in the package file structure. It contains a Java package (not to be confused with NSO packages) that provides the service code and build instructions for the `ant` tool to compile the Java code.

The function of interest is the `create()` function, located in the `ifaceRFS.java` file that the package skeleton created. Its purpose is the same as that of the XML template in the template-only service: generate configuration based on the service instance parameters. This code is also called *the create code*.

The create code usually performs the following tasks:

- Read service instance parameters.
- Prepare configuration variables.
- Apply one or more XML templates.

Reading instance parameters is done with the help of the `service` function parameter, using the [the section called “NAVU API”](#). For example:

```

public Properties create(ServiceContext context,
                         NavuNode service,
                         NavuNode ncsRoot,
                         Properties opaque)
                         throws ConfException {

    String cidr_mask_str = service.leaf("cidr-netmask").valueAsString();
    int cidr_mask = Integer.parseInt(cidr_mask_str);
}

```

The way configuration variables are prepared depends on the type of the service. For the interface addressing service with netmask, the netmask must be converted into dot-decimal format:

```

long tmp_mask = 0xffffffffL << (32 - cidr_mask);
String quad_mask =
    ((tmp_mask >> 24) & 0xff) + "." +
    ((tmp_mask >> 16) & 0xff) + "." +
    ((tmp_mask >> 8) & 0xff) + "." +
    ((tmp_mask >> 0) & 0xff);

```

The create code applies a template, with only minimal changes to the skeleton generated sample; the names and values for the `myVars.putQuoted()` function are different, since they are specific to this service.

```
Template myTemplate = new Template(context, "iface-template");
TemplateVariables myVars = new TemplateVariables();
myVars.putQuoted("NETMASK", quad_mask);
myTemplate.apply(service, myVars);
```

If required, your service code can call `myVars.putQuoted()` multiple times, to add as many variables as the template expects.

The second argument to the `Template` constructor is the file name of the XML template, without the `.xml` suffix. It allows you to instantiate and apply multiple, different templates for a single service instance. Separating the configuration into multiple templates based on functionality, called feature templates, is a great practice with bigger, complex configurations.

Finally, you must also return the `opaque` object and handle various exceptions for the function. If exceptions are propagated out of the create code, you should transform them into NSO specific ones first, so the UI can present the user with a meaningful error message.

The complete create code for the service is then:

```
public Properties create(ServiceContext context,
                         NavuNode service,
                         NavuNode ncsRoot,
                         Properties opaque)
                         throws ConfException {

    try {
        String cidr_mask_str = service.leaf("cidr-netmask").valueAsString();
        int cidr_mask = Integer.parseInt(cidr_mask_str);

        long tmp_mask = 0xffffffffL << (32 - cidr_mask);
        String quad_mask = ((tmp_mask >> 24) & 0xff) +
                           "." + ((tmp_mask >> 16) & 0xff) +
                           "." + ((tmp_mask >> 8) & 0xff) +
                           "." + ((tmp_mask) & 0xff);

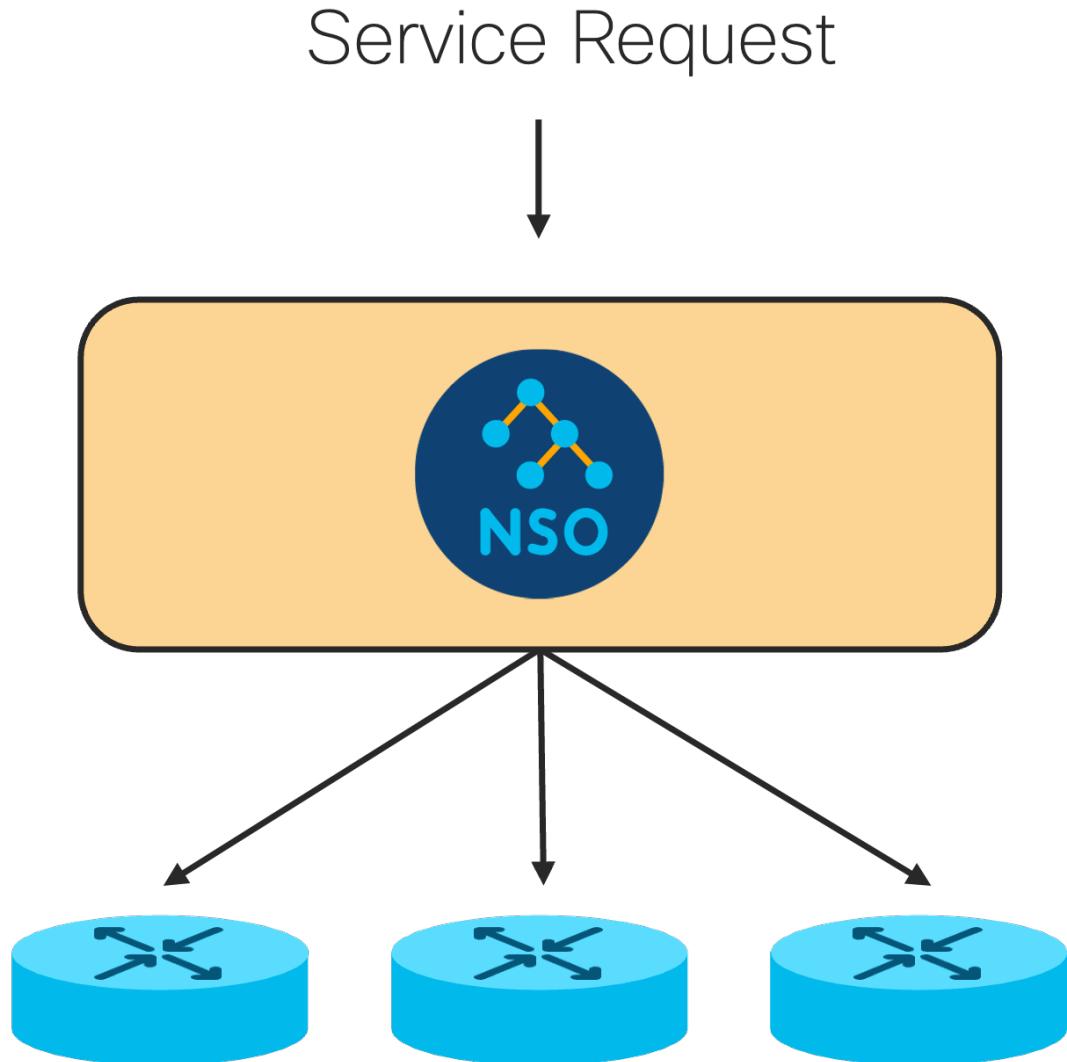
        Template myTemplate = new Template(context, "iface-template");
        TemplateVariables myVars = new TemplateVariables();
        myVars.putQuoted("NETMASK", quad_mask);
        myTemplate.apply(service, myVars);
    } catch (Exception e) {
        throw new DpCallbackException(e.getMessage(), e);
    }
    return opaque;
}
```

You can test it out in the `examples.ncs/implement-a-service iface-v2-java` example.

Configuring Multiple Devices

A service instance may require configuration on more than just a single device. In fact, it is quite common for a service to configure multiple devices.

Figure 22. Service provisioning multiple devices



There are a few ways in which you can achieve this for your services:

- In code: Using API, such as Python Maagic or Java NAVU, navigate the data model to individual device configurations under each devices `device DEVNAME config` and set the required values.
- In code with templates: Apply the template multiple times with different values, such as the device name.
- With templates only: use `foreach` or automatic (implicit) loops.

The generally recommended approach is to use either code with templates or templates with `foreach` loops. They are explicit and also work well when you configure devices of different types. Using only code extends less well to the latter case, as it requires additional logic and checks for each device type.

Automatic, implicit loops in templates are harder to understand, since the syntax looks like the one for normal leafs. A common example is a device definition as a leaf-list in the service YANG model, such as:

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

Because it is a leaf-list, the following template applies to all the selected devices, using an implicit loop:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0"
                  servicepoint="servicename">
    <devices xmlns="http://tail-f.com/ns/ncs">
        <device>
            <name>{/device}</name>
            <config>
                <!-- ... -->
            </config>
        </device>
    </devices>
</config-template>
```

Is performs the same as the one, which loops through the devices explicitly:

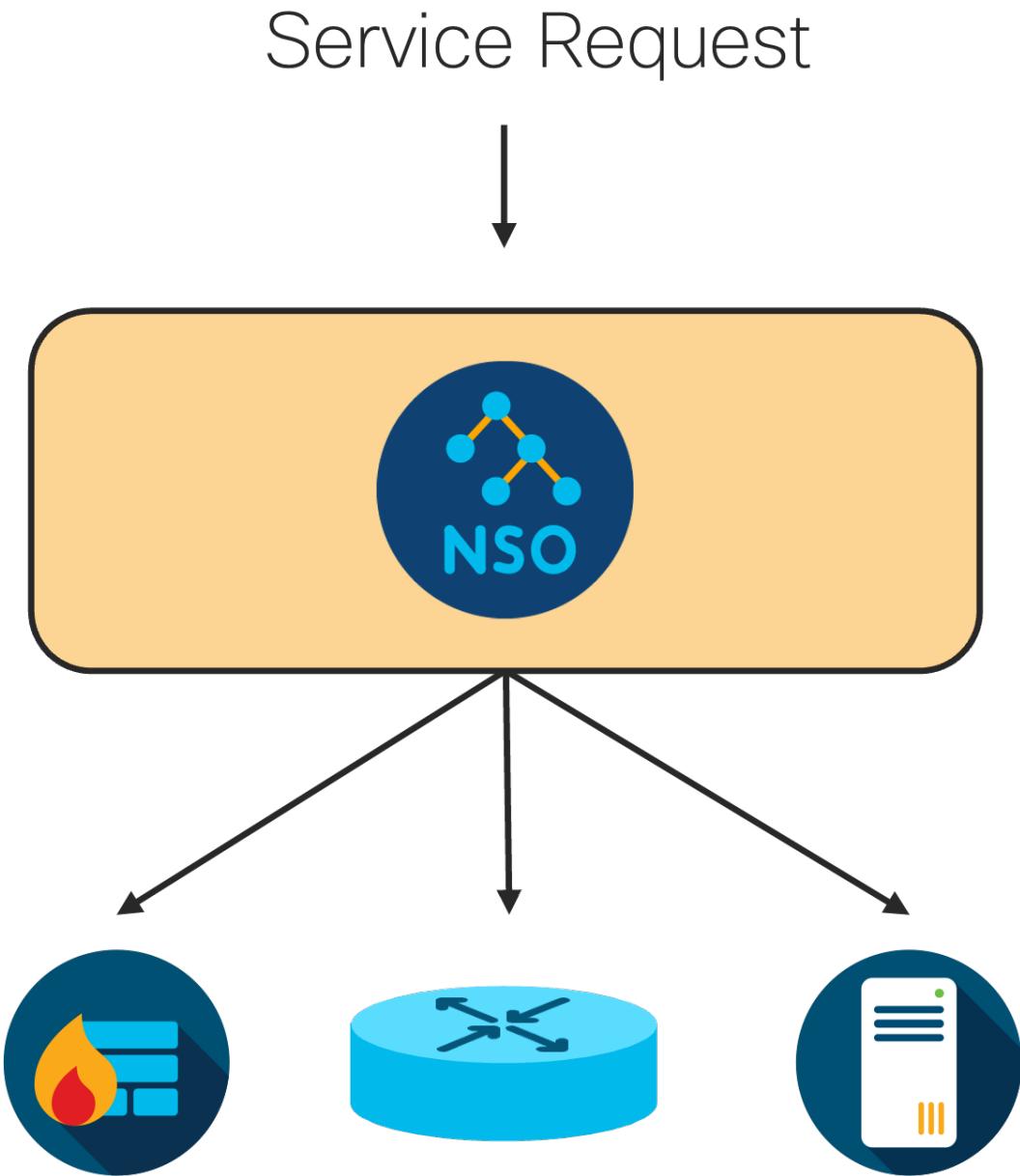
```
<config-template xmlns="http://tail-f.com/ns/config/1.0"
                  servicepoint="servicename">
    <devices xmlns="http://tail-f.com/ns/ncs">
        <?foreach {/device}?>
            <device>
                <name>{.}</name>
                <config>
                    <!-- ... -->
                </config>
            </device>
        <?end?>
    </devices>
</config-template>
```

Being explicit, the latter is usually much easier to understand and maintain for most developers. The examples.ncs/implement-a-service/dns-v3 demonstrates this syntax in the XML template.

Supporting Different Device Types

Applying the same template works fine as long as you have a uniform network with similar devices. What if two different devices can provide the same service but require different configuration? Should you create two different services in NSO? No. Services allow you to abstract and hide the device specifics through device-independent service model, while still allowing customization of device configuration per device type.

Figure 23. Service provisioning multiple device types



One way to do this is to apply a different XML template from service code, depending on the device type. However, the same is also possible through XML templates alone.

When NSO applies configuration elements in the template, it checks the XML namespaces that are used. If the target device does not support a particular namespace, NSO simply skips that part of the template. Consequently, you can put configuration for different device types in the same XML template and only the relevant parts will be applied.

Consider the following example:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0">
```

```

<devices xmlns="http://tail-f.com/ns/ncs">
  <device>
    <name>{/device}</name>
    <config>
      <!-- Part for device with the cisco-ios NED -->
      <interface xmlns="urn:ios">
        <GigabitEthernet>
          <name>{/interface}</name>
          <!-- ... -->
        </GigabitEthernet>
      </interface>

      <!-- Part for device with the router-nc NED -->
      <sys xmlns="http://example.com/router">
        <interfaces>
          <interface>
            <name>{/interface}</name>
            <!-- ... -->
          </interface>
        </interfaces>
      </sys>
    </config>
  </device>
</devices>
</config-template>

```

Due to the `xmlns="urn:ios"` attribute, the first part of the template (the `interface GigabitEthernet`) will only apply to Cisco IOS-based device. While the second part (the `sys interfaces interface`) will only apply to the netsim-based router-nc-type devices, as defined by the `xmlns` attribute on the `sys` element.

In case you need to further limit what configuration applies where and namespace-based filtering is too broad, you can also use the `if-ned-id` XML processing instruction. Each NED package in NSO defines a unique *ned-id*, which distinguishes between different device types (and possibly firmware versions). Based on the configured ned-id of the device, you can apply different parts of the XML template. For example:

```

<config-template xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>{/device}</name>
      <config>
        <?if-ned-id cisco-ios-cli-3.0:cisco-ios-cli-3.0?>
        <interface xmlns="urn:ios">
          <GigabitEthernet>
            <name>{/interface}</name>
            <!-- ... -->
          </GigabitEthernet>
        </interface>
        <?end?>
      </config>
    </device>
  </devices>
</config-template>

```

The preceding template applies configuration for the interface only if the selected device uses the `cisco-ios-cli-3.0` ned-id. You can find the full code as part of the `examples.ncs/implement-a-service/iface-v3` example.

Shared Service Settings and Auxiliary 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 situations 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: Often a set of policies is defined that is shared between service instances. The policies, such as QoS, have data models of their own (not service models) and the mapping code reads data from those.
- Topology information: the service mapping might need to know how devices are connected, such as which network switches lie between two routers.
- Resources such as VLAN IDs or IP addresses, which might not be given as input parameters. They may be modeled separately in NSO or fetched from an external system.

It is important to design the service model considering the above requirements: what is input and what is available from other sources. In the latter case, in terms of implementation, an important distinction is made between accessing the existing data and allocating new resources. You must take special care for resource allocation, such as VLAN or IP address assignment, as discussed later on. For now, let us focus on using pre-existing shared data.

One example of such use is to define QoS policies "on the side." Only a reference to an existing QoS policy is supplied as input. This is a much better approach than giving all QoS parameters to every service instance. But note that, if you modify the QoS definitions the services are referring to, this will not immediately change the existing deployed service instances. In order to have the service implement the changed policies, you need to perform a **re-deploy** of the service.

A simpler example is a modified DNS configuration service that allows selecting from a predefined set of DNS servers, instead of supplying the DNS server directly as a service parameter. The main benefit in this case is that clients have no need to be aware of the actual DNS servers (and their IPs). In addition, this approach simplifies the management for the network operator, as all the servers are kept in a single place.

What is required to implement such a service? There are two parts. The first is the model and data that defines the available DNS server options, which are shared (used) across all the DNS service instances. The second is a modification to the service inputs and mapping logic to use this data.

For the first part, you must create a data model. If the shared data is specific to one service type, such as the DNS configuration, you can define it alongside the service instance model, in the service package. But sometimes this data may be shared between multiple types of service. Then it makes more sense to create a separate package for the shared data models.

In this case, define a new top-level container in the service's YANG file as:

```
container dns-options {
    list dns-option {
        key name;

        leaf name {
            type string;
        }

        leaf-list servers {
            type inet:ipv4-address;
        }
    }
}
```

```
}
```

Note that the container is defined *outside* the service list because this data is not specific to individual service instances:

```
container dns-options {
    // ...
}

list dns {
    key name;

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

    // ...
}
```

The dns-options container includes a list of dns-option items. Each item defines a set of DNS servers (leaf-list) and a name for this set.

Once the shared data model is compiled and loaded into NSO, you can define the available DNS server sets:

```
admin@ncs(config)# dns-options dns-option lon servers 192.0.2.3
admin@ncs(config-dns-option-lon)# top
admin@ncs(config)# dns-options dns-option sto servers 192.0.2.3
admin@ncs(config-dns-option-sto)# top
admin@ncs(config)# dns-options dns-option sjc servers [ 192.0.2.5 192.0.2.6 ]
admin@ncs(config-dns-option-sjc)# commit
```

You must also update the service instance model to allow clients to pick one of these DNS servers:

```
list dns {
    key name;

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

    leaf name {
        type string;
    }

    leaf target-device {
        type string;
    }

    // Replace the old, explicit IP with a reference to shared data
    // leaf dns-server-ip {
    //     type inet:ip-address {
    //         pattern "192\.\.0\.\.2\..\*";
    //     }
    // }
    leaf dns-servers {
        mandatory true;
        type leafref {
            path "/dns-options/dns-option/name";
        }
    }
}
```

Different ways exist to model the service input for dns-servers. The first option you might think about might be using a string type and a pattern to limit the inputs to one of lon, sto, or sjc. Another option

would be to use a YANG enum type. But both of these have the drawback that you need to change the YANG model if you add or remove available dns-option items.

Using a leafref allows NSO to validate inputs for this leaf by comparing them to the values, returned by the path XPath expression. So, whenever you update the /dns-options/dns-option items, the change is automatically reflected in the valid dns-server values.

At the same time, you must also update the mapping to take advantage of this service input parameter. The service XML template is very similar to the previous one. The main difference is the the way in which the DNS addresses are read from the CDB, using the special deref() XPath function:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0"
    servicepoint="dns">
<devices xmlns="http://tail-f.com/ns/ncs">
    <device>
        <name>{/target-device}</name>
        <config>
            <ip xmlns="urn:ios">
                <name-server>{deref(/dns-servers)/../servers}</name-server>
            </ip>
        </config>
    </device>
</devices>
</config-template>
```

The deref() function “jumps” to the item selected by the leafref. Here, leafref's path points to /dns-options/dns-option/name, so this is where deref(/dns-servers) ends: at the name leaf of the selected dns-option item.

The following code, which performs the same thing but in a more verbose way, further illustrates how the DNS server value is obtained:

```
<ip xmlns="urn:ios">
    <?set dns_option = {/dns-servers}?>      <!-- Set $dns_option to e.g. 'lon' -->
    <?set-root-node {}?>                      <!-- Make '/' point to datastore root,
                                                instead of service instance -->
    <name-server>{/dns-options/dns-option[name=$dns_option]/servers}</name-server>
</ip>
```

The complete service is available in the examples.ncs/implement-a-service/dns-v3 example.

Service Actions

NSO provides some service actions out of the box, such as **re-deploy** or **check-sync**. You can also add others. A typical use case is to implement some kind of a self-test action that tries to verify the service is operational. The latter could use **ping** or similar network commands, as well as verify device operational data, such as routing table entries.

This action supplements the built-in **check-sync** or **deep-check-sync** action, which checks for the required device configuration.

For example, a DNS configuration service might perform a domain lookup to verify the Domain Name System is working correctly. Likewise, an interface configuration service could ping an IP address or check interface status.

The action consists of the YANG model for action inputs and outputs, as well as the action code that is executed when a client invokes the action.

Typically, such actions are defined per service instance, so you model them under the service list:

```

list iface {
    key name;

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

    leaf name { /* ... */ }
    leaf device { /* ... */ }
    leaf interface { /* ... */ }
    // ... other statements omitted ...

    action test-enabled {
        tailf:actionpoint iface-test-enabled;
        output {
            leaf status {
                type enumeration {
                    enum up;
                    enum down;
                    enum unknown;
                }
            }
        }
    }
}

```

The action needs no special inputs; because it is defined on the service instance, it can find the relevant interface to query. The output has a single leaf, called `status`, that uses an `enumeration` type for explicitly defining all the possible values it can take (up, down, or unknown).

Note that using the `action` statement requires you to also use the `yang-version 1.1` statement in the YANG module header (see [the section called “Actions”](#)).

Action Code in Python

NSO Python API contains a special-purpose base class, `ncs.dp.Action`, for implementing actions. In the `main.py` file, add a new class that inherits from it and implements an action callback:

```

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

```

The callback receives a number of arguments, one of them being `kp`. It contains a keypath value, identifying the data model path, to the service instance in this case, it was invoked on.

The keypath value uniquely identifies each node in the data model and is similar to an XPath path, but encoded a bit differently. You can use it with the `ncs.maagic.cd()` function to navigate to the target node.

```

root = ncs.maagic.get_root(trans)
service = ncs.maagic.cd(root, kp)

```

The newly defined `service` variable allows you to access all of the service data, such as `device` and `interface` parameters. This allows you to navigate to the configured device and verify status of the interface. The method likely depends on the device type and is not shown in this example.

The action class implementation then resembles the following:

```

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

```

```

root = ncs.maagic.get_root(trans)
service = ncs.maagic.cd(root, kp)

device = root.devices.device[service.device]

status = 'unknown'      # Replace with your own code that checks
                        # e.g. operational status of the interface

output.status = status

```

Finally, do not forget to register this class on the action point in the Main application.

```

class Main(ncs.application.Application):
    def setup(self):
        ...
        self.register_action('iface-test-enabled', IfaceActions)

```

You can test the action in the examples.ncs/implement-a-service iface-v4-py example.

Action Code in Java

Using the Java programming language, all callbacks, including service and action callback code, are defined using annotations on a callback class. The class NSO looks for is specified in the package-meta-data.xml file. This class should contain an @ActionCallback() annotated method that ties it back to the action point in the YANG model:

```

@ActionCallback(callPoint="iface-test-enabled",
                callType=ActionCBType.ACTION)
public ConfXMLParam[] test_enabled(DpActionTrans trans, Conftag name,
                                    ConfObject[] kp, ConfXMLParam[] params)
throws DpCallbackException {
    ...
}

```

The callback receives a number of arguments, one of them being kp. It contains a keypath value, identifying the data model path, to the service instance in this case, it was invoked on.

The keypath value uniquely identifies each node in the data model and is similar to an XPath path, but encoded a bit differently. You can use it with the com.tailf.navu.KeyPath2NavuNode class to navigate to the target node.

```

NavuContext context = new NavuContext(maapi);
NavuContainer service =
    (NavuContainer)KeyPath2NavuNode.getNode(kp, context);

```

The newly defined service variable allows you to access all of the service data, such as device and interface parameters. This allows you to navigate to the configured device and verify status of the interface. The method likely depends on the device type and is not shown in this example.

The complete implementation requires you to supply your own Maapi read transaction and resembles the following:

```

@ActionCallback(callPoint="iface-test-enabled",
                callType=ActionCBType.ACTION)
public ConfXMLParam[] test_enabled(DpActionTrans trans, Conftag name,
                                    ConfObject[] kp, ConfXMLParam[] params)
throws DpCallbackException {
    int port = NcsMain.getInstance().getNcsPort();

    // Ensure socket gets closed on errors, also ending any ongoing
    // session and transaction
    try (Socket socket = new Socket("localhost", port)) {

```

```

Maapi maapi = new Maapi(socket);
maapi.startUserSession("admin", InetAddress.getByName("localhost"),
                      "system", new String[] {}, MaapiUserSessionFlag.PROTO_TCP);

NavuContext context = new NavuContext(maapi);
context.startRunningTrans(Conf.MODE_READ);

NavuContainer root = new NavuContainer(context);
NavuContainer service =
    (NavuContainer)KeyPath2NavuNode.getNode(kp, context);

String status = "unknown"; // Replace with your own code that
                           // checks e.g. operational status of
                           // the interface

String nsPrefix = name.getPrefix();
return new ConfXMLParam[] {
    new ConfXMLParamValue(nsPrefix, "status", new ConfBuf(status)),
};
} catch (Exception e) {
    throw new DpCallbackException(name.toString() + " action failed",
                                   e);
}
}
}

```

You can test the action in the `examples.ncs/implement-a-service/iface-v4-java` example.

Operational Data

In addition to device configuration, services may also provide operational status or statistics. This is operational data, modeled with `config false` statements in YANG, and cannot be directly set by clients. Instead, clients can only read this data, for example to check service health.

What kind of data a service exposes depends heavily on what the service does. Perhaps the interface configuration service needs to provide information on whether a network interface was enabled and operational at the time of the last check (because such a check could be expensive).

Taking `iface` service as a base, consider how you can extend the instance model with another operational leaf to hold the interface status data as of the last check.

```

list iface {
    key name;

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

    // ... other statements omitted ...

    action test-enabled {
        tailf:actionpoint iface-test-enabled;
        output {
            leaf status {
                type enumeration {
                    enum up;
                    enum down;
                    enum unknown;
                }
            }
        }
    }
}

```

```

leaf last-test-result {
    config false;
    type enumeration {
        enum up;
        enum down;
        enum unknown;
    }
}

```

The new leaf `last-test-result` is designed to store the same data as the `test-enabled` action returns. Importantly, it also contains a `config false` substatement, making it operational data.

When faced with duplication of type definitions, as seen in the preceding code, the best practice is to consolidate the definition in a single place and avoid potential discrepancies in the future. You can use a `typedef` statement to define a custom YANG data type.



Note The `typedef` statements should come before data statements, such as containers and lists in the model.

```

typedef iface-status-type {
    type enumeration {
        enum up;
        enum down;
        enum unknown;
    }
}

```

Once defined, you can use the new type as you would any other YANG type. For example:

```

leaf last-test-status {
    config false;
    type iface-status-type;
}

action test-enabled {
    tailf:actionpoint iface-test-enabled;
    output {
        leaf status {
            type iface-status-type;
        }
    }
}

```

Users can then view operational data with the help of the `show` command. The data is also available through other NB interfaces, such as NETCONF and RESTCONF.

```
admin@ncs# show iface test-instance1 last-test-status
iface test-instance1 last-test-status up
```

But where does the operational data come from? The service application code provides this data. In this example, the `last-test-status` leaf captures the result of the enabled check, which is implemented as a custom action. So, here it is the action code that sets the leaf's value.

This approach works well when operational data is updated based on some event, such as a received notification or a user action, and NSO is used to cache its value.

For cases, where this is insufficient, NSO also allows producing operational data on demand, each time a client requests it, through the Data Provider API. See [the section called “DP API”](#) for this alternative approach.

Writing Operational Data in Python

Unlike configuration data, which always requires a transaction, you can write operational data to NSO with or without a transaction. Using a transaction allows you to easily compose multiple writes into a single atomic operation but has some small performance penalty due to transaction overhead.

If you avoid transactions and write data directly, you must use the low-level CDB API, which requires manual connection management and does not support Maagic API for data model navigation.

```
with contextlib.closing(socket.socket()) as s:
    _ncs.cdb.connect(s, _ncs.cdb.DATA_SOCKET, ip='127.0.0.1', port=_ncs.PORT)
    _ncs.cdb.start_session(s, _ncs.cdb.OPERATIONAL)
    _ncs.cdb.set_elem(s, 'up', '/iface{test-instance1}/last-test-status')
```

The alternative, transaction-based approach uses high-level MAAPI and Maagic objects:

```
with ncs.maapi.single_write_trans('admin', 'python', db=ncs.OPERATIONAL) as t:
    root = ncs.maagic.get_root(t)
    root iface['test-instance1'].last_test_status = 'up'
    t.apply()
```

When used as part of the action, the action code might be as follows:

```
def cb_action(self, uinfo, name, kp, input, output, trans):
    with ncs.maapi.single_write_trans('admin', 'python',
                                      db=ncs.OPERATIONAL) as t:
        root = ncs.maagic.get_root(t)
        service = ncs.maagic.cd(root, kp)

        # ...
        service.last_test_status = status
        t.apply()

    output.status = status
```

Note that you have to start a new transaction in the action code, even though `trans` is already supplied, since `trans` is read-only and cannot be used for writes.

Another thing to keep in mind with operational data is that NSO by default does not persist it to storage, only keeps it in RAM. One way for the data to survive NSO restarts is to use the `tailf:persistent` statement, such as:

```
leaf last-test-status {
    config false;
    type iface-status-type;
    tailf:cdb-oper {
        tailf:persistent true;
    }
}
```

You can also register a function with the service application class to populate the data on package load, if you are not using `tailf:persistent`.

```
class ServiceApp(Application):
    def setup(self):
        ...
        self.register_fun(init_oper_data, lambda _: None)

    def init_oper_data(state):
        state.log.info('Populating operational data')
        with ncs.maapi.single_write_trans('admin', 'python',
```

```

        db=ncs.OPERATIONAL) as t:
    root = ncs.maagic.get_root(t)
    # ...
    t.apply()

    return state

```

The examples.ncs/implement-a-service/iface-v5-py example implements such code.

Writing Operational Data in Java

Unlike configuration data, which always requires a transaction, you can write operational data to NSO with or without a transaction. Using a transaction allows you to easily compose multiple writes into a single atomic operation but has some small performance penalty due to transaction overhead.

If you avoid transactions and write data directly, you must use the low-level CDB API, which does not support NAVU for data model navigation.

```

int port = NcsMain.getInstance().getNcsPort();

// Ensure socket gets closed on errors, also ending any ongoing session/lock
try (Socket socket = new Socket("localhost", port)) {
    Cdb cdb = new Cdb("IfaceServiceOpervWrite", socket);
    CdbSession session = cdb.startSession(CdbDbType.CDB_OPERATIONAL);

    String status = "up";
    ConfPath path = new ConfPath("/iface{}/last-test-status",
        "test-instance1");
    session.setElem(ConfEnumeration.getEnumByLabel(path, status), path);

    session.endSession();
}

```

The alternative, transaction-based approach uses high-level MAAPI and NAVU objects:

```

int port = NcsMain.getInstance().getNcsPort();

// Ensure socket gets closed on errors, also ending any ongoing
// session and transaction
try (Socket socket = new Socket("localhost", port)) {
    Maapi maapi = new Maapi(socket);
    maapi.startUserSession("admin", InetAddress.getByName("localhost"),
        "system", new String[] {}, MaapiUserSessionFlag.PROTO_TCP);

    NavuContext context = new NavuContext(maapi);
    context.startOperationalTrans(Conf.MODE_READ_WRITE);

    NavuContainer root = new NavuContainer(context);
    NavuContainer service =
        (NavuContainer)KeyPath2NavuNode.getNode(kp, context);

    // ...
    service.leaf("last-test-status").set(status);
    context.applyClearTrans();
}

```

Note the use of the `context.startOperationalTrans()` function to start a new transaction against the operational data store. In other respects, the code is the same as for writing configuration data.

Another thing to keep in mind with operational data is that NSO by default does not persist it to storage, only keeps it in RAM. One way for the data to survive NSO restarts is to model the data with the `tailf:persistent` statement, such as:

```

leaf last-check-status {
    config false;
    type iface-status-type;
    tailf:cdb-oper {
        tailf:persistent true;
    }
}

```

You can also register a custom `com.tailf.ncs.ApplicationComponent` class with the service application to populate the data on package load, if you are not using `tailf:persistent`. Please refer to the section called “[The application component type](#)” for details.

The `examples.ncs/implement-a-service iface-v5-java` example implements such code.

Nano Services for Provisioning with Side Effects

A FASTMAP service cannot perform explicit function calls with 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 an action to generate authentication key files or start a virtual machine. All such actions must occur before the service is created and provided as input parameters. This restriction is because 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.

Nano services use a technique called reactive FASTMAP (RFM) and provide a framework to safely execute actions with side effects by implementing the service as several smaller (nano) steps or stages. 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 services discussed previously in this chapter were modeled to give all required parameters to the service instance. The mapping logic code could immediately do its work. Sometimes this is not possible. Two examples that require staged provisioning where a nano service step executing an action is the best practice solution:

- Allocating a resource from an external system, such as an IP address, or generating an authentication key file using an external command. It is impossible to do this allocation from within the normal FASTMAP `create()` code since there is no way to deallocate the resource on commit, abort, or failure and when deleting the service. Furthermore, the `create()` code runs within the transaction lock. The time spent in services `create()` code should be as short as possible.
- The service requires the start of one or more Virtual Machines, Virtual Network Functions. The VMs do not 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 concepts of nano services are covered in detail by [Chapter 31, Nano Services for Staged Provisioning](#). The example in `examples.ncs/development-guide/nano-services/netsim-sshkey` implements SSH public key authentication setup using a nano service. The nano service uses the following steps in a plan that produces the generated, distributed, and configured states:

-
- | | |
|---------------|---|
| Step 1 | Generates the NSO SSH client authentication key files using the OpenSSH <code>ssh-keygen</code> utility from a nano service side-effect action implemented in Python. |
| Step 2 | Distributes the public key to the netsim (ConfD) network elements to be stored as an authorized key using a Python service <code>create()</code> callback. |
| Step 3 | Configures NSO to use the public key for authentication with the netsim network elements using a Python service <code>create()</code> callback and service template. |

- Step 4** Test the connection using the public key through a nano service side-effect executed by the NSO built-in **connect** action.

Upon deletion of the service instance, NSO restores the configuration. The only delete step in the plan is the generated state side-effect action that deletes the key files. The example is described in more detail by Chapter 5, *Developing and Deploying a Nano Service* in *Getting Started*.

The `basic-vrouter`, `netsim-vrouter`, and `mpls-vpn-vrouter` examples in the `examples.ncs/development-guide/nano-services` directory start, configure, and stop virtual devices. In addition, the `mpls-vpn-vrouter` example manages Layer3 VPNs in a service provider MPLS network consisting of physical and virtual devices. Using a Network Function Virtualization (NFV) setup, the L3VPN nano service instructs a VM manager nano service to start a virtual device in a multi-step process consisting of the following:

- Step 1** When the L3VPN nano service `pe-create` state step `create` or `delete` a `/vm-manager/start` service configuration instance, the VM manager nano service instruct a VNF-M, called ESC, to start or stop the virtual device.
- Step 2** Wait for the ESC to start or stop the virtual device by monitoring and handling events. Here NETCONF notifications.
- Step 3** Mount the device in the NSO device tree.
- Step 4** Fetch the ssh-keys and perform a `sync-from` on the newly created device.

See the `mpls-vpn-vrouter` example for details on how the `l3vpn.yang` YANG model `l3vpn-plan` `pe-created` state and `vm-manager.yang` `vm-plan` for more information. `vm-manager` plan states with a nano-callback have their callbacks implemented by the `escstart.java` `escstart` class. Nano services are documented by [Chapter 31, Nano Services for Staged Provisioning](#).

Service Troubleshooting

Service troubleshooting is an inevitable part of any NSO development process and eventually a part of their operational tasks as well. By their nature, NSO services are composed primarily out of user-defined code, models, and templates. This gives you plenty of opportunities to make unintended mistakes in mapping code, use incorrect indentations, create invalid configuration templates, and much more. Not only that, they also rely on southbound communication with devices of many different versions and vendors, which presents you with yet another domain that can cause issues in your NSO services.

This is why it is important to have a systematic approach when debugging and troubleshooting your services:

- *Understand the problem* - First, you need to make sure you fully understand the issue you are trying to troubleshoot. Why is this issue happening? When did it first occur? Does it happen on only specific deployments or devices? What is the error message like? Is it consistent and can it be replicated? What do the logs say?
- *Identify the root cause* - When you understand the issues, their triggers, conditions, any additional insights that NSO allows you to inspect, you can start breaking down the problem to identify its root cause.
- *Form and implement the solution* - Once the root cause (or several of them) is found, you can focus on producing a suitable solution. This might be a simple NSO operation, modification of service package codebase, a change in southbound connectivity of managed devices, and any other action or combination required to achieve a working service.

Common troubleshooting steps

You can use these general steps to give you a high-level idea of how to approach troubleshooting your NSO services:

Step 1 Ensure that your NSO instance is installed and running properly. You can verify the overall status with `ncs --status` shell command. To find out more about installation problems and potential runtime issues, check the section called “Troubleshooting” in *Administration Guide*.

If you encounter a blank CLI when you connect to NSO you must also make sure that your user is added to correct NACM group (for example `ncsadmin`) and that the rules for this group allow the user to view and edit your service through CLI. You can find out more about groups and authorization rules in Chapter 9, *The AAA infrastructure* in *Administration Guide*.

Step 2 Verify that you are using the latest version of your packages. This means copying latest packages into load path, recompiling the package YANG models and code with the `make` command and reloading the packages. In the end, you must expect the NSO packages to be successfully reloaded to proceed with troubleshooting. You can read more about loading packages in [the section called “Loading Packages”](#). If nothing else, successfully reloading packages will at least make sure that you can use and try to create service instances through NSO.

Compiling packages uses the `ncsc` compiler internally, which means that this part of the process reveals any syntax errors that might exist in YANG models or Java code. You do not need to rely on `ncsc` for compile-level errors though and should use specialized tools such as `pyang` or `yanger` for YANG, and one of the many IDEs and syntax validation tools for Java.

```
yang/demo.yang:32: error: expected keyword 'type' as substatement to 'leaf'
make: *** [Makefile:41: ../load-dir/demo.fxs] Error 1

[javac] /nso-run/packages/demo/src/java/src/com/example/demo/demoRFS.java:52: error: ';' expected
[javac]           Template myTemplate = new Template(context, "demo-template")
[javac]
[javac] 1 error
[javac] 1 warning
```

BUILD FAILED

Additionally, reloading packages can also supply you with some valuable information. For example, it can tell you that the package requires a higher version of NSO which is specified in the `package-meta-data.xml` file, or about any Python related syntax errors.

```
admin@ncs# packages reload
Error: Failed to load NCS package: demo; requires NCS version 6.3

admin@ncs# packages reload
reload-result {
    package demo
    result false
    info SyntaxError: invalid syntax
}
```

Last but not least, package reloading also provides some information on the validity of your XML configuration templates based on the NED namespace you are using for a specific part of configuration, or just general syntactic errors in your template.

```
admin@ncs# packages reload
reload-result {
    package demol
    result false
    info demo-template.xml:87 missing tag: name
}
```

Common troubleshooting steps

```

reload-result {
    package demo2
    result false
    info demo-template.xml:11 Unknown namespace: 'ios-xr'
}
reload-result {
    package demo3
    result false
    info demo-template.xml:12: The XML stream is broken. Run-away < character found.
}

```

Step 3

Examine what the template and XPath expressions evaluate to. If some service instance parameters are missing or are mapped incorrectly, there might be an error in the service template parameter mapping or in their XPath expressions. Use the CLI pipe command **debug template** to show all the XPath expression results from your service configuration templates or **debug xpath** to output all XPath expression results for the current transaction (e.g. as a part of the YANG model as well).

In addition, you can use the **xpath eval** command in CLI configuration mode to test and evaluate arbitrary XPath expressions. The same can be done with **ncs_cmd** from command shell. To see all the XPath expression evaluations in your system, you can also enable and inspect the **xpath.trace** log. You can read more about debugging templates and XPath in [the section called “Debugging templates”](#). If you are using multiple versions of the same NED, make sure that you are using the correct processing instructions as described in [the section called “Namespaces and multi-NED support”](#) when applying different bits of configuration to different versions of devices.

```

admin@ncs# devtools true
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# xpath eval /devices/device
admin@ncs(config)# xpath eval /devices/device[name='r0']

```

Step 4

Validate that your custom service code is performing as intended. Depending on your programming language of choice, there might be different options to do that. If you are using Java, you can find out more on how to configure logging for the internal Java VM Log4j in [the section called “Logging”](#). You can use a debugger as well, to see the service code execution line by line. To learn how to use Eclipse IDE to debug Java package code, read [the section called “Using Eclipse to Debug the Package Java Code”](#).

The same is true for Python. NSO uses the standard **logging** module for logging, which can be configured as per instructions in [the section called “Debugging of Python packages”](#). Python debugger can be set up as well with **debugpy** or **pydevd-pycharm** modules.

Step 5

Inspect NSO logs for hints. NSO features extensive logging functionality for different components, where you can see everything from user interactions with the system, to low level communications with managed devices. For best results, set the logging level to DEBUG or lower. To learn what types of logs there are and how to enable them, consult the section called “Logging” in *Administration Guide*.

Another useful option is to append a custom trace ID to your service commits. The trace ID can be used to follow the request in logs from its creation all the way to the configuration changes that get pushed to the device. In case no trace ID is specified, NSO will generate a random one, but custom trace IDs are useful for focused troubleshooting sessions.

```

admin@ncs(config)# commit trace-id myTrace1
Commit complete.

```

Trace ID can also be provided as a commit parameter in your service code, or as a RESTCONF query parameter. See `examples.ncs/development-guide/commit-parameters` for an example.

Step 6

Measuring the time it takes for specific commands to complete can also give you some hints about what is going on. You can do this by using the **timecmd**, which requires the devtools to be enabled.

```

admin@ncs# devtools true
admin@ncs(config)# timecmd commit
Commit complete.

```

Command executed in 5.31 seconds.

Another useful tool to examine how long a specific event or command takes is the progress trace. See how it is used in [Chapter 30, Progress Trace](#).

Step 7

Double-check your servicepoints in the model, templates and in code. Since configuration templates don't get applied if the servicepoint attribute doesn't match the one defined in the service model or are not applied from the callbacks registered to specific servicepoints, make sure they match and that they are not missing. Otherwise you might notice errors such as the following ones.

```
admin@ncs# packages reload
reload-result {
    package demo
    result false
    info demo-template.xml:2 Unknown servicepoint: notdemo
}

admin@ncs(config-demo-s1)# commit dry-run
Aborted: no registration found for callpoint demo/service_create of type=external
```

Step 8

Verify YANG imports and namespaces. If your service depends on NED or other YANG files, make sure their path is added to where the compiler can find them. If you are using the standard service package skeleton, you can add to that path by editing your service package `Makefile` and adding the following line.

```
YANGPATH += ../../my-dependency/src/yang \
```

Likewise, when you use data types from other YANG namespaces in either your service model definition or by referencing them in XPath expressions.

```
// Following XPath might trigger an error if there is collision for the 'interfaces' node with other modules
path "/ncs:devices/ncs:device['r0']/config/interfaces/interface";
yang/demo.yang:25: error: the node 'interfaces' from module 'demo' (in node 'config' from 'tailf-ncs') has the same name as another node

// And the following XPath will not, since it uses namespace prefixes
path "/ncs:devices/ncs:device['r0']/config/iosxr:interfaces/iosxr:interface";
```

Step 9

Trace the southbound communication. If the service instance creation results in different configuration than would be expected from the NSO point of view, especially with custom NED packages, you can try enabling the southbound tracing (either per-device or globally).

```
admin@ncs(config)# devices global-settings trace pretty
admin@ncs(config)# devices global-settings trace-dir ./my-trace
admin@ncs(config)# commit
```

Common troubleshooting steps



CHAPTER 8

Templates

NSO comes with a flexible and powerful built-in templating engine, which is based on XML. The templating system simplifies how you apply configuration changes across devices of different types and provides additional validation against the target data model. Templates are a convenient, declarative way of updating structured configuration data and allow you to avoid lots of boilerplate code.

You will most often find this type of configuration templates used in services, which is why they are sometimes also called service templates. However, we mostly refer to them simply as XML templates, since they are defined in XML files.

NSO loads templates as part of a package, looking for XML files in the `templates` subdirectory. You then apply an XML template through API or by connecting it with a service through a service-point, allowing NSO to use it whenever a service instance needs updating.



Note

XML templates are distinct from so-called “device templates”, which are dynamically created and applied as needed by the operator, for example in the CLI. There are also other types of templates in NSO, unrelated to XML templates described here.

- [Structure of a template, page 94](#)
- [Other ways to generate the XML template structure, page 95](#)
- [Values in a template, page 97](#)
- [Conditional statements, page 98](#)
- [Loop statements, page 99](#)
- [Template operations, page 100](#)
- [Operations on ordered lists and leaf-lists, page 100](#)
- [Macros in templates, page 102](#)
- [XPath context in templates, page 104](#)
- [Namespaces and multi-NED support, page 105](#)
- [Passing deep structures from API, page 106](#)
- [Service callpoints and templates, page 108](#)
- [Debugging templates, page 109](#)
- [Processing instructions reference, page 112](#)

- [XPath functions, page 115](#)

Structure of a template

Template is an XML file with the `config-template` root element, residing in the `http://tail-f.com/ns/config/1.0` namespace. The root contains configuration elements according to NSO YANG schema and *XML processing instructions*.

Configuration element structure is very much like the one you would find in a NETCONF message, since it uses the same encoding rules defined by YANG. Additionally, each element can specify a `tags` attribute that refines how the configuration is applied.

A typical template for configuring an NSO-managed device is:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device tags="nocreate">
      <name>{/name}</name>
      <config tags="merge">
        <!-- ... -->
      </config>
    </device>
  </devices>
</config-template>
```

The first line defines the root node. It contains elements that follow the same structure as that used by the CDB, in particular the **devices device name config** path in the CLI. In the printout, two elements, `device` and `config`, also have a `tags` attribute.

You can write this structure by studying the YANG schema if you wish. However, a more typical approach is to start with manipulating NSO configuration by hand, such as through the NSO CLI or web UI. Then generate the XML structure with the help of NSO output filters. You can use **commit dry-run outformat xml** or **show ... | display xml** commands, or even the **ncs_load** utility. For a worked, step-by-step example, refer to [the section called “A Template is All You Need”](#).

```
admin@ncs(config)# devices device rtr01 config ...
admin@ncs(config-device-rtr01)# commit dry-run outformat xml
result-xml {
  local-node {
    data <devices xmlns="http://tail-f.com/ns/ncs">
      <device>
        <name>rtr01</name>
        <config>
          <!-- ... -->
        </config>
      </device>
    </devices>
  }
}
admin@ncs(config-device-rtr01)# commit
admin@ncs# show running-config devices device rtr01 config ... | display xml
<config xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>rtr01</name>
      <config>
        <!-- ... -->
      </config>
    </device>
  </devices>
```

```
</config>
```

Having the basic structure in place, you can then fine-tune the template by adding different processing instructions and tags, as well as replacing static values with variable references using the XPath syntax.

Note that a single template can configure multiple devices of different type, services, or any other configurable data in NSO; basically the same as you can do in a CLI commit. But a single, gigantic template can become a burden to maintain. That is why many developers prefer to split up bigger configurations into multiple *feature templates*, either by functionality or by device type.

Finally, the name of the file, without the .xml extension, is the name of the template. The name allows you to reference the template from code later on. Since all the template names reside in the same namespace, it is a good practice to use a common naming scheme, preferably `<package name>-<feature>.xml` to ensure template names are unique.

Other ways to generate the XML template structure

The NSO CLI features a **templatize** command that allows you to analyze a given configuration and find common configuration patterns. You can use these to, for example, create configuration template for a service.

Suppose you have existing interface configuration on a device:

```
admin@ncs# show running-config devices device c0 config interface GigabitEthernet
devices device c0
  config
    interface GigabitEthernet0/0/0/0
      ip address 10.1.2.3 255.255.255.0
    exit
    interface GigabitEthernet0/0/0/1
      ip address 10.1.4.3 255.255.255.0
    exit
    interface GigabitEthernet0/0/0/2
      ip address 10.1.9.3 255.255.255.0
    exit
!
!
```

Using the **templatize** command, you can search for patterns in this part of configuration, which produces the following:

```
admin@ncs# templatize devices device c0 config interface GigabitEthernet
Found potential templates at:
  devices device c0 \ config \ interface GigabitEthernet {$GigabitEthernet-name}

Template path:
  devices device c0 \ config \ interface GigabitEthernet {$GigabitEthernet-name}
Variables in template:
  {$GigabitEthernet-name}  {$address}

<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">
          <GigabitEthernet>
            <name>{$GigabitEthernet-name}</name>
            <ip>
              <address>
```

Other ways to generate the XML template structure

```

<primary>
    <address>{$address}</address>
    <mask>255.255.255.0</mask>
</primary>
</address>
</ip>
</GigabitEthernet>
</interface>
</config>
</device>
</devices>
</config>

```

In this case, NSO finds a single pattern (the only one) and creates the corresponding template. In general, NSO might produce a number of templates. As an example, try running the command within the `examples.ncs/implement-a-service/dns-v3` environment.

```

$ cd $NCS_DIR/examples.ncs/implement-a-service/dns-v3
$ make demo
admin@ncs# templatize devices device c*

```

The algorithm works by searching the data at the specified path. For any list it encounters, it compares every item in the list with its siblings. If the two items have the same structure but not necessarily the same actual values (for leafs), that part of configuration can be made into a template. If the two list items use the same value for a leaf, the value is used directly in the generated template. Otherwise, a unique variable name is created and used in its place, as shown in the example.

However, **templatize** requires you to reference existing configuration in NSO. If such configuration is not readily available to you and you want to avoid manually creating sample configuration in NSO first, you can use the `sample-xml-skeleton` functionality of the **yanger** utility to generate sample XML data directly:

```

$ cd $NCS_DIR/packages/neds/cisco-ios-cli-3.8/
$ yanger -f sample-xml-skeleton \
    --sample-xml-skeleton-doctype=config \
    --sample-xml-skeleton-path='/ip/name-server' \
    --sample-xml-skeleton-defaults \
    src/yang/tailf-ned-cisco-ios.yang
<?xml version='1.0' encoding='UTF-8'?>
<config xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <ip xmlns="urn:ios">
        <name-server>
            <name-server-list>
                <address/>
            </name-server-list>
            <vrf>
                <name/>
                <name-server-list>
                    <address/>
                </name-server-list>
            </vrf>
        </name-server>
    </ip>
</config>

```

You can replace the value of `--sample-xml-skeleton-path` with the path to the part of configuration you want to generate.

In case the target data model contains submodules, or references other non-built-in modules, you must also tell **yanger** where to find additional modules with the `-p` parameter, such as adding `-p src/yang/` to the invocation.

Values in a template

Some XML elements, notably those that represent leafs or leaf-lists, specify element text content as values that you wish to configure, such as:

```
<name>rtr01</name>
```

NSO converts the string value to the actual value type of the YANG model automatically when the template is applied.

Along with hard-coded, static content (`rtr01`), the value may also contain curly brackets (`{ . . . }`), which the templating engine treats as XPath 1.0 expressions.

The simplest form of an XPath expression is a plain XPath variable:

```
<name>{$CE}</name>
```

A value can contain any number of `{ . . . }` expressions and strings. The end result is the concatenation of all the strings and XPath expressions. For example:

```
<description>Link to PE: {$PE} - {$PE_INT_NAME}</description>
```

might evaluate to

```
<description>Link to PE: pe0 - GigabitEthernet0/0/0/3</description>
```

if you set PE to `pe0` and PE_INT_NAME to `GigabitEthernet0/0/0/3` when applying the template.

You set the values for variables in the code where you apply the template. NSO also sets some predefined variables, which you can reference:

<code>\$DEVICE</code>	The name of the current device. Cannot be overriden.
<code>\$TEMPLATE_NAME</code>	The name of the current template. Cannot be overriden.
<code>\$SCHEMA_OPAQUE</code>	Defined if the template is registered for a servicepoint (the top node in the template has <code>servicepoint</code> attribute) and the corresponding <code>ncs:servicepoint</code> statement in the YANG model has <code>tailf:opaque</code> substatement.
	Set to the value of the <code>tailf:opaque</code> statement.
<code>\$OPERATION</code>	Defined if the template is registered for a servicepoint with the <code>cbtype</code> attribute set to <code>pre-/post-modification</code> (see the section called “Service callpoints and templates”). Contains the requested service operation; create, update, or delete.

The `{ . . . }` expression can also be any other valid XPath 1.0 expression. To address a reachable node, you might for example use

```
/endpoint/ce/device
```

or

```
../ce/device
```

to select a leaf node, `device`. NSO then uses the value of this leaf, say `ce5`, when constructing the value of the expression.

However, there are some special cases. If the result of the expression is a node-set (e.g. multiple leafs), and the target is a leaf-list or a list's key leaf, the template configures multiple destination nodes. This handling allows you to set multiple values for a leaf-list or set multiple list items.

Similarly, if the result is an empty node-set, nothing is set (the set operation is ignored).

Finally, what nodes are reachable in the XPath expression, and how, depends on the root node and context used in the template. See [the section called “XPath context in templates”](#).

Conditional statements

The `if`, and the accompanying `elif`, `else`, processing instructions make it possible to apply parts of the template, based on a condition. For example:

```
<policy-map xmlns="urn:ios" tags="merge">
  <name>{$POLICY_NAME}</name>
  <class>
    <name>{$CLASS_NAME}</name>
    <?if {qos-class/priority = 'realtime'}?>
      <pri<priority-realtime>
        <percent>{$CLASS_BW}</percent>
      </priority-realtime>
    <?elif {qos-class/priority = 'critical'}?>
      <pri<priority-critical>
        <percent>{$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 preceding template shows how to produce different configuration, for network bandwidth management in this case, when different `qos-class/priority` values are specified.

In particular, the sub-tree containing the `pri<priority-realtime>` tag will only be evaluated if `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', but 'critical' instead.

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 not 'realtime' nor 'critical'.

In your own code you can of course use just a subset of these instructions, such as a simple `if - end` conditional evaluation. But note that every conditional evaluation must end with the `end` processing instruction, to allow nesting multiple conditionals.

The evaluation of the XPath statements used in the `if` and `elif` processing instructions follows 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.
- The value of the argument is either an empty string or numeric zero.

- The argument is of boolean type and evaluates to false, such as using the `not(true())` function.

Loop statements

The `foreach` and `for` processing instructions allow you to avoid needless repetition: they iterate over a set of values and apply statements in a sub-tree several times. For example:

```
<ip xmlns="urn:ios">
  <route>
    <?foreach { /tunnel } ?>
      <ip-route-forwarding-list>
        <prefix>{network}</prefix>
        <mask>{netmask}</mask>
        <forwarding-address>{tunnel-endpoint}</forwarding-address>
      </ip-route-forwarding-list>
    <?end?>
  </route>
</ip>
```

The printout shows the use of `foreach` to configure a set of IP routes (the list `ip-route-forwarding-list`) for a Cisco network router. If there is a `tunnel` list in the service model, the `{ /tunnel }` expression selects all the items from the list. If this is a non-empty set, then the sub-tree containing `ip-route-forwarding-list` is evaluated once for every item in that node-set.

For each iteration, the initial context is set to one node, that is, the node being processed in that iteration. The XPath function `current()` retrieves this initial context if needed. Using the context, you can access the node data with relative XPath paths, e.g. the `{network}` code in the example refers to `/tunnel[...]/network` for the current item.

`foreach` only supports a single XPath expression as its argument and the result needs to be a node-set, not a simple value. However, you may use XPath union operator to join multiple node-sets in a single expression when required: `{some-list-1 | some-leaf-list-2}`.

Similarly, `for` is a processing instruction that uses a variable to control the iteration, in line with traditional programming languages. For example, the following template disables the first four (0-3) interfaces on a Cisco router:

```
<interface xmlns="urn:ios">
  <?for i=0; {$i < 4}; i={$i + 1}?>
    <FastEthernet>
      <name>0/{$i}</name>
      <shutdown/>
    </FastEthernet>
  <?end?>
</interface>
```

In this example, three semicolon-separated clauses follow the `for` keyword:

- The first clause is the initial step executed before the loop is entered the first time. The format of the clause is that of a variable name followed by equals sign and an expression. The latter may combine literal strings and XPath expressions surrounded by `{ }`. The 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.

The `foreach` and `for` expressions make the loop explicit, which is why they are the first choice for most programmers. Alternatively, under certain circumstances, the template invokes an implicit loop, as described in the section called “[XPath context in templates](#)”.

Template operations

The most common use-case for templates is to produce new configuration but other behavior is possible too. This is accomplished by setting the `tags` attribute on XML elements.

NSO supports the following `tags` values, colloquially referred to as “tags”:

- `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 must not already exist. An error is raised if the node exists.

```
<GigabitEthernet tags="create">
  <name>{link/interface-number}</name>
  <description tags="merge">Link to PE</description>
  ...

```

- `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>
  ...

```

Tags `merge` and `nocreate` are inherited to their sub-nodes until a new tag is introduced.

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` behavior.

Tag `delete` applies only to the current node; any children (except keys specifying the list/leaf-list entry to delete) are ignored.

Operations on ordered lists and leaf-lists

For ordered-by-user lists and leaf-lists, where item order is significant, you can use the `insert` attribute to specify where in the list, or leaf-list, the node should be inserted. You specify whether 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, such as ACLs, you may need to ensure a particular order:

```

<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>

```

However, 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 some constraints on element positions that need to be fulfilled.

Following the ACL rules example, suppose that initially the list contains only the "deny-all" rule:

```

<rule>
  <name>deny-all</name>
  <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 the `insert="first"` operation. If there are two services creating one entry each, say 10.0.0.0/8 and 192.168.0.0/24 respectively, then the resulting configuration looks like:

```

<rule>
  <name>service-2</name>
  <ip>192.168.0.0</ip>
  <mask>255.255.255.0</mask>
  <action>permit</action>
</rule>
<rule>
  <name>service-1</name>
  <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>

```

Note that the rule for the second service comes first because it was configured last and inserted as the first item in the list.

If you now try to check-sync the first service (10.0.0.0/8), it will report as out-of-sync, and re-deploying it would move the 10.0.0.0/8 rule first. But what you really want is to ensure the deny-all rule comes last. This is when the `guard` attribute comes in handy.

If both the `insert` and `guard` attributes are specified on a list entry in a template, then the template engine first checks 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` attribute:

- If the element exists and fulfills this constraint, then its position is 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, in the ACL example, the template can specify the guard as follows:

```
<rule insert="first" guard="deny-all">
  <name>{$NAME}</name>
  <ip>{$IP}</ip>
  <mask>{$MASK}</mask>
  <action>permit</action>
</rule>
```

A guard can be specified literally (e.g. `guard="deny-all"` if "name" is the key of the list) or using an XPath expression (e.g. `guard="{$LASTRULE}"`). 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 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.
- 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 *value* attribute.
- 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 *or value* attribute.

Macros in templates

Templates support macros - named XML snippets that facilitate reuse and simplify complex templates. When you call a previously defined macro, the templating engine inserts the macro data, expanded with the values of the supplied arguments. The following example demonstrates the use of a macro.

Example 24. Template with macros

```
1 <config-template xmlns="http://tail-f.com/ns/config/1.0">
  <?macro GbEth name='{/name}' ip mask='255.255.255.0'?>
    <GigabitEthernet>
      <name>$name</name>
      5   <ip>
        <address>
          <primary>
            <address>$ip</address>
            <mask>$mask</mask>
          10       </primary>
          </address>
        </ip>
      </GigabitEthernet>
    <?endmacro?>
  15  <?macro GbEthDesc name='{/name}' ip mask='255.255.255.0' desc?>
    <?expand GbEth name='$name' ip='$ip' mask='$mask'?>
      <GigabitEthernet>
        <name>$name</name>
```

```

20      <description>$desc</description>
21      </GigabitEthernet>
22      <?endmacro?>

23      <devices xmlns="http://tail-f.com/ns/ncs">
24          <device tags="nocreate">
25              <name>{/device}</name>
26              <config tags="merge">
27                  <interface xmlns="urn:ios">
28                      <?expand GbEthDesc name='0/0/0/0' ip='10.250.1.1'
29                          desc='Link to core'?>
30                  </interface>
31                  </config>
32          </device>
33      </devices>
34  </config-template>

```

When using macros, be mindful of the following:

- A macro must be a valid chunk of XML, or a simple 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. Each parameter may define a default value.

In the preceding example, a macro is defined as

```
<?macro GbEth name='{/name}' ip mask='255.255.255.0'?>
```

Here, `GbEth` is the name of the macro. This macro takes three parameters, `name`, `ip`, and `mask`. The parameters `name` and `mask` have default values, and `ip` does not.

The default value for `mask` is a fixed string, while the one for `name` by default gets its value through an XPath expression.

- A macro can be expanded in another location in the template using the `<?expand?>` processing instruction. As shown in the example (line 29), 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, not supplying a value causes an error.

- 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 (line 17 in the example).

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 are 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 those macros.

When reporting errors in a template using 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 might resemble `service.xml:19:8 Invalid parameters for processing instruction set.` - meaning 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.

XPath context in templates

When the evaluation of a template starts, the XPath context node and root node are both set to either the service instance data node (with a template-only service) or the node specified with the API call to apply the template (usually service instance data node as well).

The *root node* is used as the starting point for evaluating absolute paths starting with / and puts a limit on where you can navigate with ../../.

You can access data outside the current root node subtree by dereferencing a leafref type leaf or by changing the root node from within the template.

To change the root node within the template, use the `set-root-node` XML processing instruction. The instruction 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. Note that, as all processing instructions, the effect of `set-root-node` only applies until the closing parent tag.

The *context node* refers to the node that is used as the starting point for navigation with relative paths, such as ../../device or device.

You can change the current context node using the `set-context-node` or other context-related processing instructions. For example: <?set-context-node { .. }?> changes the context node to the parent of the current context node.

There is a special case where NSO automatically changes the evaluation context as it progresses through and applies the template, which makes it easier to work with lists. There are two conditions required to trigger this special case:

- 1 The value being set in the template is the key of a list.
- 2 The XPath expression used for this key evaluates to a node-set, not a value.

To illustrate, consider the following example.

Suppose you are using the template to configure interfaces on a device. Target device YANG model defines the list of interfaces as:

```
list interface {
    key "name";
    leaf name {
        type string;
    }
    leaf address {
        type inet:ip-address;
    }
}
```

You also use a service model that allows configuring multiple links:

```
// ...
container links {
    list link {
        key "intf-name";
        leaf intf-name {
            type string;
        }
        leaf intf-addr {
```

```
        type inet:ip-address;
    }
}
```

The context changing mechanism allows you to configure device interface with the specified address using the template:

```
<interface>
  <name>{/links/link[0]/intf-name}</name>
  <address>{intf-addr}</address>
</interface>
```

The `/links/link[0]/intf-name` evaluates to a node and the evaluation context node is changed to the parent of this node, `/links/link[0]`, because name is a key leaf. Now you can refer to `/links/link[0]/intf-addr` with a simple relative path `{intf-addr}`.

The true power and usefulness of context changing becomes evident when used together with XPath expressions that produce node-sets with multiple nodes. You can create a template that configures multiple interfaces with their corresponding addresses (note the use of `link` instead of `link[0]`):

```
<interface>
  <name>{/links/link/intf-name}</name>
  <address>{intf-addr}</address>
</interface>
```

The first expression returns a node-set possibly including multiple leafs. NSO then configures multiple list items (interfaces), based on their name. The context change mechanism triggers as well, making {intf-addr} refer to the corresponding leaf in the same link definition. Alternatively, you can achieve the same outcome with a loop (see [the section called “Loop statements”](#)).

However, in some situations you may not desire to change the context. You can avoid it by making the XPath expression return a value instead of a node/node-set. The simplest way is to use the XPath `string()` function, for example:

```
<interface>
  <name>{string(/links-list/intf-name)}</name>
</interface>
```

Namespaces and multi-NED support

When a device makes itself known to NSO, it presents a list of capabilities (see the section called “Capabilities, Modules and Revision Management” in *User Guide*), which includes what YANG modules that particular device supports. Since each YANG module defines a unique XML namespace, this information can be used in a template.

Hence, a template may include configuration for many diverse devices. The templating system streamlines this by applying only those pieces of the template that have a namespace matching one advertised by the device (see the section called “[Supporting Different Device Types](#)”).

Additionally, the system performs validation of the template against the specified namespace when loading the template as part of the package load sequence, allowing you to detect a lot of the errors at load time instead of at run time.

In case the namespace matching is insufficient, such as when you want to check for a particular version of a NED, you can use special processing instructions `if-ned-id` or `if-ned-id-match`. See [the section called “Processing instructions reference”](#) for details and [the section called “Supporting Different Device Types”](#) for an example.

However, strict validation against the currently loaded schema may become a problem for developing generic, reusable templates that should run in different environments with different sets of NEDs and NED versions loaded. For example, NSO instance having fewer NED versions than the template is designed for may result in some elements not being recognized, while having 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 has a concept of *supported-ned-ids*. This is a set of ned-ids the package developer declares in the *package-meta-data.xml* file, indicating all NEDs the XML templates contained in this package are designed to support. This gives NSO a hint on how to interpret the template.

Example 25. Package declaring supported-ned-id

```
<ncs-package xmlns="http://tail-f.com/ns/ncs-packages">
  <name>mypackage</name>
  <!-- ... -->

  <!-- Exact NED id match, requires namespace -->
  <supported-ned-id xmlns:id="http://tail-f.com/ns/ned-id/cisco-ios-cli-3.0">
    id:cisco-ios-cli-3.0
  </supported-ned-id>

  <!-- Regex-based NED id match -->
  <supported-ned-id-match>router-nc-1</supported-ned-id-match>
</ncs-package>
```

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 the 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 in the strict mode, using the full set of currently loaded ned-ids. This may make the package less reusable between different systems, but is usually fine in environments where the package is intended to be used in runtime systems fully under control of the package developer.

Passing deep structures from API

When applying the template via API, you typically pass parameters to a template through variables, as described in [the section called “Templates and Code”](#) and [the section called “Values in a template”](#). One limitation of this mechanism is that a variable can only hold one string value. Yet, sometimes there is a need to pass not just a single value, but a list, map, or even more complex data structures from API to the template.

One way to achieve this is to use smaller templates, such as invoking the template repeatedly, one-by-one for each list item (or perhaps 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 is split between Java or Python code and the template, which makes it harder to understand and implement.

An alternative approach described in this section involves modeling the required auxiliary data as operational data and populating it in the code, before applying the template. For a service, the service callback code in Java or Python first populates the auxiliary data, then passes control to the template, which handles the main service configuration logic. The auxiliary data is accessible in the template, by means of XPath, just like any other service input data.

There are different approaches to modeling 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 by native XPath functions is converting a network prefix into a network mask or vice versa. Below is a snippet of a 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 input IP address contains a prefix, but the target device accepts an IP address with network mask instead, then you can use an auxiliary operational leaf to pass the mask (calculated from the prefix) to the template.

```
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 code that calls the template needs to populate the mask. For example, using the Python Maagic API in a service:

```
def cb_create(self, tctx, root, service, proplist):
    interface_list = service.interface
    for intf in interface_list:
        prefix = intf.address.split('/')[1]
        intf.mask = ipaddress.IPv4Network(0, int(prefix)).netmask
```

```
# Template variables don't need to contain mask
# as it is passed via (operational) database
template = ncs.template.Template(service)
template.apply('iface-template')
```

The corresponding `iface-template` might then be as simple as:

```
<interface>
<name>{/interface/name}</name>
<ip-address>{substring-before(address, '/')}</ip-address>
<ip-mask>{mask}</ip-mask>
</interface>
```

Service callpoints and templates

The archetypical use case for XML templates is service provisioning and NSO allows you to directly invoke a template for a service, without writing boilerplate code in Python or Java. You can take advantage of this feature by configuring the `servicepoint` attribute on the root `config-template` element. For example:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0"
    servicepoint="some-service">
    <!-- ... -->
</config-template>
```

Adding the attribute registers this template for the given servicepoint, defined in the YANG service model. Without any additional attributes, the registration corresponds to the standard `create` service callback.



Note

While the template (file) name is not referred to in this case, it must still be unique in an NSO node.

In a similar manner you can register templates for each state of a nano service, using `componenttype` and `state` attributes. [the section called “Nano Service Callbacks”](#) contains examples.

Services also have pre- and post-modification callbacks, further described in [the section called “Service Callbacks”](#), which you can also implement with templates. Simply put, pre- and post-modification templates are applied before and after applying the main service template.

These pre- and post-modification templates can only be used in classic (non-nano) services when the `create` callback is implemented as a template. That is, they cannot be used together with `create` callbacks implemented in Java or Python. If you want to mix the two approaches for the same service, consider using nano services.

To define a template as pre- or post-modification, appropriately configure the `cbtype` attribute, along with `servicepoint`. The `cbtype` attribute supports these three values:

- pre-modification
- create
- post-modification



Note

NSO supports only a single registration for each servicepoint and callback type. Therefore, you cannot register multiple templates for the same servicepoint/cbtype combination.

The `$OPERATION` variable is set internally by NSO in pre- and post-modification templates to contain the service operation, i.e., `create`, `update` or `delete`, that triggered the callback. The `$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. Note that the service 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.

Example 26. Post modification template

```
<config-template xmlns="http://tail-f.com/ns/config/1.0"
                 servicepoint="some-service"
                 cbtype="post-modification">
    <if {$OPERATION = 'create'}?>
        <devices xmlns="http://tail-f.com/ns/ncs">
            <device>
                <name>{/device}</name>
                <config>
                    <!-- ... -->
                </config>
            </device>
        </devices>
    <elseif {$OPERATION = 'update'}?>
        <!-- ... -->
    <else?>
        <!-- $OPERATION = 'delete' -->
        <!-- ... -->
    </end?>
</config-template>
```

Debugging templates

You can request additional information when applying templates in order to understand what is going on. When applying or committing a template in the CLI, the `debug` pipe command enables debug information:

```
admin@ncs(config)# commit dry-run | debug template
admin@ncs(config)# commit dry-run | debug xpath
```

The **debug xpath** option outputs *all* XPath evaluations for the transaction, and is not limited to the XPath expressions inside templates.

The **debug template** option outputs XPath expression results from the template, under which context expressions are evaluated, what operation is used, and how it effects the configuration, for all templates that are invoked. You can narrow it down to only show debugging information for a template of interest:

```
admin@ncs(config)# commit dry-run | debug template 13vpn
```

Additionally, the template and xpath debugging can be combined:

```
admin@ncs(config)# commit dry-run | debug template | debug xpath
```

For XPath evaluation, you can also inspect the XPath trace log if it is enabled (e.g. with **tail -f logs/xpath.trace**). XPath trace is enabled in the `ncs.conf` configuration file and is enabled by default for the examples.

Another option to help you get the XPath selections right is to 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 namespace changes.

```
admin@ncs# show running-config devices device c0 config ios:interface | display xpath
/devices/device[name='c0']/config/ios:interface/FastEthernet[name='1/0']
```

Example debug template output

```
/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 with and 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 following 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]
```

Example debug template output

The following text walks through the output of the **debug template** command for a dns-v3 example service, found in `examples.ncs/implement-a-service/dns-v3`. To try it out for yourself, start the example with **make demo** and configure a service instance:

```
admin@ncs# config
admin@ncs(config)# load merge example.cfg
admin@ncs(config)# commit dry-run | debug template
```

The XML template used in the service is simple but non-trivial:

```
1 <config-template xmlns="http://tail-f.com/ns/config/1.0"
                     servicepoint="dns">
    <devices xmlns="http://tail-f.com/ns/ncs">
        <?foreach {/target-device}?>
        5      <device>
            <name>{.}</name>
            <config>
                <ip xmlns="urn:ios">
                    <?if {/dns-server-ip}?>
                    10                  <!-- If dns-server-ip is set, use that. -->
                    <name-server>{/dns-server-ip}</name-server>
                    <?else?>
                        <!-- Otherwise, use the default one. -->
                        <name-server>192.0.2.1</name-server>
                    15                  <?end?>
                    </ip>
                </config>
            </device>
            <?end?>
        20      </devices>
    </config-template>
```

Applying the template produces a substantial amount of output. Let's interpret it piece by piece. The output starts with:

```
Processing instruction 'foreach': evaluating the node-set \
  (from file "dns-template.xml", line 4)
Evaluating "/target-device" (from file "dns-template.xml", line 4)
Context node: /dns[name='instance1']
Result:
For /dns[name='instance1']/target-device[.=='c1'], it evaluates to []
For /dns[name='instance1']/target-device[.=='c2'], it evaluates to []
```

The templating engine found the `foreach` in the `dns-template.xml` file at line 4. In this case, it is the only `foreach` block in the file but in general there might be more. The `{ /target-device }` expression is evaluated using the `/dns[name='instance1']` context, resulting in the complete `/dns[name='instance1']/target-device` path. Note that the latter is based on the root node (not shown in the output), not the context node (which happens to be the same as the root node at the start of template evaluation).

NSO found two nodes in the leaf-list for this expression, which you can verify in the CLI:

```
admin@ncs(config)# show full-configuration dns instance1 target-device | display xpath
/dns[name='instance1']/target-device [ c1 c2 ]
```

Next comes:

```
Processing instruction 'foreach': next iteration: \
    context /dns[name='instance1']/target-device[.='c1'] \
    (from file "dns-template.xml", line 4)
Evaluating "." (from file "dns-template.xml", line 6)
Context node: /dns[name='instance1']/target-device[.='c1']
Result:
For /dns[name='instance1']/target-device[.='c1'], it evaluates to "c1"
```

The template starts with the first iteration of the loop with the `c1` value. Since the node was an item in a leaf-list, the context refers to the actual value. If instead it was a list, the context would refer to a single item in the list.

```
Operation 'merge' on existing node: /devices/device[name='c1'] \
    (from file "dns-template.xml", line 6)
```

This line signifies the system “applied” line 6 in the template, selecting the `c1` device for further configuration. The line also informs you the device (the item in the `/devices/device` list with this name) exists.

```
Processing instruction 'if': evaluating the condition \
    (from file "dns-template.xml", line 9)
Evaluating conditional expression "boolean(/dns-server-ip)" \
    (from file "dns-template.xml", line 9)
Context node: /dns[name='instance1']/target-device[.='c1']
Result: true - continuing
```

The template then evaluates the `if` condition, resulting in processing of the lines 10 and 11 in the template:

```
Processing instruction 'if': recursing (from file "dns-template.xml", line 9)
Evaluating "/dns-server-ip" (from file "dns-template.xml", line 11)
Context node: /dns[name='instance1']/target-device[.='c1']
Result:
For /dns[name='instance1'], it evaluates to "192.0.2.110"
Operation 'merge' on non-existing node: \
    /devices/device[name='c1']/config/ios:ip/name-server[.='192.0.2.110'] \
    (from file "dns-template.xml", line 11)
```

The last line shows how a new value is added to the target leaf-list, that was not there (non-existing) before.

```
Processing instruction 'else': skipping (from file "dns-template.xml", line 12)
Processing instruction 'foreach': next iteration: \
    context /dns[name='instance1']/target-device[.='c2'] \
    (from file "dns-template.xml", line 4)
```

As the `if` statement matched, the `else` part does not apply and a new iteration of the loop starts, this time with the `c2` value.

Now the same steps take place for the other, c2, device:

```
Evaluating "." (from file "dns-template.xml", line 6)
Context node: /dns[name='instance1']/target-device[.='c2']
Result:
For /dns[name='instance1']/target-device[.='c2'], it evaluates to "c2"
Operation 'merge' on existing node: /devices/device[name='c2'] \
    (from file "dns-template.xml", line 6)
Processing instruction 'if': evaluating the condition \
    (from file "dns-template.xml", line 9)
Evaluating conditional expression "boolean(/dns-server-ip)" \
    (from file "dns-template.xml", line 9)
Context node: /dns[name='instance1']/target-device[.='c2']
Result: true - continuing
Processing instruction 'if': recursing (from file "dns-template.xml", line 9)
Evaluating "/dns-server-ip" (from file "dns-template.xml", line 11)
Context node: /dns[name='instance1']/target-device[.='c2']
Result:
For /dns[name='instance1'], it evaluates to "192.0.2.110"
Operation 'merge' on non-existing node: \
    /devices/device[name='c2']/config/ios:ip/name-server[.='192.0.2.110'] \
        (from file "dns-template.xml", line 11)
Processing instruction 'else': skipping (from file "dns-template.xml", line 12)
```

Finally, the template processing completes as there are no more nodes in the loop, and NSO outputs the new dry-run configuration:

```
cli {
    local-node {
        data devices {
            device c1 {
                config {
                    ip {
                        -
                            name-server 192.0.2.1;
                        +
                            name-server 192.0.2.1 192.0.2.110;
                    }
                }
            }
            device c2 {
                config {
                    ip {
                        +
                            name-server 192.0.2.110;
                    }
                }
            }
        }
    }
}
+
dns instance1 {
    target-device [ c1 c2 ];
    dns-server-ip 192.0.2.110;
}
```

Processing instructions reference

NSO template engine supports a number of XML processing instructions to allow more dynamic templates:

- **Syntax:**

```
<?set v = value??>
```

Description: Allows you to assign new variable or manipulate existing value of variable v. If used to create a new variable, 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.

- **Syntax:**

```
<if {expression}>
...
<elif {expression}>
...
<else?>
...
<end?>
```

Description: 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”](#).

- **Syntax:**

```
<foreach {expression}>
...
<end?>
```

Description: 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”](#).

- **Syntax:**

```
<for v = start_value; {progress condition}; v = next_value?>
...
<end?>
```

Description: This processing instruction allows you to iterate over the same set of template tags by changing a variable value. The variable visibility scope obeys the same rules as the `set` processing instruction, except the variable value is carried over to the next iteration instead of being discarded at the end of each iteration.

Only the condition expression is mandatory, either or both of initial and next value assignment can be omitted, e.g.:

```
<for ; {condition}; ?>
```

For the detailed description see [the section called “Loop statements”](#).

- **Syntax:**

```
<copy-tree {source}?>
```

Description: This instruction is analogous to `copy_tree()` 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 `copy-tree` instruction in the template within the current context.

- **Syntax:**

```
<set-root-node {expression}?>
```

Description: 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. The expression must evaluate to exactly one node in the data tree.

- **Syntax:**

```
<?set-context-node {expression}?>
```

Description: Allows you 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.

- **Syntax:**

```
<?save-context name?>
```

Description: Store both the current context node and the root node of the XPath accessible tree with *name* being the key to access it later. It is possible to switch to this context later using *switch-context* with the name. Multiple contexts can be stored simultaneously under different names. Using *save-context* with the same name multiple times will result in the stored context being overwritten.

- **Syntax:**

```
<?switch-context name?>
```

Description: Used to switch to a context stored using *save-context* 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. *switch-context* 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 causes an error.

- **Syntax:**

```
<?if-ned-id ned-ids?>
  ...
<?elif-ned-id ned-ids?>
  ...
<?else?>
  ...
<?end?>
```

Description: If there are multiple versions of the same NED expected to be loaded in the system, which define different versions of the same namespace, this processing instruction helps to resolve ambiguities in the schema between different versions of the NED. The part of the template following this processing instruction, up to matching *elif-ned-id*, *else* or *end* processing instruction, is 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 *ned-ids* must contain one or more qualified ned-id identities separated by spaces. *elif-ned-id* is optional and used to define a part of template that applies to devices with another set of ned-ids than previously specified. Multiple *elif-ned-id* instructions are allowed in a single block of *if-ned-id* instructions. The set of ned-ids specified as a parameter to *elif-ned-id* instruction must be non-intersecting with the previously specified ned-ids in this block.

The *else* 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 “Namespaces and multi-NED support”](#).

- **Syntax:**

```
<?if-ned-id-match regex?>
...
<?elif-ned-id-match regex?>
...
<?else?>
...
<?end?>
```

Description: The `if-ned-id-match` and `elif-ned-id-match` processing instructions work similarly to `if-ned-id` and `elif-ned-id` but they accept a regular expression as argument instead of a list of ned-ids. The regular expression is matched against all of the ned-ids supported by the package. If the `if-ned-id-match` processing instruction is nested inside of another `if-ned-id-match` or `if-ned-id` processing instruction, then the regular expression will only be matched against the subset of ned-ids matched by the encompassing processing instruction. The `if-ned-id-match` and `elif-ned-id-match` processing instructions are only allowed inside a device's mounted configuration subtree rooted at `/devices/device/config`.

- **Syntax:**

```
<?macro name params...?>
...
<?endmacro?>
```

Description: Define a new macro with the specified name and optional parameters. Macro definitions must come at the top of the template, right after the `config-template` tag. For the detailed description see [the section called “Macros in templates”](#).

- **Syntax:**

```
<?expand name params...?>
```

Description: Insert and expand the named macro, using the specified values for parameters. For the detailed description see [the section called “Macros in templates”](#).

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 `{$x < 3}` a value `x='1'` is converted to integer 1 implicitly), in other cases an explicit conversion is needed. For example, using the expression `{$x > $y}`, if `x='9'` and `y='11'`, the result of the expression is true due to alphabetic order as both variables are strings. In order to compare the values as numbers, an explicit conversion of at least one argument is required: `{number($x) > $y}`.

XPath functions

This section lists a few useful functions, available in XPath expressions. The list is not exhaustive; please refer to the [XPath standard](#), [YANG standard](#), and NSO-specific extensions in the section called “**XPATH FUNCTIONS**” in *Manual Pages* for a full list.

Type conversion:

- [bit-is-set\(\)](#)

- [boolean\(\)](#)
- [enum-value\(\)](#)
- [number\(\)](#)
- [string\(\)](#)

String handling:

- [concat\(\)](#)
- [contains\(\)](#)
- [normalize-space\(\)](#)
- [re-match\(\)](#)
- [starts-with\(\)](#)
- [substring\(\)](#)
- [substring-after\(\)](#)
- [substring-before\(\)](#)
- [translate\(\)](#)

Model navigation:

- [current\(\)](#)
- [deref\(\)](#)
- [last\(\)](#)
- [sort-by\(\)](#) in *Manual Pages*

Other:

- [compare\(\)](#) in *Manual Pages*
- [count\(\)](#)
- [max\(\)](#) in *Manual Pages*
- [min\(\)](#) in *Manual Pages*
- [not\(\)](#)
- [sum\(\)](#)



CHAPTER 9

Services Deep Dive

This section of documentation discusses all of the implementation details of services in NSO. The reader should be already familiar with all the concepts described in the introductory chapters and [Chapter 7, *Implementing Services*](#).

For an introduction to services, see [Chapter 5, *Developing a Simple Service*](#) instead.

- [Common Service Model, page 117](#)
- [Services and Transactions, page 118](#)
- [Service Callbacks, page 121](#)
- [Persistent Opaque Data, page 122](#)
- [Defining Static Service Conflicts, page 123](#)
- [Reference Counting Overlapping Configuration, page 124](#)
- [Stacked Services, page 125](#)
- [Caveats and Best Practices, page 126](#)
- [Service Discovery and Import, page 128](#)
- [Partial Sync, page 135](#)

Common Service Model

Each service type in NSO extends a part of the data model (a list or a container) with the `ncs:servicepoint` statement and the `ncs:service-data` grouping. This is what defines an NSO service.

The service point instructs NSO to involve the service machinery (Service Manager) for management of that part of the data tree and the `ncs:service-data` grouping contains definitions common to all services in NSO. Defined in `tailf-ncs-services.yang`, `ncs:service-data` includes parts that are required for proper operation of FASTMAP and the Service Manager. Every service must therefore use this grouping as part of its data model.

In addition, `ncs:service-data` provides a common service interface to the users, consisting of:

check-sync, deep-check-sync
actions

Check if the configuration created by the service is (still) there. That is, a re-deploy of this service would produce no changes.

The deep variant also retrieves the latest configuration from all the affected devices, making it relatively expensive.

re-deploy, reactive-re-deploy actions	Re-run the service mapping logic and deploy any changes from the current configuration. Non-reactive variant supports commit parameters, such as dry-run. Reactive variant performs an asynchronous re-deploy as the user of the original commit and uses the commit parameters from the latest commit of this service. It is often used with nano services, such as restarting a failed nano service.
un-deploy action	Remove the configuration produced by the service instance but keep the instance data, allowing a re-deploy later. This action effectively deactivates the service, while keeping it in the system.
get-modifications action	Show the changes in the configuration that this service instance produced. Behaves as if this was the only service that made the changes.
touch action	Available in the configure mode, marks the service as being changed and allows re-deploying multiple services in the same transaction.
directly-modified, modified containers	List devices and services the configuration produced by this service affects directly or indirectly (through other services).
used-by-customer-service leaf-list	List of customer services (defined under <code>/services/customer-service</code>) that this service is part of. Customer service is an optional concept that allows you to group multiple NSO services as belonging to the same customer.
commit-queue container	Contains commit queue items related to this service. See the section called “Commit Queue” in <i>User Guide</i> for details.
created, last-modified, last-run leafs	Date and time of the main service events.
log container	Contains log entries for important service events, such as those related to the commit queue or generated by user code. Defined in <code>tailf-ncs-log.yang</code> .
plan-location leaf	Location of the plan data if service plan is used. See Chapter 31, Nano Services for Staged Provisioning for more on service plans and using alternative plan locations.

While not part of `ncs:service-data` as such, you may consider the `service-commit-queue-event` notification part of the core service interface. The notification provides information about the state of the service when service uses the commit queue. As an example, an event-driven application uses this notification to find out when a service instance has been deployed to the devices. See the `showcase_rc.py` script in `examples.ncs/development-guide/concurrency-model/perf-stack/` for sample Python code, leveraging the notification. See `tailf-ncs-services.yang` for the full definition of the notification.

NSO Service Manager is responsible for providing the functionality of the common service interface, requiring no additional user code. This interface is the same for classic and nano services, whereas nano services further extend the model.

Services and Transactions

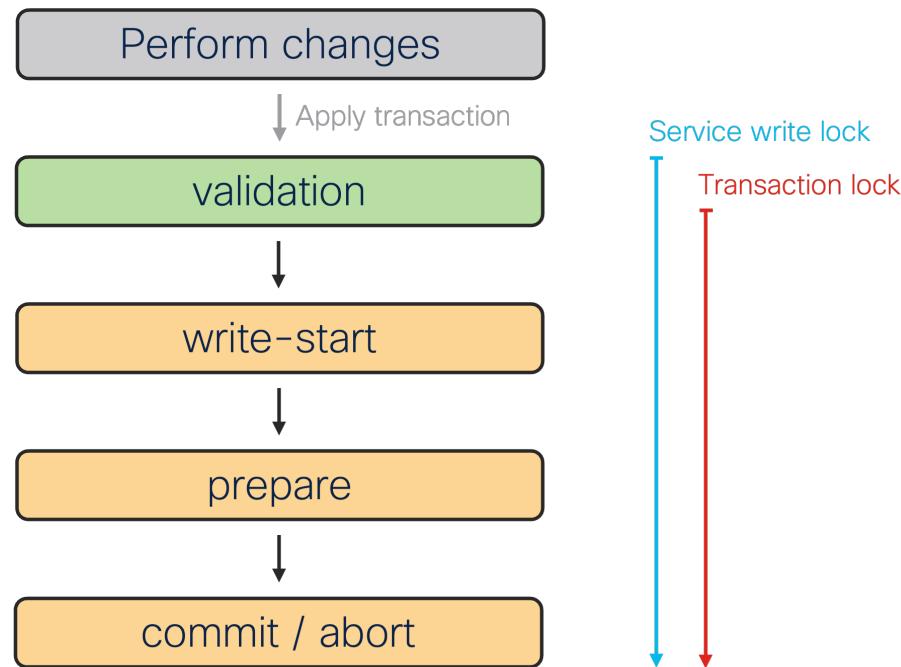
NSO calls into Service Manager when accessing actions and operational data under the common service interface, or when the service instance configuration data (the data under the service point) changes. NSO being a transactional system, configuration data changes happen in a transaction.

When applied, a transaction goes through multiple stages, as shown by the progress trace (e.g. using `commit | details` in the CLI). The detailed output breaks up the transaction into four distinct phases:

- 1 validation
- 2 write-start
- 3 prepare
- 4 commit

These phases deal with how the network-wide transactions work: *validation* phase prepares and validates the new configuration (including NSO copy of device configurations), then the CDB processes the changes and prepares them for local storage in the *write-start* phase. The *prepare* stage sends out the changes to network through the Device Manager and the HA system. The changes are staged (e.g. in candidate data store) and validated if device supports it, otherwise the changes are activated immediately. If all systems took the new configuration successfully, enter the *commit* phase, marking the new NSO configuration as active and activating or committing the staged configuration on remote devices. Otherwise, enter the *abort* phase, discarding changes and ask NEDs to revert activated changes on devices that do not support transactions (e.g. without candidate data store).

Figure 27. Typical transaction phases



There are also two types of locks involved with the transaction that are of interest to service developer; the *service write lock* and the *transaction lock*. The latter is a global lock, required to serialize transactions, while the former is a per-service-type lock for serializing services that cannot be run in parallel. See [Chapter 23, Scaling and Performance Optimization](#) for more details and their impact on performance.

The first phase, historically called *validation*, does more than just validate data and is the phase a service deals with the most. The other three support the NSO service framework but a service developer rarely interacts with directly.

We can further break down the first phase into the following stages:

- 1 rollback creation
- 2 pre-transform validation
- 3 transforms
- 4 full data validation
- 5 conflict check and transaction lock

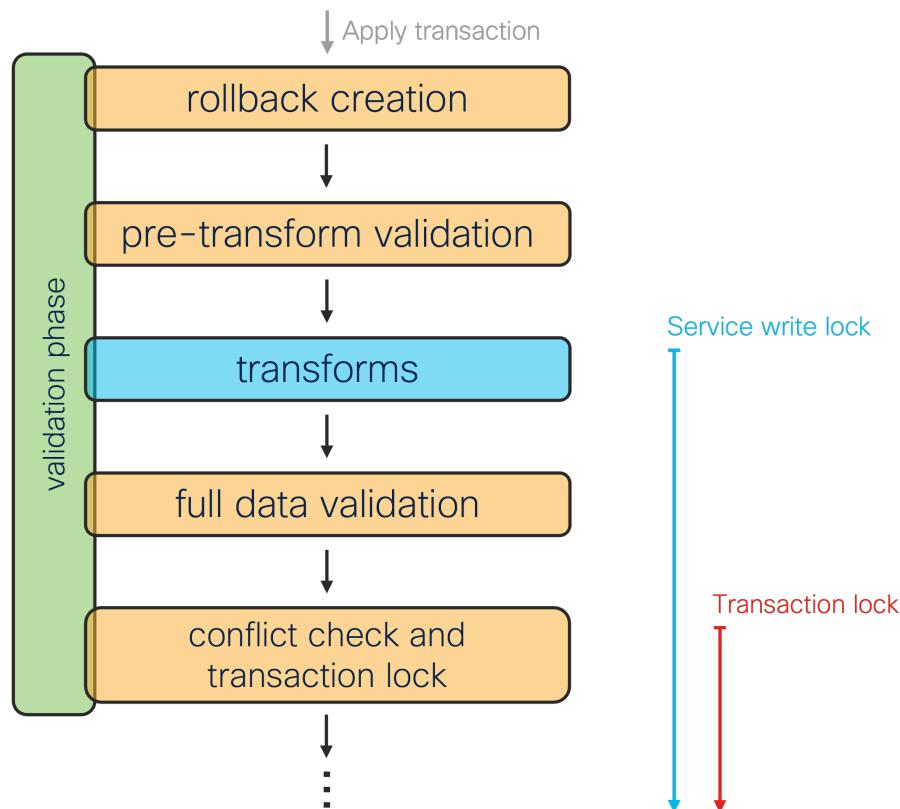
When transaction starts applying, NSO captures the initial intent and creates a rollback file, which allows one to reverse or roll back the intent. For example, the rollback file might contain the information that you changed a service instance parameter but it would not contain the service-produced device changes.

Then the first, partial validation takes place. It ensures the service input parameters are valid according to the service YANG model, so the service code can safely use provided parameter values.

Next, NSO runs transaction hooks and performs the necessary transforms, which alter the data before it is saved, for example encrypting passwords. This is also where Service Manager invokes FASTMAP and service mapping callbacks, recording the resulting changes. NSO takes service write locks in this stage, too.

After transforms, there are no more changes to the configuration data and the full validation starts, including YANG model constraints over the complete configuration, custom validation through validation points, and configuration policies (the section called “Policies” in *User Guide*).

Figure 28. Stages of transaction validation phase



Throughout the phase, transaction engine makes checkpoints, so it can restart the transaction faster in case of concurrency conflicts. The check for conflicts happens at the end of this first phase, when NSO

also takes the global transaction lock. Concurrency is further discussed in [Chapter 24, NSO Concurrency Model](#).

Service Callbacks

The main callback associated with a service point is the *create* callback, designed to produce the required (new) configuration, while FASTMAP takes care of the other operations, such as update and delete.

NSO implements two additional, optional callbacks for scenarios where *create* is insufficient. These are *pre-* and *post-modification* callbacks that NSO invokes before (pre) or after (post) *create*. These callbacks work outside of the scope tracked by FASTMAP. That is, changes done in *pre-* and *post-modification* do not automatically get removed during update or delete of the service instance.

For example, you can use the *pre-modification* callback to check the service prerequisites (pre-check) or make changes that you want persisted even after the service is removed, such as enabling some global device feature. The latter may be required when NSO is not the only system managing the device and removing the feature configuration would break non-NSO managed services.

Similarly, you might use *post-modification* to reset the configuration to some default after the service is removed. Say the service configures an interface on a router for customer VPN. However, when the service is deprovisioned (removed), you don't want to simply erase the interface configuration. Instead, you want to put it in shutdown and configure it for a special, unused VLAN. The *post-modification* callback allows you to achieve this goal.

The main difference from *create* callback is that *pre-* and *post-modification* are called on update and delete, as well as service create. Since the service data node may no longer exist in case of delete, the API for these callbacks does not supply the `service` object. Instead, the callback receives the operation and key path to the service instance. See the following API signatures for details.

Example 29. Service callback signatures in Python

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

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

@Service.post_modification
def cb_post_modification(self, tctx, op, kp, root, proplist): ...
```

The Python callbacks use the following function arguments:

tctx	A <code>TransCtxRef</code> object containing transaction data, such as user session and transaction handle information.
op	Integer representing operation: <code>create</code> (<code>ncs.dp.NCS_SERVICE_CREATE</code>), <code>update</code> (<code>ncs.dp.NCS_SERVICE_UPDATE</code>), or <code>delete</code> (<code>ncs.dp.NCS_SERVICE_DELETE</code>) of the service instance.
kp	A <code>HKeypathRef</code> object with a key path of the affected service instance, such as <code>/svc:my-service{instance1}</code> .
root	A Maagic node for the root of the data model.
service	A Maagic node for the service instance.
proplist	Opaque service properties, see the section called “Persistent Opaque Data” .

Example 30. Service callback signatures in Java

```

@ServiceCallback(servicePoint = "...",
                 callType = ServiceCBType.PRE_MODIFICATION)
public Properties preModification(ServiceContext context,
                                  ServiceOperationType operation,
                                  ConfPath path,
                                  Properties opaque)
throws DpCallbackException;

@ServiceCallback(servicePoint= "...",
                 callType=ServiceCBType.CREATE)
public Properties create(ServiceContext context,
                        NavuNode service,
                        NavuNode ncsRoot,
                        Properties opaque)
throws DpCallbackException;

@ServiceCallback(servicePoint = "...",
                 callType = ServiceCBType.POST_MODIFICATION)
public Properties postModification(ServiceContext context,
                                   ServiceOperationType operation,
                                   ConfPath path,
                                   Properties opaque)
throws DpCallbackException;

```

The Java callbacks use the following function arguments:

context	A ServiceContext object for accessing root and service instance NavuNode in the current transaction.
operation	ServiceOperationType enum representing operation: CREATE, UPDATE, DELETE of the service instance.
path	A ConfPath object with a key path of the affected service instance, such as /svc:my-service{instance1}.
ncsRoot	A NavuNode for the root of the ncs data model.
service	A NavuNode for the service instance.
opaque	Opaque service properties, see the section called “Persistent Opaque Data” .

See `examples.ncs/development-guide/services/post-modification-py` and `examples.ncs/development-guide/services/post-modification-java` examples for a sample implementation of the *post-modification* callback.

Additionally, you may implement these callbacks with templates. Refer to [the section called “Service callpoints and templates”](#) for details.

Persistent Opaque Data

FASTMAP greatly simplifies service code, so it usually only needs to deal with the initial mapping. NSO achieves this by first discarding all the configuration performed during the *create* callback of the previous run. In other words, the service create code always starts anew, with a blank slate.

If you need to keep some private service data across runs of the create callback, or pass data between callbacks, such as pre- and post-modification, you can use *opaque properties*.

The opaque object is available in the service callbacks as an argument, typically named `proplist` (Python) or `opaque` (Java). It contains a set of named properties with their corresponding values.

If you wish to use the opaque properties, it is crucial that your code returns the properties object from the create call, otherwise, the service machinery will not save the new version.

Compared to pre- and post-modification callbacks, which also persist data outside of FASTMAP, NSO deletes the opaque data when the service instance is deleted, unlike with the pre- and post-modification data.

Example 31. Using proplist in Python

```
@Service.create
def cb_create(self, tctx, root, service, proplist):
    intf = None
    # proplist is of type list[tuple[str, str]]
    for pname, pvalue in proplist:
        if pname == 'INTERFACE':
            intf = pvalue

        if intf is None:
            intf = '...'
            proplist.append('INTERFACE', intf)

    return proplist
```

Example 32. Using opaque in Java

```
public Properties create(ServiceContext context,
                        NavuNode service,
                        NavuNode ncsRoot,
                        Properties opaque)
                        throws DpCallbackException {
    // In Java API, opaque is null when service instance is first created.
    if (opaque == null) {
        opaque = new Properties();
    }
    String intf = opaque.getProperty("INTERFACE");
    if (intf == null) {
        intf = "...";
        opaque.setProperty("INTERFACE", intf);
    }

    return opaque;
}
```

The examples.ncs/development-guide/services/post-modification-py and examples.ncs/development-guide/services/post-modification-java examples showcase the use of opaque properties.

Defining Static Service Conflicts

NSO by default enables concurrent scheduling and execution of services to maximize throughput. However, concurrent execution can be problematic for non-thread-safe services or services that are known to always conflict with themselves or other services, such as when they read and write the same shared data. See [Chapter 24, NSO Concurrency Model](#) for details.

To prevent NSO from scheduling a service instance together with an instance of another service, declare a static conflict in the service model, using the ncs:conflicts-with extension. The following example

shows a service with two declared static conflicts, one with itself and one with another service, named `other-service`.

Example 33. Service with declared static conflicts

```
list example-service {
    key name;
    leaf name {
        type string;
    }
    uses ncs:service-data;
    ncs:servicepoint example-service {
        ncs:conflicts-with example-service;
        ncs:conflicts-with other-service;
    }
}
```

This means each service instance will wait for other service instances that have started sooner than this one (and are of `example-service` or `other-service` type) to finish before proceeding.

Reference Counting Overlapping Configuration

FASTMAP knows that a particular piece of configuration belongs to a service instance, allowing NSO to revert the change as needed. But 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 naively checks for existence and creates the resource when it is missing, then the resource will be tracked with the first service instance only. If, later on, this first instance is removed, then the shared resource is also removed, affecting all other instances.

A well-known solution to this kind of problem is reference counting. NSO uses reference counting by default with the XML templates and Python Maagic API, while in Java Maapi and Navu APIs, the `sharedCreate()`, `sharedSet()`, and `sharedSetValues()` functions need to be used.

When enabled, the reference counter allows FASTMAP algorithm to 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()` and `sharedSetValues()` functions also gets an additional attribute called *backpointer*. Backpointer points back to the service instance that created the entity in the first place. This makes it possible to look at part of the configuration, say under `/devices` tree, and answer the question: which parts of the device configuration were created by which service?

To see reference counting in action, start the `examples.ncs/implement-a-service/iface-v3` example with **make demo** and configure a service instance.

```
admin@ncs(config)# iface instance1 device c1 interface 0/1 ip-address 10.1.2.3 cidr-netmask 28
admin@ncs(config)# commit
```

Then configure another service instance with the same parameters and use the `display service-meta-data` pipe to show the reference counts and backpointers:

```
admin@ncs(config)# iface instance2 device c1 interface 0/1 ip-address 10.1.2.3 cidr-netmask 28
admin@ncs(config)# commit dry-run
cli {
    local-node {
        data +iface instance2 {
            +    device c1;
```

```

        +
        +      interface 0/1;
        +      ip-address 10.1.2.3;
        +      cidr-netmask 28;
        +}
    }
}

admin@ncs(config)# commit and-quit
admin@ncs# show running-config devices device c1 config interface\
  GigabitEthernet 0/1 | display service-meta-data
devices device c1
config
! Refcount: 2
! Backpointer: [ /iface:iface[iface:name='instance1'] /iface:iface[iface:name='instance2']
interface GigabitEthernet0/1
! Refcount: 2
ip address 10.1.2.3 255.255.255.240
! Refcount: 2
! Backpointer: [ /iface:iface[iface:name='instance1'] /iface:iface[iface:name='instance2']
ip dhcp snooping trust
exit
!
!
```

Notice how **commit dry-run** produces no new device configuration but the system still tracks the changes. If you wish, remove the first instance and verify the `GigabitEthernet 0/1` configuration is still there, but is gone when you also remove the second one.

But what happens if the two services produce different configuration for the same node? Say, one sets the IP address to `10.1.2.3` and the other to `10.1.2.4`. Conceptually, these two services are incompatible and instantiating both at the same time produces broken configuration (instantiating the second service instance breaks the configuration for the first). What is worse is that the current configuration depends on the order the services were deployed or re-deployed. For example, re-deploying the first service will change the configuration from `10.1.2.4` back to `10.1.2.3` and vice versa. Such inconsistencies break the declarative configuration model and really should be avoided.

In practice, however, NSO does not prevent services from producing such configuration. But note that we strongly recommend against it and that there are associated limitations, such as service un-deploy not reverting configuration to that produced by the other instance (but when all services are removed, the original configuration is still restored).

Stacked Services

Much like a service in NSO can provisioning device configurations, it can also provision other, non-device data, as well as other services. We call the approach of services provisioning other services *service stacking* and the services that are involved — *stacked*.

Service stacking concepts usually come into play for bigger, more complex services. There are a number of reasons why you might prefer stacked services to a single monolithic one:

- Smaller, more manageable services with simpler logic
- Separation of concerns and responsibility
- Clearer ownership across teams for (parts of) overall service
- Smaller services reusable as components across the solution
- Avoiding overlapping configuration between service instances causing conflicts, such as using one service instance per device (see examples in the section called “[Designing for Maximal Transaction Throughput](#)”)

Stacked services are also the basis for LSA, which takes this concept even further. See Layered Service Architecture for details.

The standard naming convention with stacked services distinguishes between a Resource-Facing Service (RFS), that directly configures one or more devices, and a Customer-Facing Service (CFS), that is the top-level service, configuring only other services, not devices. There can be more than two layers of services in the stack, too.

While NSO does not prevent a single service from configuring devices as well as services, in the majority of cases this results in a less clean design and is best avoided.

Overall, creating stacked services is very similar to the non-stacked approach. First, you can design the RFS services as usual. Actually, you might take existing services and reuse those. These then become your lower-level services, since they are lower in the stack.

Then you create a higher-level service, say a CFS, that configures another service, or a few, instead of a device. You can even use a template-only service to do that, such as:

Example 34. Template for configuring another service (stacking)

```
<config-template xmlns="http://tail-f.com/ns/config/1.0"
    servicepoint="top-level-service">
    <iface xmlns="http://com/example/iface">
        <name>instance1</name>
        <device>c1</device>
        <interface>0/1</interface>
        <ip-address>10.1.2.3</ip-address>
        <cidr-netmask>28</cidr-netmask>
    </iface>
</config>
```

The preceding example references an existing *iface* service, such as the one in the `examples.ncs/implement-a-service/iface-v3` example. The output shows hard-coded values but you can change those as you would for any other service.

In practice, you might find it beneficial to modularize your data model and potentially reuse parts in both, the lower- and higher-level service. This avoids duplication while still allowing you to directly expose some of the lower-level service functionality through the higher-level model.

The most important principle to keep in mind is that the data created by any service is owned by that service, regardless of how the mapping is done (through code or templates). If the user deletes a service instance, FASTMAP will automatically delete whatever the service created, including any other services. Likewise, if the operator directly manipulates service data that is created by another service, the higher-level service becomes out of sync. The **check-sync** service action checks this for services as well as devices.

In stacked service design, the lower-level service data is under the control of the higher-level service and must not be directly manipulated. Only the higher-level service may manipulate that data. However, two higher-level services may manipulate the same structures, since NSO performs reference counting (see [the section called “Reference Counting Overlapping Configuration”](#)).

Caveats and Best Practices

This section lists some specific advice for implementing services, as well as any known limitations you might run into.

You may also obtain some useful information by using the **debug service** commit pipe command, such as **commit dry-run | debug service**. The command display the net effect of the service create code, as well as issue warnings about potentially problematic usage of overlapping shared data.

- **Service callbacks must be deterministic**

NSO invokes service callbacks in a number of situations, such as for dry-run, check sync, and actual provisioning. If a service does not create the same configuration from the same inputs, NSO sees it as being out of sync, resulting in a lot of configuration churn and making it incompatible with many NSO features.

If you need to introduce some randomness or rely on some other nondeterministic source of data, make sure to cache the values across callback invocations, such as by using opaque properties ([the section called “Persistent Opaque Data”](#)) or persistent operational data ([the section called “Operational Data”](#)) populated in a *pre-modification* callback.

- **Never overwrite service inputs**

Service input parameters capture client intent and a service should never change its own configuration. Such behavior not only muddles the intent but is also temporary when done in the create callback, as the changes are reverted on the next invocation.

If you need to keep some additional data that cannot be easily computed each time, consider using opaque properties ([the section called “Persistent Opaque Data”](#)) or persistent operational data ([the section called “Operational Data”](#)) populated in a *pre-modification* callback.

- **No service ordering in a transaction**

NSO is a transactional system and as such does not have the concept of order inside a single transaction. That means NSO does not guarantee any specific order in which the service mapping code executes if the same transaction touches multiple service instances. Likewise, your code should not make any assumptions about running before or after other service code.

- **Return value of create callback**

The *create* callback is not the exclusive user of the opaque object; the object can be chained in several different callbacks, such as *pre-* and *post-modification*. Therefore, returning `None`/`null` from create callback is not a good practice. Instead, always return the opaque object even if the create callback does not use it.

- **Avoid delete in service create**

Unlike creation, deleting configuration does not support reference counting, as there is no data left to reference count. This means the deleted elements are tied to the service instance that deleted them.

Additionally, FASTMAP must store the entire deleted tree and restore it on every service change or re-deploy, only to be deleted again. Depending on the amount of deleted data, this is potentially an expensive operation.

So, a general rule of thumb is to never use delete in service create code. If an explicit delete is used, **debug service** may display the following warning:

```
*** WARNING ***: delete in service create code is unsafe if data is  
shared by other services
```

However, service may also delete data implicitly, through `when` and `choice` statements in the YANG data model. If a `when` statement evaluates to false, the configuration tree below that node is deleted. Likewise, if a `case` is set in a `choice` statement, the previously set `case` is deleted. This has the same limitations as an explicit delete.

To avoid these issues, create a separate service, which only handles deletion, and use it in the main service through the stacked service design (see [the section called “Stacked Services”](#)). This approach

allows you to reference count the deletion operation and contain the effect of restoring deleted data through a small, rarely-changing helper service. See `examples.ncs/development-guide/services/shared-delete` for an example.

Alternatively, you might consider *pre-* and *post-modification* callbacks for some specific cases.

- **Prefer `shared*`() functions**

Non-shared create and set operations in the Java and Python low-level API do not add reference counts or backpointer information to changed elements. In case there is overlap with another service, unwanted removal can occur. See [the section called “Reference Counting Overlapping Configuration”](#) for details.

In general, you should prefer `sharedCreate()`, `sharedSet()`, and `sharedSetValues()`. If non-shared variants are used in a shared context, **service debug** displays a warning, such as:

```
*** WARNING ***: set in service create code is unsafe if data is
shared by other services
```

Likewise, do not use MAAPI `load_config` variants from service code. Use the `sharedSetValues()` function to load XML data from a file or a string.

- **Reordering ordered-by-user lists**

If service code rearranges an ordered-by-user list with items that were created by another service, that other service becomes out of sync. In some cases you might be able to avoid out-of-sync scenario by leveraging special XML template syntax (see [the section called “Operations on ordered lists and leaf-lists”](#)) or using service stacking with a helper service.

In general, however, you should reconsider your design and try to avoid such scenarios.

- **Automatic upgrade of keys for existing services is unsupported**

Service backpointers, described in [the section called “Reference Counting Overlapping Configuration”](#), rely on the keys that the service model defines to identify individual service instances. If you update the model by adding, removing, or changing the type of leafs used in the service list key, while there are deployed service instances, the backpointers will not be automatically updated. Therefore, it is best to not change the service list key.

A workaround, if service key absolutely must change, is to first perform a no-networking undeploy of the affected service instances, then upgrade the model, and finally no-networking re-deploy the previously un-deployed services.

- **Avoid conflicting intents**

Consider that a service is executed as part of a transaction. If, in the same transaction, the service gets conflicting intents, for example, it gets modified and deleted, the transaction is aborted. You must decide which intent has higher priority and design your services to avoid such situations.

Service Discovery and Import

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. To take full advantage of the new system, you should consider importing the existing services into NSO. The goal is to use NSO to manage existing service instances, along with adding new ones in the future.

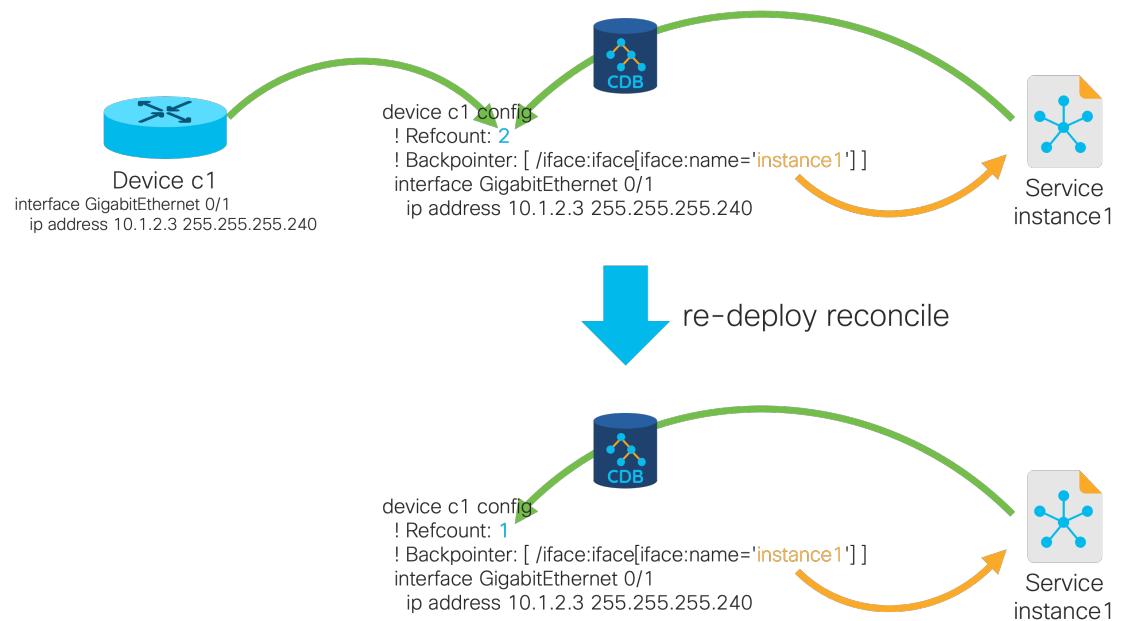
The process of identifying services and importing them into NSO is called *Service Discovery* and can be broken down into the following high-level parts:

- Implementing the service to match existing device configuration.

- Enumerating service instances and their parameters.
- Amend the service meta data references with reconciliation.

Ultimately, the problem that service discovery addresses is one of referencing or linking configuration to services. Since the network already contains target configuration, a new service instance in NSO produces no changes in the network. This means the new service in NSO by default does not own the network configuration. One side effect is that removing a service will not remove the corresponding device configuration, which is likely to interfere with service modification as well.

Figure 35. Service reconciliation



Some of the steps in the process can be automated, while others are mostly manual. The amount of work differs a lot depending on how structured and consistent the original deployment is.

Matching configuration

A prerequisite (or possibly the product in an iterative approach) is an NSO service that supports all the different variants of the configuration for the service that are used in the network. This usually means there will be a few additional parameters in the service model that allow selecting the variant of device configuration produced, as well as some covering other non-standard configuration (if such configuration is present).

In the simplest case, there is only one variant and that is the one that the service needs to produce. Let's take the `examples.ncs/implement-a-service iface-v2-py` example and consider what happens when a device already has existing interface configuration.

```
admin@ncs# show running-config devices device c1 config\
  interface GigabitEthernet 0/1
devices device c1
config
  interface GigabitEthernet0/1
    ip address 10.1.2.3 255.255.255.240
exit
```

```
!
!
```

Configuring a new service instance does not produce any new device configuration (notice that device c1 has no changes).

```
admin@ncs(config)# commit dry-run
cli {
    local-node {
        data +iface instance1 {
            +   device c1;
            +   interface 0/1;
            +   ip-address 10.1.2.3;
            +   cidr-netmask 28;
            +
        }
    }
}
```

However, when committed, NSO records the changes, just like in the case of overlapping configuration ([the section called “Reference Counting Overlapping Configuration”](#)). The main difference is that there is only a single backpointer, to a newly configured service, but the refcount is 2. The other item, that contributes to the refcount, is the original device configuration. Which is why the configuration is not deleted when the service instance is.

```
admin@ncs# show running-config devices device c1 config interface\
  GigabitEthernet 0/1 | display service-meta-data
devices device c1
config
  ! Refcount: 2
  ! Backpointer: [ /iface:iface[iface:name='instance1'] ]
interface GigabitEthernet0/1
  ! Refcount: 2
  ! Originalvalue: 10.1.2.3
  ip address 10.1.2.3 255.255.255.240
exit
!
!
```

Enumerating instances

A prerequisite for service discovery to work is that it is possible to construct a list of the already existing services. Such a list may exist in an inventory system, an external database, or perhaps just an Excel spreadsheet.

You can import the list of services in a number of ways. If you are reading it in from a spreadsheet, a Python script using NSO API directly ([Chapter 4, Basic Automation with Python](#)) and a module to read Excel files is likely a good choice.

Example 36. Sample service Excel import script

```
import ncs
from openpyxl import load_workbook

def main():
    wb = load_workbook('services.xlsx')
    sheet = wb[wb.sheetnames[0]]

    with ncs.maapi.single_write_trans('admin', 'python') as t:
        root = ncs.maagic.get_root(t)
        for sr in sheet.rows:
```

```

# Suppose columns in spreadsheet are:
# instance (A), device (B), interface (C), IP (D), mask (E)
name = sr[0].value
service = root.iface.create(name)
service.device = sr[1].value
service.interface = sr[2].value
service.ip_address = sr[3].value
service.cidr_netmask = sr[4].value

t.apply()

main()

```

Or you might generate an XML data file to import using the **ncs_load** command; use **display xml** filter to help you create a template:

```

admin@ncs# show running-config iface | display xml
<config xmlns="http://tail-f.com/ns/config/1.0">
  <iface xmlns="http://com/example/iface">
    <name>instance1</name>
    <device>c1</device>
    <interface>0/1</interface>
    <ip-address>10.1.2.3</ip-address>
    <cidr-netmask>28</cidr-netmask>
  </iface>
</config>

```

Regardless of the way you implement the data import, you can run into two kinds of problems.

On one hand, the service list data may be incomplete. Suppose that the earliest service instances deployed did not take the network mask as a parameter. Moreover, for some specific reasons, a number of interfaces had to deviate from the default of 28 and that information was never populated back in the inventory for old services after the netmask parameter was added.

Now the only place where that information is still kept may be the actual device configuration. Fortunately, you can access it through NSO, which may allow you to extract the missing data automatically, for example:

```

devconfig = root.devices.device[service.device].config
intf = devconfig.interface.GigabitEthernet[service.interface]
netmask = intf.ip.address.primary.mask
cidr = IPv4Network(f'0.0.0.0/{netmask}').prefixlen

```

On the other hand, some parameters may be NSO specific, such as those controlling which variant of configuration to produce. Again, you might be able to use a script to find this information, or it could turn out that the configuration is too complex to make such a script feasible.

In general, this can be the most tricky part of the service discovery process, making it very hard to automate. It all comes down to how good the existing data is. Keep in mind that this exercise is typically also a cleanup exercise, and every network will be different.

Reconciliation

The last step is updating the meta data, telling NSO that a given service controls (owns) the device configuration that was already present when the NSO service was configured. This is called reconciliation and you achieve it using a special **re-deploy reconcile** action for the service.

Let's examine the effects of this action on the following data:

```
admin@ncs# show running-config devices device c1 config\
```

```

interface GigabitEthernet 0/1 | display service-meta-data
devices device c1
config
! Refcount: 2
! Backpointer: [ /iface:iface[iface:name='instance1'] ]
interface GigabitEthernet0/1
! Refcount: 2
! Originalvalue: 10.1.2.3
ip address 10.1.2.3 255.255.255.240
exit
!
!
```

Having run the action, NSO has updated the refcount to remove the reference to the original device configuration:

```

admin@ncs# iface instance1 re-deploy reconcile
admin@ncs# show running-config devices device c1 config\
  interface GigabitEthernet 0/1 | display service-meta-data
devices device c1
config
! Refcount: 1
! Backpointer: [ /iface:iface[iface:name='instance1'] ]
interface GigabitEthernet0/1
! Refcount: 1
ip address 10.1.2.3 255.255.255.240
exit
!
!
```

What is more, the reconcile algorithm works even if multiple service instances share configuration. What if you had two instances of the iface service, instead of one?

Before reconciliation the device configuration would show a refcount of three.

```

admin@ncs# show running-config devices device c1 config\
  interface GigabitEthernet 0/1 | display service-meta-data
devices device c1
config
! Refcount: 3
! Backpointer: [ /iface:iface[iface:name='instance1'] /iface:iface[iface:name='instance2'] ]
interface GigabitEthernet0/1
! Refcount: 3
! Originalvalue: 10.1.2.3
ip address 10.1.2.3 255.255.255.240
exit
!
!
```

Invoking **re-deploy reconcile** on either one or both of the instances makes the services sole owners of the configuration.

```

admin@ncs# show running-config devices device c1 config\
  interface GigabitEthernet 0/1 | display service-meta-data
devices device c1
config
! Refcount: 2
! Backpointer: [ /iface:iface[iface:name='instance1'] /iface:iface[iface:name='instance2'] ]
interface GigabitEthernet0/1
! Refcount: 2
ip address 10.1.2.3 255.255.255.240
exit
!
```

```
!
!
```

This means the device configuration is removed only when you remove both service instances.

```
admin@ncs(config)# no iface instance1
admin@ncs(config)# commit dry-run outformat native
native {
}
admin@ncs(config)# no iface instance2
admin@ncs(config)# commit dry-run outformat native
native {
    device {
        name c1
        data no interface GigabitEthernet0/1
    }
}
```

The reconcile operation only removes the references to the original configuration (without the service backpointer), so you can execute it as many times as you wish. Just note that it is part of a service re-deploy, with all the implications that brings, such as potentially deploying new configuration to devices when you change the service template.

As an alternative to the **re-deploy reconcile**, you can initially add the service configuration with a **commit reconcile** variant, performing reconciliation right away.

Iterative approach

It is hard to design a service in one go when you wish to cover existing configurations that are exceedingly complex or have a lot of variance. In such cases, many prefer an iterative approach, where you tackle the problem piece-by-piece.

Suppose there are two variants of the service configured in the network; *iface-v2-py* and the newer *iface-v3*, which produces a slightly different configuration. This is a typical scenario when a different (non-NSO) automation system is used and the service gradually evolves over time. Or when a Method of Procedure (MOP) is updated if manual provisioning is used.

We will tackle this scenario to show how you might perform service discovery in an iterative fashion. We shall start with the *iface-v2-py* as the first iteration of the *iface* service, which represents what configuration the service should produce to the best of our current knowledge.

There are configurations for two service instances in the network already: for interfaces 0/1 and 0/2 on the *c1* device. So, configure the two corresponding *iface* instances.

```
admin@ncs(config)# commit dry-run
cli {
    local-node {
        data +iface instance1 {
            + device c1;
            + interface 0/1;
            + ip-address 10.1.2.3;
            + cidr-netmask 28;
        }
        +iface instance2 {
            + device c1;
            + interface 0/2;
            + ip-address 10.2.2.3;
            + cidr-netmask 28;
        }
    }
}
```

```

}
admin@ncs(config)# commit

```

You can also use the **commit no-deploy** variant to add service parameters when normal commit would produce device changes, which you do not want.

Then use the **re-deploy reconcile { discard-non-service-config } dry-run** command to observe the difference between the service-produced configuration and the one present in the network.

```

admin@ncs# iface instance1 re-deploy reconcile\
  { discard-non-service-config } dry-run
cli {
}

```

For *instance1* the config is the same, so you can safely reconcile it already.

```
admin@ncs# iface instance1 re-deploy reconcile
```

But interface 0/2 (*instance2*), which you suspect was initially provisioned with the newer version of the service, produces the following:

```

admin@ncs# iface instance2 re-deploy reconcile\
  { discard-non-service-config } dry-run
cli {
  local-node {
    data devices {
      device c1 {
        config {
          interface {
            GigabitEthernet 0/2 {
              ip {
                dhcp {
                  snooping {
                    trust;
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}

```

The output tells you that the service is missing the **ip dhcp snooping trust** part of the interface configuration. Since the service does not generate this part of the configuration yet, running **re-deploy reconcile { discard-non-service-config }** (without dry-run) would remove the DHCP trust setting. This is not what we want.

One option, and this is the default reconcile mode, would be to use **keep-non-service-config** instead of **discard-non-service-config**. But that would result in the service taking ownership of only part of the interface configuration (the IP address).

Instead, the right approach is to add the missing part to the service template. There is, however, a little problem. Adding the DHCP snooping trust configuration unconditionally to the template can interfere with the other service instance, *instance1*.

In some cases, upgrading the old configuration to the new variant is viable, but in most situations you likely want to avoid all device configuration changes. For the latter case, you need to add another

parameter to the service model that selects the configuration variant. You must update the template too, producing the second iteration of the service.

```
iface instance2
device      c1
interface   0/2
ip-address  10.2.2.3
cidr-netmask 28
variant     v3
!
```

With the updated configuration, you can now safely reconcile the *service2* service instance:

```
admin@ncs# iface instance2 re-deploy reconcile \
{ discard-non-service-config } dry-run
cli {
}
admin@ncs# iface instance2 re-deploy reconcile
```

Nevertheless, keep in mind that the `discard-non-service-config` reconcile operation only considers parts of the device configuration under nodes that are created with the service mapping. Even if all data there is covered in the mapping, there could still be other parts that belong to the service but reside in an entirely different section of the device configuration (say DNS configuration under `ip name-server`, which is outside the `interface GigabitEthernet` part) or even a different device. That kind of configuration the `discard-non-service-config` option cannot find on its own and you must add manually.

You can find the complete `iface` service as part of the `examples.ncs/development-guide/services/discovery` example.

Since there were only two service instances to reconcile, the process is now complete. In practice, you are likely to encounter multiple variants and many more service instances, requiring you to make additional iterations. But you can follow the iterative process shown here.

Partial Sync

In some cases a service may need to rely on the actual device configurations to compute the changeset. It is often a requirement to pull the current device 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. The alternative 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 device's 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` clear the transaction ID of the device in NSO unless the whole configuration has been selected (e.g. `/ncs:devices/ncs:device[ncs:name='ex0']/ncs:config`). This ensures NSO does not miss any changes to other parts of the device configuration but it does make the device out of sync.

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 37, “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 37. 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 10

The NSO Java VM

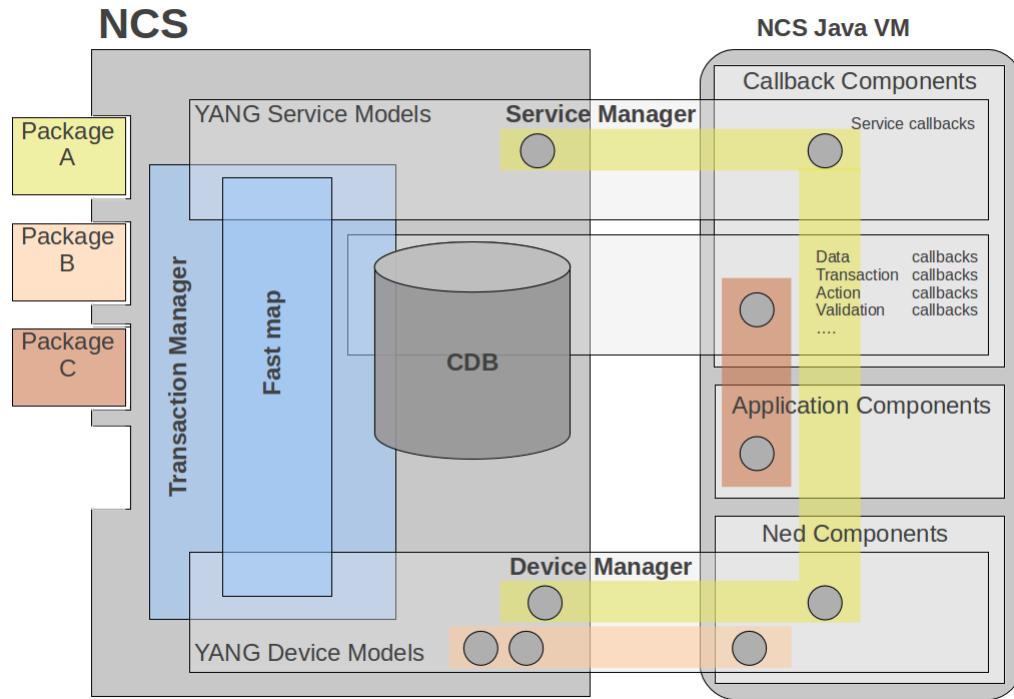
- [Overview, page 137](#)
- [YANG model, page 139](#)
- [Java Packages and the Class Loader, page 139](#)
- [The NED component type, page 141](#)
- [The callback component type, page 141](#)
- [The application component type, page 141](#)
- [The Resource Manager, page 142](#)
- [The Alarm centrals, page 144](#)
- [Embedding the NSO Java VM, page 144](#)
- [Logging, page 145](#)
- [The NSO Java VM timeouts, page 145](#)
- [Debugging Startup, page 146](#)

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 38. 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 39, “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 39. 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:

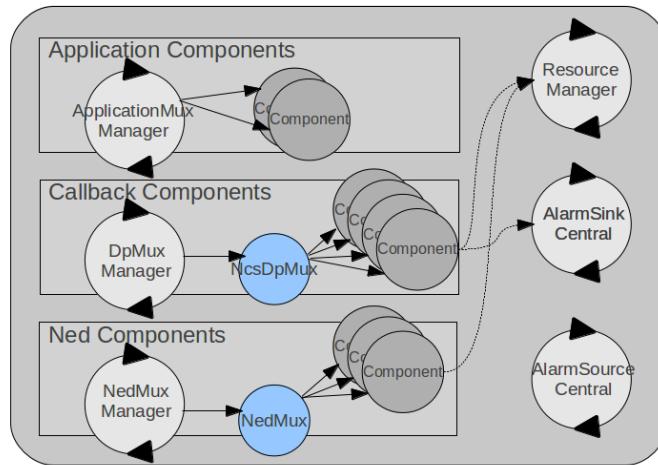
- The *ned* type. The NSO Java VM will handle NEDs of sub type *cli* and *generic* which are the ones that have a java implementation.
- The *callback* type. These are any forms of callbacks that defined by the Dp API.
- The *application* type. These are user defined daemons that implements a specific `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 40. 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 20, NED Development](#) and [Chapter 20, 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 41. 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 26, 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 42. 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 43. 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 44. 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 45. 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 46. 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 8, *Alarm Manager* in *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 47. 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.

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 *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.

failed	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:

not-connected	The Java application code is not connected to NSO.
initializing	The Java application code is connected to NSO, but not yet initialized.
running	The Java application code is connected and initialized.
timeout	The Java application connected to NSO, but failed to initialize within the stipulated timeout 'initialization-time'.



CHAPTER 11

The NSO Python VM

- [Introduction, page 149](#)
- [YANG model, page 149](#)
- [Structure of the User provided code, page 151](#)
- [Debugging of Python packages, page 155](#)
- [Using non-standard Python, page 155](#)
- [Caveats, page 156](#)

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 48, “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)#{
```

The `sanity-checks/self-assign-warning` control the self-assignment warnings for Python services with off, log, and alarm (default) modes. An example of a self-assignment:

```
class ServiceCallbacks(Service):
    @Service.create
    def cb_create(self, tctx, root, service, proplist):
        self.counter = 42
```

As several service invocations may run in parallel, self-assignment will likely cause difficult to debug issues. An alarm or a log entry will contain a warning and a keypath to the service instance that caused the warning. Example log entry:

```
<WARNING> ... Assigning to self is not thread safe: /mysrvc:mysrvc{2}
```

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 48. 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 sanity-checks
      |   +-rw self-assign-warning? enumeration
```

```

    ---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 49. `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 50. 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*.

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

The *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, as a separate worker process will be created for each Service, Nano Service and Action.

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 51. 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 52. 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
    maagic 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
# be 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
/usr/local/nso/bin
$ cp ncs-start-python-vm my-start-python-vm
$ # Use your favourite editor to update the last lines of the new
$ # file to start the desired Python executable.
```

And add the following snippet to ncs.conf:

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

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

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.

Caveats

Using multiprocessing

Using the multiprocessing library from Python components where the *callpoint-model* is set to *threading* can cause unexpected disconnects from NSO if errors occur in the code executed by the multiprocessing library.

To work around this, either use *multiprocessing* as the *callpoint-model* or force the start method to be *spawn* by executing:

Example 53. Set start method to spawn

```
if multiprocessing.get_start_method() != 'spawn':
    multiprocessing.set_start_method('spawn', force=True)
```



CHAPTER 12

Embedded Erlang applications

- [Introduction, page 157](#)
- [Erlang API, page 157](#)
- [Application configuration, page 158](#)
- [Example, page 159](#)

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 13

The YANG Data Modeling Language

- [The YANG Data Modeling Language, page 161](#)
- [YANG in NSO, page 161](#)
- [YANG Introduction, page 162](#)
- [Working With YANG Modules, page 169](#)
- [Integrity Constraints, page 171](#)
- [The when statement, page 172](#)
- [Using the Tail-f Extensions with YANG, page 173](#)
- [Custom Help Texts and Error Messages, page 175](#)
- [An Example: Modeling a List of Interfaces, page 176](#)
- [More on leafrefs, page 184](#)
- [Using Multiple Namespaces, page 185](#)
- [Module Names, Namespaces and Revisions, page 186](#)
- [Hash Values and the id-value Statement, page 187](#)
- [NSO caveats, page 188](#)

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>.

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 54. YANG built-in types

Name	Type	Description
binary	Text	Any binary data
bits	Text/Number	A set of bits or flags
boolean	Text	"true" or "false"
decimal64	Number	64-bit fixed point real number
empty	Empty	A leaf that does not have any value.

Name	Type	Description
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-area {
            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 *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 *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 *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 *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 *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 *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 *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>
        </link>
    </links>
</config>
```

```

<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 *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, `confd` 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 14

Using CDB

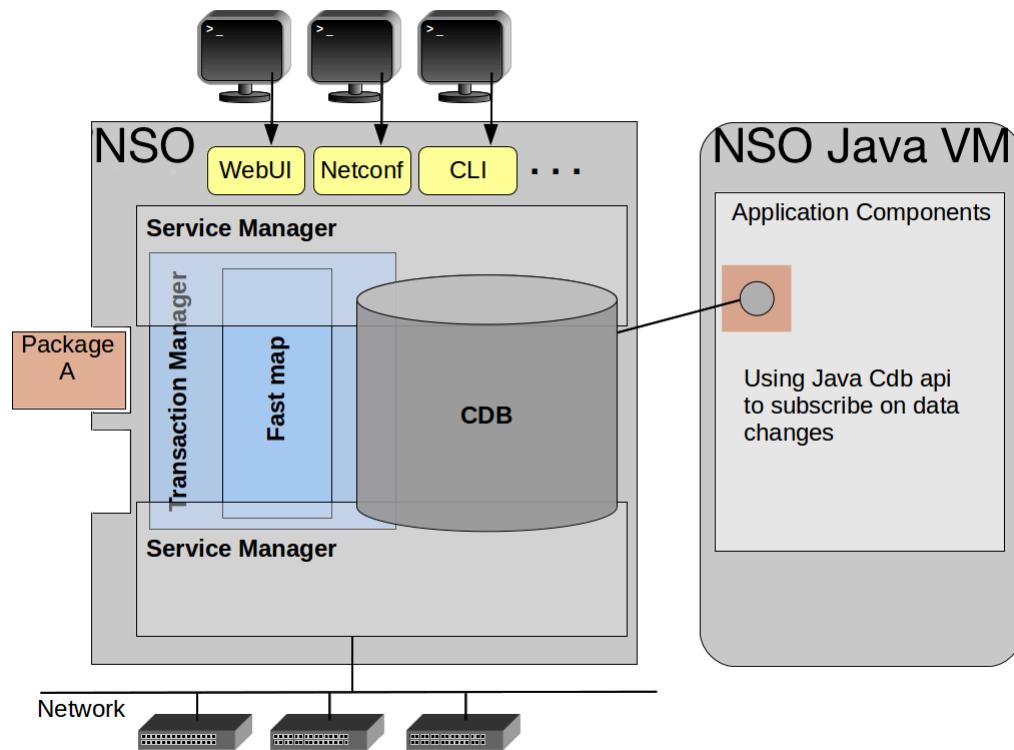
- [Introduction, page 191](#)
- [The NSO Data Model, page 192](#)
- [Addressing Data Using Keypaths, page 194](#)
- [Subscriptions, page 195](#)
- [Sessions, page 196](#)
- [Loading Initial Data Into CDB, page 197](#)
- [Operational Data in CDB, page 198](#)
- [Example, page 199](#)
- [Automatic Schema Upgrades and Downgrades, page 210](#)
- [Using Initialization Files for Upgrade, page 212](#)
- [New Validation Points, page 215](#)
- [Writing an Upgrade Package Component, page 215](#)

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 55, “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 55. 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 56. 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 57. 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 57, “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 /13vpn: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 `13vpn: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 when the 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 58, “CDB Init”](#)), CDB will try to initialize the database from all XML documents found in the same directory.

Example 58. 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 57, “L3 VPN YANG Extract”](#) the initial data for topology can be found in `topology.xml` as seen in [Example 59, “Initial Data for Topology”](#)

Example 59. 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 *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 *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 60, “Wrapper for Multiple Top Level Tags”](#) like this:

Example 60. 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 `examples.ncs/getting-started/developing-with-ncs/0-router-network` 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:

- *Plain CDB Subscriber* - This `cdb` subscriber subscribes to changes under the path `/devices/device{ex0}/config`. Whenever a change occurs there, the code iterates through the change and prints the values.
- *CdbCfgSubscriber* - An more advanced CDB subscriber that subscribes to changes under the path `/devices/device/config/sys/interfaces/interface`.
- *OperSubscriber* - An operational data subscribe that subscribes to changes under the path `/t:test/stats-item`.

The `cdb` package include the YANG shown in [Example 61, “1-cdb Simple Config Data”](#).

Example 61. 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 {
            config false;
            tailf:cdb-oper;
```

Example

```
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 62. Plain CDB Subscriber Java Code

```
public class PlainCdbSub implements ApplicationComponent {
    private static final 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;

    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 {
            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 oldValue,
                                         ConfObject newValue,
                                         Object state) {
        try {
            String kpString = Conf.kpToString(kp);
            LOGGER.info("diffIterate: kp= " + kpString + ", OP=" + op
                       + ", old_value=" + oldValue + ", new_value="
                       + newValue);
            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 63, “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 63. Resource Annotation

```
@Resource(type = ResourceType.CDB, scope = Scope.INSTANCE,
           qualifier = "plain")
private Cdb cdb;
```

The `init()` method (shown in [Example 64, “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 64. 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 65, “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 65. Plain CDB Subscriber

```
public void run() {
    try {
        while (!requestStop) {
            try {
                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 66, “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 66. Plain Subscriber Iterator Implementation

```

private class Iter implements CdbDiffIterate {
    public DiffIterateResultFlag iterate(ConfObject[] kp,
                                         DiffIterateOperFlag op,
                                         ConfObject oldValue,
                                         ConfObject newValue,
                                         Object state) {
        try {
            String kpString = Conf.kpToString(kp);
            LOGGER.info("diffIterate: kp= " + kpString + ", OP=" + op
                        + ", old_value=" + oldValue + ", new_value="
                        + newValue);
            return DiffIterateResultFlag.ITER_RECURSE;
        } catch (Exception e) {
            return DiffIterateResultFlag.ITER_CONTINUE;
        }
    }
}

```

The `finish()` method ([Example 67, “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 67. Plain Subscriber finish

```

public void finish() {
    requestStop = true;
    LOGGER.warn(" PlainSub in finish () =>");
    try {
        // ResourceManager will close the resource (cdb) used by this
    }
}

```

Example

```

    // 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 68, "Plain Subscriber Startup"](#) shows how to do this.

Example 68. 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 69, "Populate Data using CLI"](#)).

Example 69. 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 70, “Plain Subscriber Output”](#) (with each row split over several to fit on the page).

Example 70. 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 71, “Run CdbCfgSubscriber Example”](#) shows how to build and run the example.

Example 71. 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 72, “Subscriber Output”](#)). In the end of this file the last DUMP DB will show only one remaining interface.

Example 72. 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
<INFO> 05-Feb-2015::16:10:23,354 ConfigCdbSub
  (cdb-examples:CdbCfgSubscriber)-Run-1: -           name: {0}
<INFO> 05-Feb-2015::16:10:23,354 ConfigCdbSub
  (cdb-examples:CdbCfgSubscriber)-Run-1: -           description: null
<INFO> 05-Feb-2015::16:10:23,355 ConfigCdbSub
  (cdb-examples:CdbCfgSubscriber)-Run-1: -           vlan-id:null
<INFO> 05-Feb-2015::16:10:23,355 ConfigCdbSub
  (cdb-examples:CdbCfgSubscriber)-Run-1: -           ADDRESS-FAMILY
<INFO> 05-Feb-2015::16:10:23,355 ConfigCdbSub
  (cdb-examples:CdbCfgSubscriber)-Run-1: -           key: {192.168.1.115}
<INFO> 05-Feb-2015::16:10:23,355 ConfigCdbSub
  (cdb-examples:CdbCfgSubscriber)-Run-1: -           prefixLength: 32
<INFO> 05-Feb-2015::16:10:23,355 ConfigCdbSub
  (cdb-examples:CdbCfgSubscriber)-Run-1: -           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 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 73, “1-cdb Simple Operational Data”](#))

Example 73. 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 74, “Creating Operational Data using Navu API”](#).

Example 74. Creating Operational Data using Navu API

```
public static void createEntry(String key)
    throws IOException, 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<>();
    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 75, “Deleting Operational Data using CDB API”](#).

Example 75. Deleting Operational Data using CDB API

```
public static void deleteEntry(String key)
    throws IOException, 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%{x}",
```

```

        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 76, "CDB Operational Subscriber Java code"](#)).

Example 76. CDB Operational Subscriber Java code

```

public class OperCdbSub implements ApplicationComponent, CdbDiffIterate {
    private static final 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;
    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[] points = cdbSubscription.read();
                    CdbSession cdbSession
                        = cdbData.startSession(CdbDBType.CDB_OPERATIONAL);
                    EnumSet<DiffIterateFlags> diffFlags
                        = EnumSet.of(DiffIterateFlags.ITER_WANT_PREV);
                    cdbSubscription.diffIterate(points[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;
        default:
            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 `deloper` 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 77, “Populating Operational Data”](#)).

Example 77. Populating Operational Data

```

$ make clean all
$ ncs
$ ./setoper eth0
$ ./setoper ethX
$ ./deloper 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 78, “Operational subscription Output”](#).

Example 78. 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 79, “Enabling Developer Logging”](#)

Example 79. 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 and the leaf will be deleted after CDB upgrade.

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 *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 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, lets 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 80, “New YANG module for the ServerManager Package”](#).

Example 80. 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.";
    }

    revision "2006-09-01" {
        description "Initial servers data model";
    }

    /* A set of server structures */
    container servers {
        list server {
            key name;
            max-elements 64;
```

```
    leaf name {
        type string;
    }
    leaf ip {
        type inet:ip-address;
        mandatory true;
    }
    leaf port {
        type inet:port-number;
        mandatory true;
    }
    leaf protocol {
        type enumeration {
            enum tcp;
            enum udp;
        }
        mandatory true;
    }
}
}
```

The differences from the earlier version of the YANG module can be seen in [Example 81, “Difference between YANG Modules”](#).

Example 81. 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 82, “Protocol Upgrade Init File”](#) like this:

Example 82. 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 83, "Configuration after Upgrade"](#). Here is what the configuration looks like after upgrade if the "smtp" server has been deleted before upgrade:

Example 83. 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 a 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.

New Validation Points

One case the system does not handle directly is addition of new custom validation points using the `tailf:validate` statement during an upgrade. The issue that surfaces is that the schema upgrade is performed before the (new) user code gets deployed and therefore the code required for validation is not yet available. It results in an error similar to “no registration found for callpoint `NEW-VALIDATION/validate`” or simply “application communication failure”.

One way to solve this problem is to first redeploy the package with the custom validation code, then perform the schema upgrade through the full **packages reload** action. For example, suppose you are upgrading the package `test-svc`. Then you first perform **packages package test-svc redeploy**, followed by **packages reload**. The main downside to this approach is that the new code must work with the old data model, which may require extra effort when there are major data model changes.

An alternative is to temporarily disable the validation by starting the NSO with the `--ignore-initial-validation` option. In this case, you should stop the ncs process and start it using `--ignore-initial-validation` and `--with-package-reload` options to perform the schema upgrade without custom validation. However, this may result in data in the CDB that would otherwise not pass custom validation. If you still want to validate the data, you can write an upgrade component to do this one-time validation.

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.

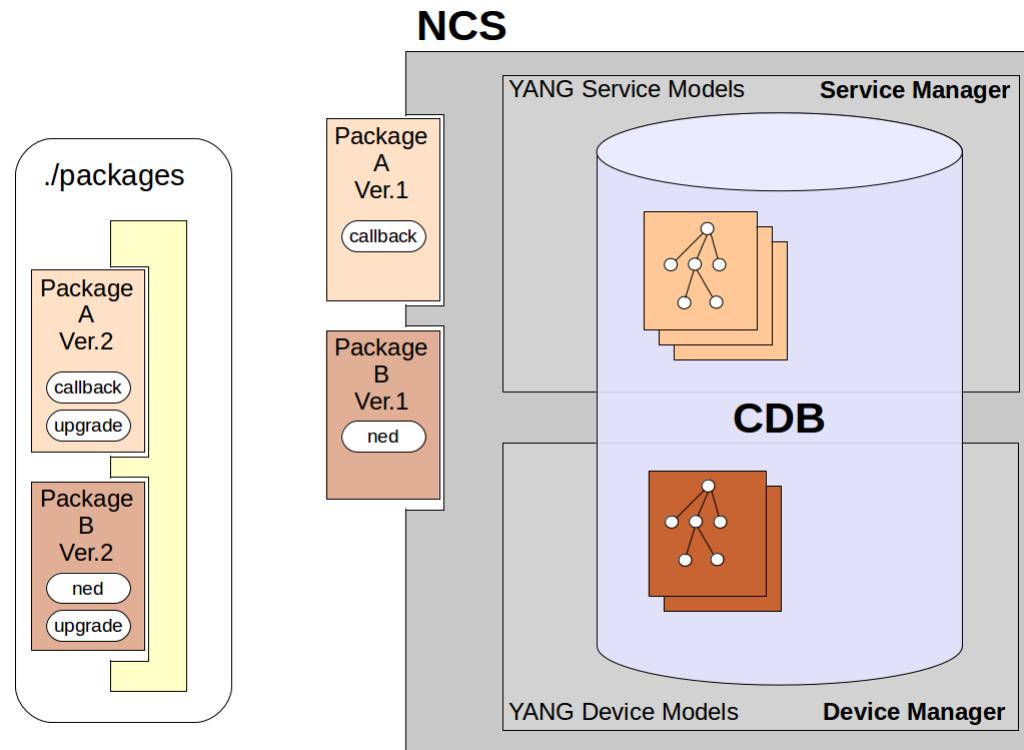
A *upgrade* component is a Java class with a standard `main()` method that becomes a standalone program that is run as part of the package *reload* action.

As with any package component types, the *upgrade* components has to be defined in the *package-metadata.xml* file for the package ([Example 84, “Upgrade Package Components”](#)).

Example 84. Upgrade Package Components

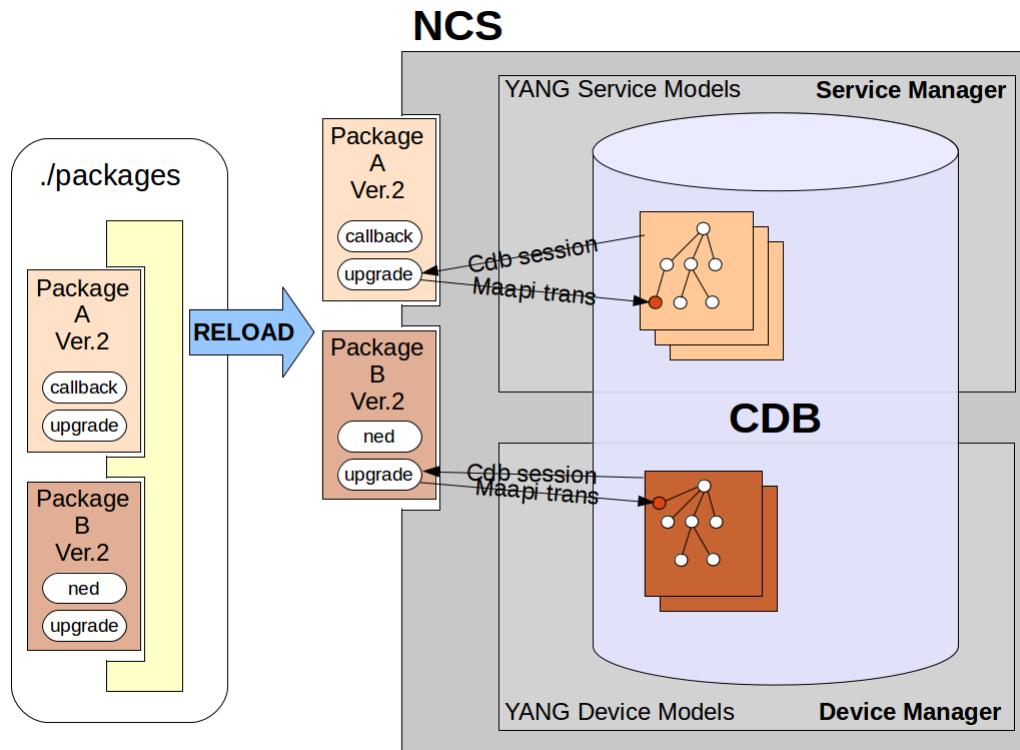
```
<ncs-package xmlns="http://tail-f.com/ns/ncs-packages">
    ...
    <component>
        <name>do-upgrade</name>
        <upgrade>
            <java-class-name>com.example.DoUpgrade</java-class-name>
        </upgrade>
    </component>
</ncs-package>
```

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 85. 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 86, “YANG Service diff”](#)).

Example 86. 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 87. 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 88, “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 88. 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 89, “Upgrade Get Transaction”](#)).

Example 89. 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);
```

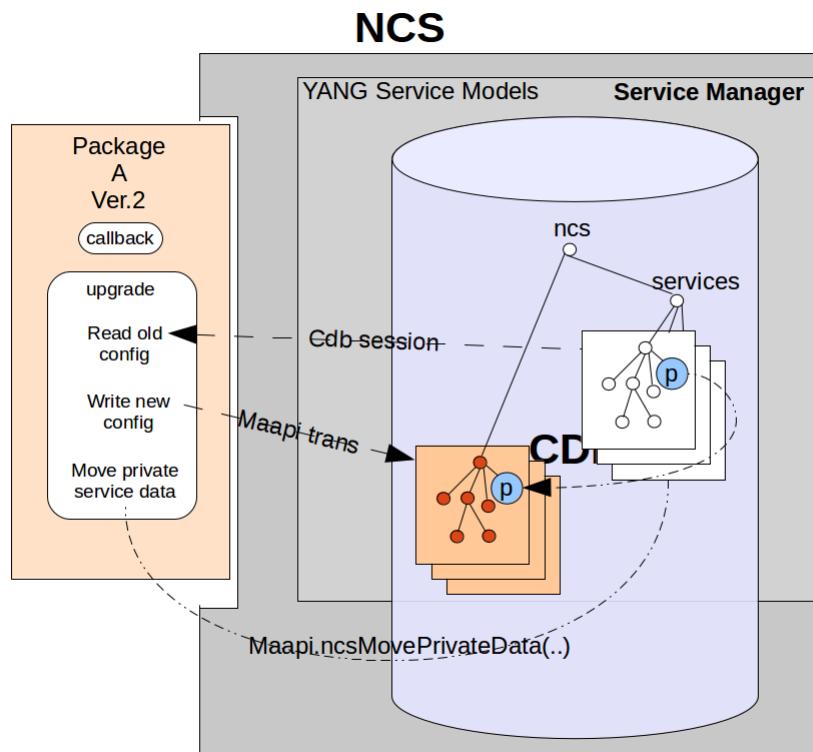
The value for new leaf introduced in the new version of the YANG model is calculated, and the value is set using Maapi and the init transaction:

```
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 90. 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 91. 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 15

Java API Overview

- [Introduction, page 227](#)
- [MAAPI, page 230](#)
- [CDB API, page 233](#)
- [DP API, page 236](#)
- [NED API, page 248](#)
- [NAVU API, page 248](#)
- [ALARM API, page 256](#)
- [NOTIF API, page 258](#)
- [HA API, page 260](#)
- [Java API Conf Package, page 260](#)
- [Namespace classes and the loaded schema, page 263](#)

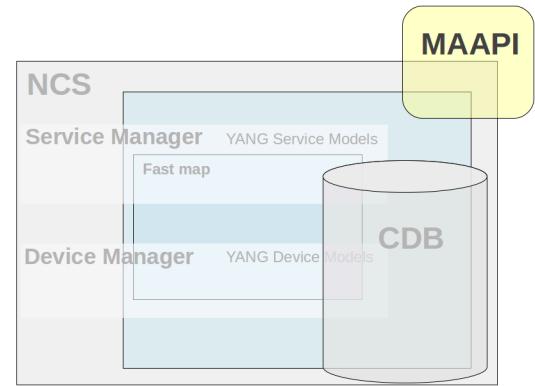
Introduction

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`). The jar files and their dependencies can be found under `$NCS_DIR/java/jar/`.

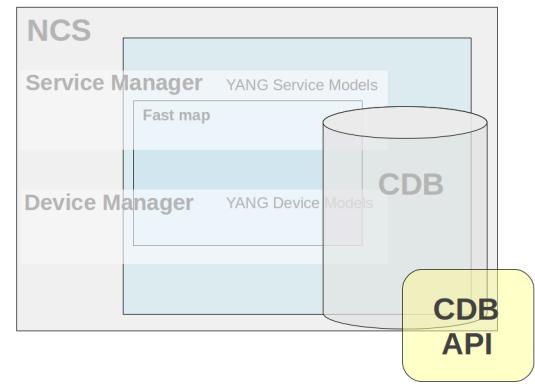
For convenience the Java build tool Apache ant (<https://ant.apache.org/>) is used to run all of the examples. However this tool is not a requirement for NSO.

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:

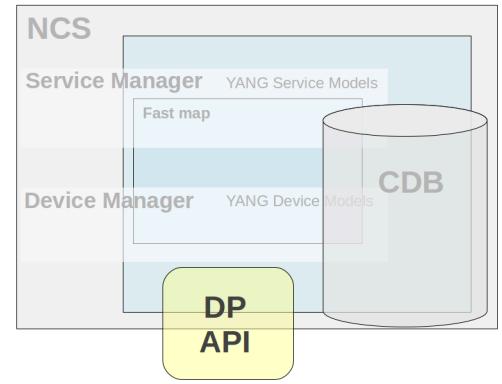
MAPI - (Management Agent API) Northbound interface that is transactional and user session based. Using this interface both configuration and operational data can be read. Configuration data can be written and committed as one transaction. The API is complete in the way that it is possible to write a 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.

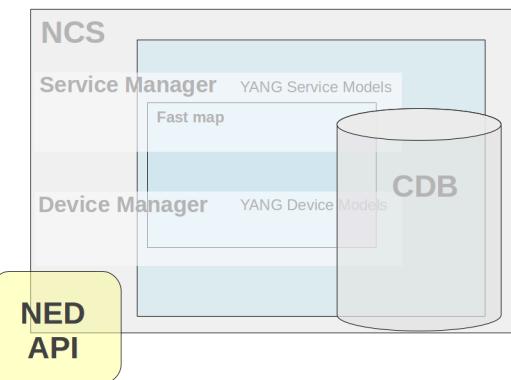


DP API - Southbound interface that enables callbacks, hooks and transforms. This API makes it possible to provide the service callbacks that handles service to device mapping logic. Other usual cases are external data providers for operational data or action callback implementations. There are also transaction 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.

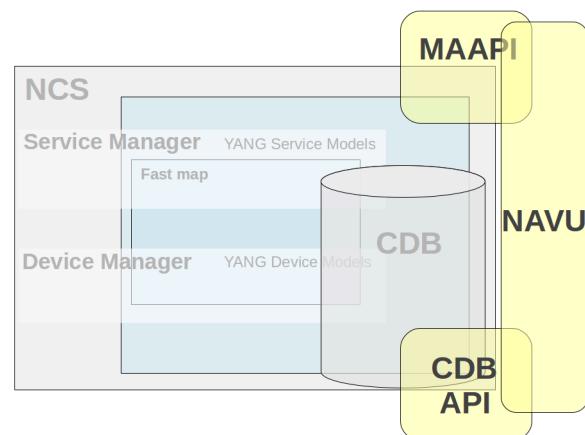


NED API - (Network Equipment Driver)

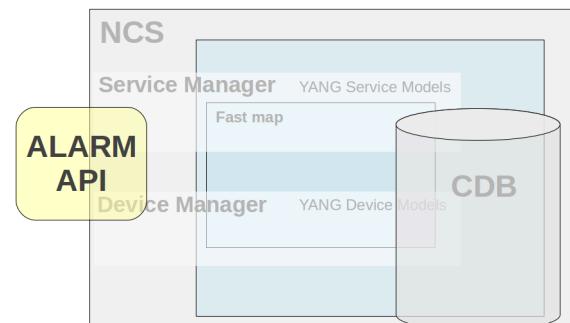
Southbound interface that mediate communication for devices that do not speak neither NETCONF nor SNMP. All prepackaged NEDs for different devices are written using this interface. It is possible to use the same interface to write your own NED. There are two types of NEDs, CLI NEDs and Generic NEDs. 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.



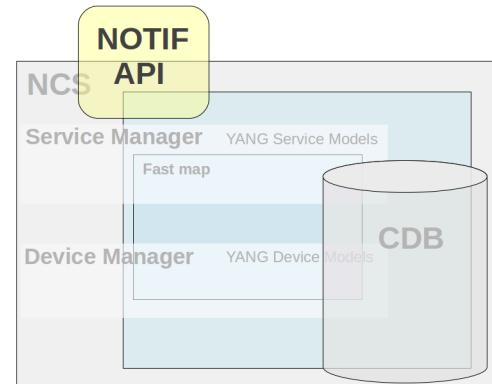
NAVU API - (Navigation Utilities) API that resides on top of the Maapi and Cdb APIs. It provides schema model navigation and instance data handling (read/write). Uses either a Maapi or Cdb context as data access and incorporates a subset of functionality from these (navigational and data read/write calls). Its major use is in service implementations which normally is about navigating device models and setting device data.



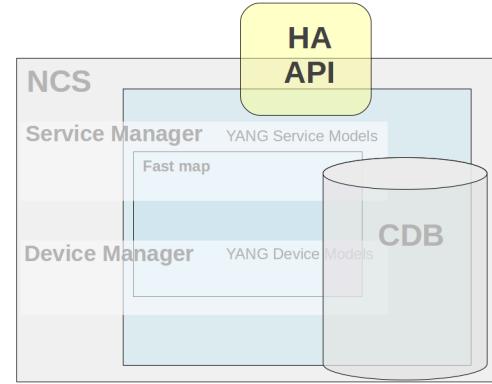
ALARM API - Eastbound api used both to consume and produce alarms in alignment with the NSO Alarm model. To consume alarms the AlarmSource interface is used. To produce a new alarm the AlarmSink interface is used. There is also a possibility to buffer produced alarms and make asynchronous writes to CDB 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, PRIMARY or SECONDARY. 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 92. 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 93. 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 94. 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 handler. This transaction handler is used when invoking the various Maapi methods.

An example of a typical transactional method is the `getElem()` method:

Example 95. 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 96. 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)`.

Calling the `applyTrans()` method also performs additional validation of the new data as required by the data model and may fail if the validation fails. You can perform the validation beforehand, using the `validateTrans()` method.

Additionally, applying transaction can fail in case of a conflict with another, concurrent transaction. The best course of action in this case is to retry the transaction. Please see [the section called “Handling Conflicts”](#) for details.

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 103, “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 97. 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 98. 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.cwd() 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 99. 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 subtrees. 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.

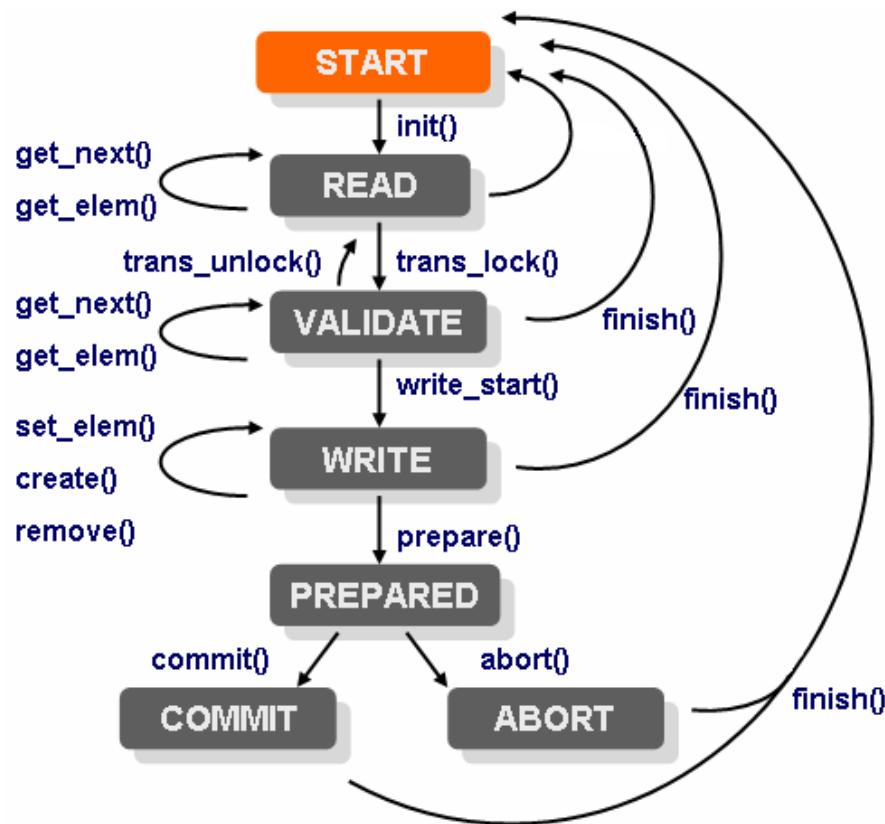
If the underlying database supports real atomic transactions, this is a good place to start such a transaction.

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 100. 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 101. 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 102. 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 18, 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;
        }
    }
}
```

Validation Callbacks

```

        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 103. 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 20, NED Development](#) for more information.

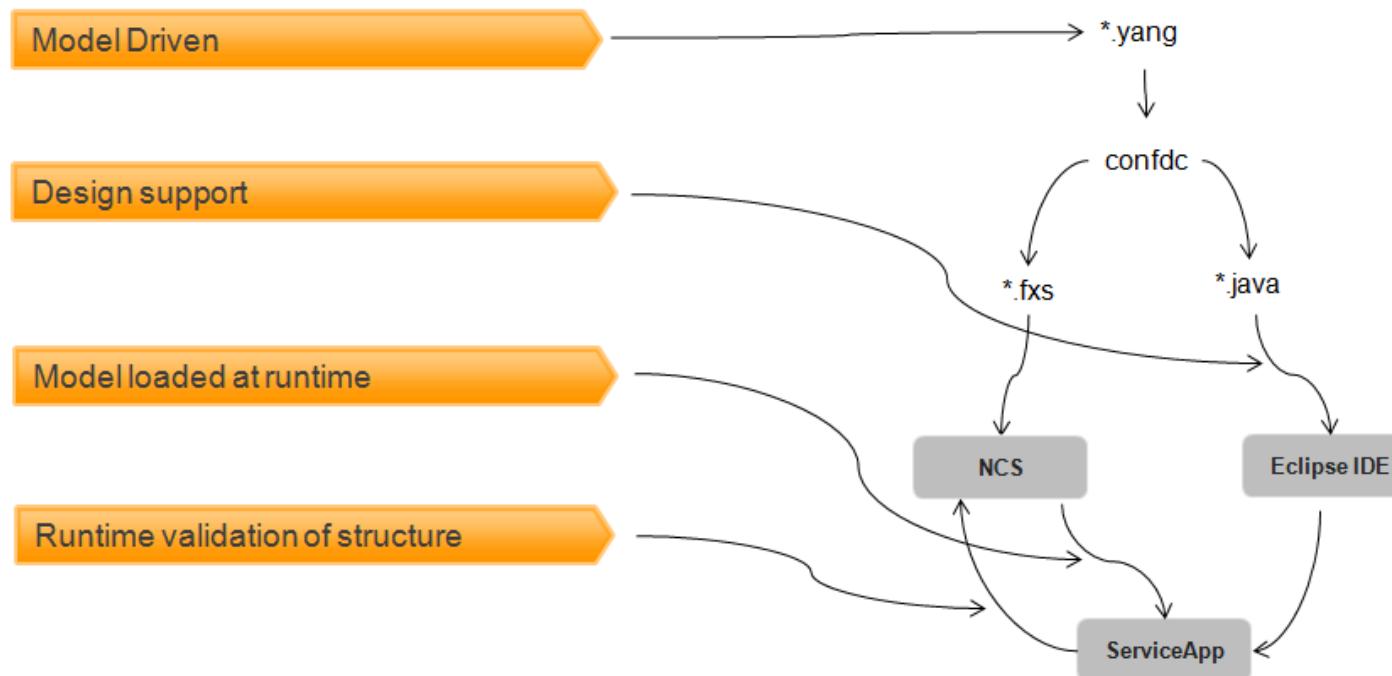
NAVU API

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.

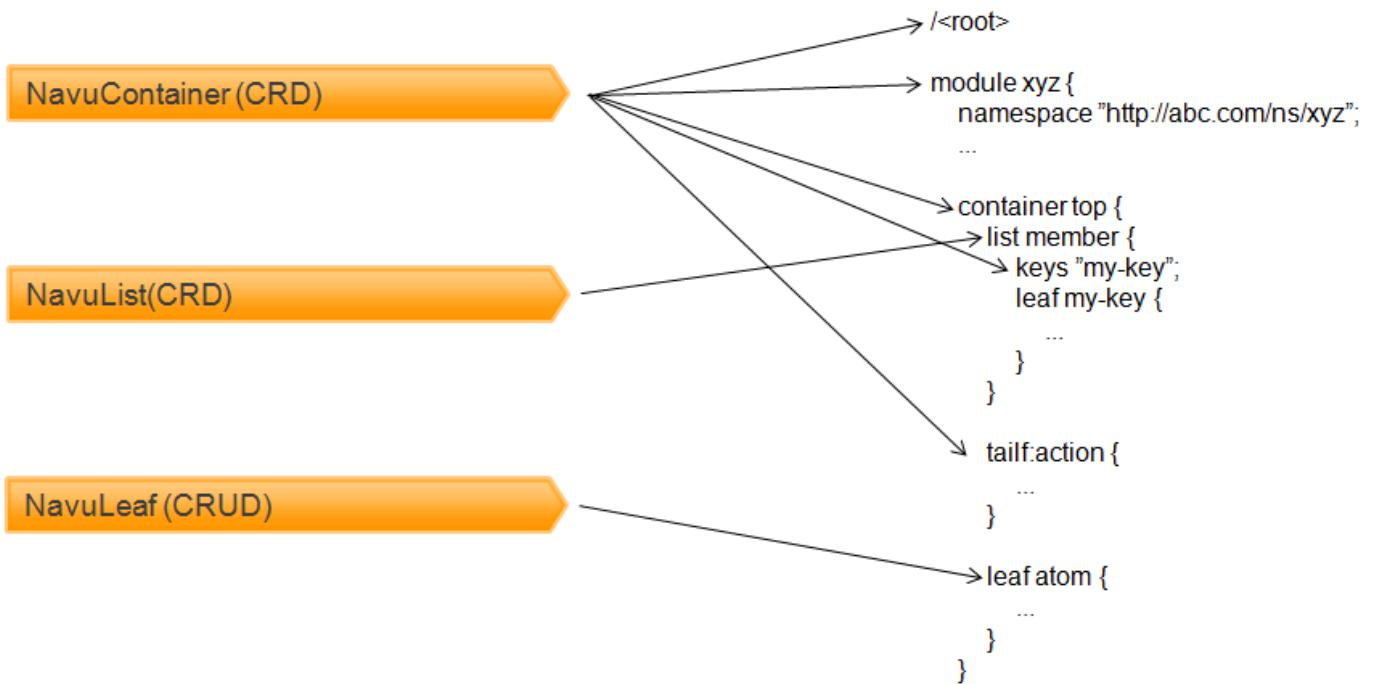
The ncsc tool can also generate java classes from the .yang files. These files, extending the ConfNamespace baseclass, are the java representation of the models and contains all defined nametags and their corresponding hash values. These java classes can, optionally, be used as help classes in the service applications to make NAVU navigation type safe, e.g. eliminating errors from misspelled model container names.

Figure 104. 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 105. 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 106. NSO Module

```

module tailf-ncs {
    namespace "http://tail-f.com/ns/ncs";
    ...
}
  
```

Example 107. 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 108. 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 109. 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 110. 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 111. 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 112. 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 113. 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 114. 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 115. 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 116. 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_ival:" +
((ConfXMLParamValue)result[2]).getValue().toString());
}
.....
context.finishClearTrans();

```

Or we could do it with xPathSelect()

Example 117. 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 8, *Alarm Manager in 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 ConfIdentityRef(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_COMPACTION` - This event is generated after each CDB compaction performed by NSO. The application receives a `CompactionNotification` where `CompactionNotification.dbfile` indicates which datastore was compacted, and `CompactionNotification.compaction_type` indicates whether the compaction was triggered manually or automatically by the system.
- `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 *Administration Guide*)

The following example configures three nodes in a HA cluster. One is set as primary and the other two as secondaries.

Example 118. 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 ConfigIPv4(InetAddress.getByName("localhost")));
ha0.bePrimary(primary.nodeid);

ha1.beSecondary(new ConfBuf("node1"), primary, true);
ha2.beSecondary(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 119. `ConfValue` types

Constant	YANG type	<code>ConfValue</code>	Description
J_STR	string	<code>ConfBuf</code>	Human readable string
J_BUF	string	<code>ConfBuf</code>	Human readable string
J_INT8	int8	<code>ConfInt8</code>	8-bit signed integer
J_INT16	int16	<code>ConfInt16</code>	16-bit signed integer
J_INT32	int32	<code>ConfInt32</code>	32-bit signed integer
J_INT64	int64	<code>ConfInt64</code>	64-bit signed integer
J_UINT8	uint8	<code>ConfUInt8</code>	8-bit unsigned integer
J_UINT16	uint16	<code>ConfUInt16</code>	16-bit unsigned integer
J_UINT32	uint32	<code>ConfUInt32</code>	32-bit unsigned integer
J_UINT64	uint64	<code>ConfUInt64</code>	64-bit unsigned integer
J_IPV4	inet:ipv4-address	<code>ConfIPv4</code>	64-bit unsigned
J_IPV6	inet:ipv6-address	<code>ConfIPv6</code>	IP v6 Address
J_BOOL	boolean	<code>ConfBoolean</code>	Boolean value
J_QNAME	xs:QName	<code>ConfQName</code>	A namespace/tag pair
J_DATETIME	yang:date-and-time	<code>ConfDateTime</code>	Date and Time Value
J_DATE	xs:date	<code>ConfDate</code>	XML schema Date
J_ENUMERATION	enum	<code>ConfEnumeration</code>	An enumeration value

Constant	YANG type	ConfValue	Description
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("lo0"));
```

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);
maapi.loadSchemas();

ArrayList<ConfNamespace> nsList = maapi.getAutoNsList();
```

Namespace classes and the loaded schema

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 16

Python API Overview

- [Introduction, page 265](#)
- [Python API overview, page 266](#)
- [Python scripting, page 267](#)
- [High-level MAAPI API, page 267](#)
- [Maagic API, page 268](#)
- [Maagic examples, page 274](#)
- [PlanComponent, page 277](#)
- [Python packages, page 278](#)
- [Low-level APIs, page 283](#)
- [Advanced topics, page 285](#)

Introduction

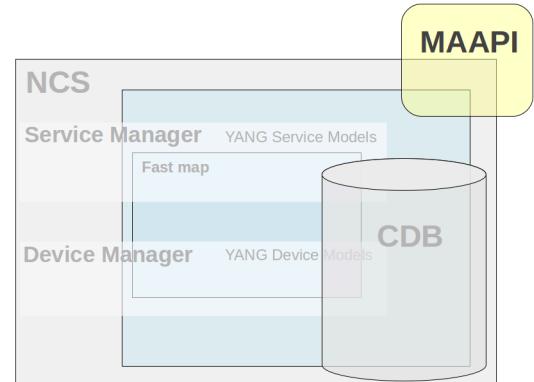
The NSO Python library contains a variety of APIs for different purposes. In this chapter we introduce these and explain their usage. The NSO Python modules deliverables are found in two variants, the low-level APIs and the high-level APIs.

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.

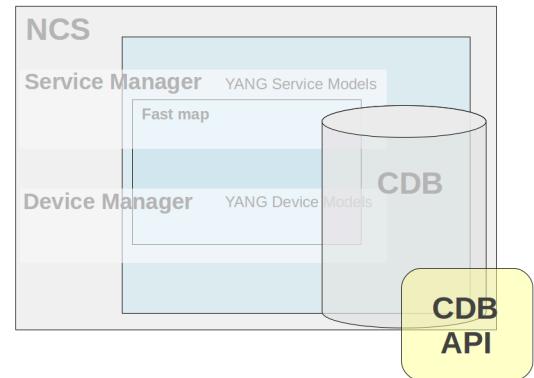
The high-level APIs is an abstraction layer on top of the low-level APIs to make them easier to use, to improve code readability and development rate for common use cases. E.g. services and action callbacks and common scripting towards NSO.

Python API overview

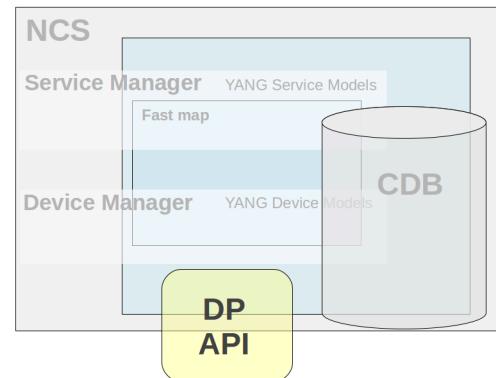
MAAPI - (Management Agent API) Northbound interface that is transactional and user session based. Using this interface, both configuration and operational data can be read. Configuration and operational data can be written and committed as one transaction. The API is complete in the way that it is possible 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.



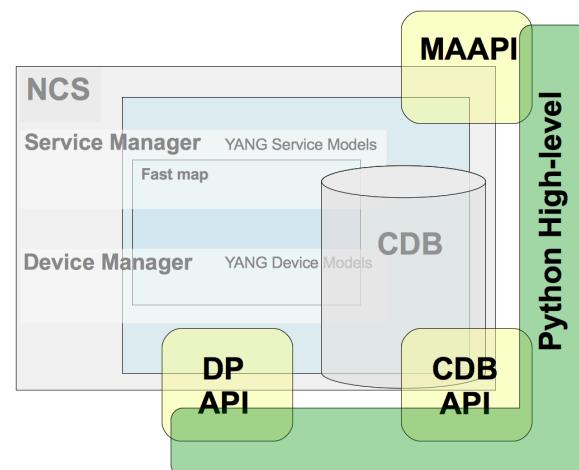
Python low-level CDB API - Southbound interface provides access to the CDB configuration database. Using this interface, configuration data can be read. In addition, operational data that is stored in CDB can be read and written. This interface has a subscription mechanism to subscribe to changes. A 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 120. 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)
```



Warning

The example code here shows how to start a transaction but does not properly handle the case of concurrency conflicts when writing data. Please see the section called “[Handling Conflicts](#)” for details.

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 121. 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 122. 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.



Object deletion

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)`.

```
# /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 123. 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 124. 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 125. 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 126. 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 127. 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 nano service. More info about plan-data is found in [Chapter 31, Nano Services for Staged Provisioning](#).

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 nano 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 128. 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 129. 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 tailf: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 130. 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_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 131. 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 (<code>create</code> , <code>set_elem</code> , ...) for configuration modifications.
<i>service</i>	ncs.maagic.Node	An object pointing to the service with the current transaction context, using shared operations (<code>create</code> , <code>set_elem</code> , ...) 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.

ValidationPoint handler

The Python high-level API provides an easy way to implement a validation point handler. The easiest way to create a handler is to use the **ncs-make-package** command. It creates ready to use skeleton code.

```

$ cd packages
$ ncs-make-package --service-skeleton python pyvalidation --component-class
  validation.ValidationApplication \
  --disable-service-example --validation-example

```

The generated package skeleton:

```

$ tree pyaction
pyaction/
+-- README
+-- doc/
+-- load-dir/
+-- package-meta-data.xml

```

```

+-- python/
|   +-- pyaction/
|       +-- __init__.py
|       +-- validation.py
+-- src/
|   +-- Makefile
|   +-- yang/
|       +-- validation.yang
--- templates/

```

This example validation point handler accepts all values except 'invalid'.

When debugging Python packages refer to [the section called “Debugging of Python packages”](#).

Example 132. Validation implementation

```

# -*- mode: python; python-indent: 4 -*-
import ncs
from ncs.dp import ValidationError, ValidationPoint

# -----
# VALIDATION EXAMPLE
# -----
class Validation(ValidationPoint):
    @ValidationPoint.validate
    def cb_validate(self, tctx, keypath, value, validationpoint):
        self.log.info('validate: ', str(keypath), '=', str(value))
        if value == 'invalid':
            raise ValidationError('invalid value')
        return ncs.CONFD_OK

# -----
# COMPONENT THREAD THAT WILL BE STARTED BY NCS.
# -----
class ValidationApplication(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('ValidationApplication RUNNING')

        # When using actions, this is how we register them:
        #
        self.register_validation('pyvalidation-valpoint', Validation)

        # 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('ValidationApplication FINISHED')

```

Test the validation by setting the value to invalid and validating the transaction from the NSO CLI:

```
admin@ncs% set validation validate-value invalid
```

```
admin@ncs% validate
Failed: 'validation validate-value': invalid value
[ok][2016-04-22 10:30:39]
```

Table 133. ValidationPoint handler callback parameters

Parameter	Type	Description
<i>self</i>	ncs.dp.ValidationPoint	The validation point object.
<i>tctx</i>	ncs.TransCtxRef	Transaction context.
<i>kp</i>	ncs.HKeypathRef	The keypath of the node being validated.
<i>value</i>	ncs.Value	Current value of the node being validated.
<i>validationstring</i>		The validation point that triggered the validation.

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 MAAP API

This API is a direct mapping of the NSO MAAP C API. See `pydoc3 _ncs.maapi` and `man confd_lib_maapi` for further information.

Note that additional care must be taken when using this API in service code, as it also exposes functions that do not perform reference counting (see [the section called “Reference Counting Overlapping Configuration”](#)).

In service code, you should use the `shared_*` set of functions, such as:

```
shared_apply_template
shared_copy_tree
shared_create
shared_insert
shared_set_elem
shared_set_elem2
shared_set_values
```

and avoid the non-shared variants:

```
load_config()
load_config_cmds()
load_config_stream()
apply_template()
copy_tree()
create()
insert()
set_elem()
set_elem2()
set_object
```

```
    set_values()
```

The following example is a script to read and de-crypt a password using the Python low-level MAAP API.

Example 134. Setting of configuration data using MAAP

```
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 MAAP API.

Example 135. 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 _ncs.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 136. 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)

```

Advanced topics

Schema loading - internals

When schemas are loaded, either upon direct request or automatically by methods and classes in the `maapi` module, they are statically cached inside the Python VM. This fact presents a problem if one wants to connect to several different NSO nodes with diverging schemas from the same Python VM.

Take for example the following program that connects to two different NSO nodes (with diverging schemas) and show their ned-id's.

Example 137. Reading NED-id's (`read_nedids.py`)

```
import ncs

def print_ned_ids(port):
    with ncs.maapi.single_read_trans('admin', 'system', db=ncs.OPERATIONAL, port=port,
                                     dev_ned_id = ncs.maagic.get_node(t, '/devices/ned-ids/ned-id')
                                     for id in dev_ned_id.keys():
                                         print(id)

if __name__ == '__main__':
    print('== lsa-1 ==')
    print_ned_ids(4569)
    print('== lsa-2 ==')
    print_ned_ids(4570)
```

Running this program may produce output like this:

```
$ python3 read_nedids.py
== lsa-1 ==
{ned:lsb-netconf}
{ned:netconf}
{ned:snmp}
{cisco-nso-nc-5.5:cisco-nso-nc-5.5}
== lsa-2 ==
{ned:lsb-netconf}
{ned:netconf}
{ned:snmp}
{<_ncs.Value type=C_IDENTITYREF(44) value='idref<211668964'...>]}
{<_ncs.Value type=C_IDENTITYREF(44) value='idref<151824215'>'}
{<_ncs.Value type=C_IDENTITYREF(44) value='idref<208856485'...>}'
```

The output shows identities in string format for the active NEDs on the different nodes. Note that for `lsa-2` the last three lines do not show the name of the identity but instead the representation of a `_ncs.Value`. The reason for this is that `lsa-2` has different schemas which does not include these identities. Schemas for this Python VM was loaded and cached during the first call to `ncs.maapi.single_read_trans()` so no schema loading occurred during the second call.

The way to make the program above work as expected is to force reloading of schemas by passing an optional argument to `single_read_trans()` like so:

```
with ncs.maapi.single_read_trans('admin', 'system', db=ncs.OPERATIONAL, port=port,
                                 load_schemas=ncs.maapi.LOAD_SCHEMAS_RELOAD) as t:
```

Running the program with this change may produce something like this:

```
== lsa-1 ==
{ned:lsb-netconf}
{ned:netconf}
{ned:snmp}
```

```
{cisco-nso-nc-5.5:cisco-nso-nc-5.5}
==  lsa-2 ==
{ned:lsa-netconf}
{ned:netconf}
{ned:snmp}
{cisco-asa-cli-6.13:cisco-asa-cli-6.13}
{cisco-ios-cli-6.72:cisco-ios-cli-6.72}
{router-nc-1.0:router-nc-1.0}
```

Now, this was just an example of what may happen when wrong schemas are loaded. Implications may be more severe though, especially if maagic nodes are kept between reloads. In such cases, accessing an "invalid" maagic object may in best case result in undefined behavior making the program not work, but might even crash the program. So care need to be taken to not reload schemas in a Python VM if there are dependencies to other parts in the same VM that need previous schemas.

Functions and methods that accept the `load_schemas` argument:

- `ncs.maapi.Maapi()` constructor
- `ncs.maapi.single_read_trans()`
- `ncs.maapi.single_write_trans()`



CHAPTER 17

NSO Packages

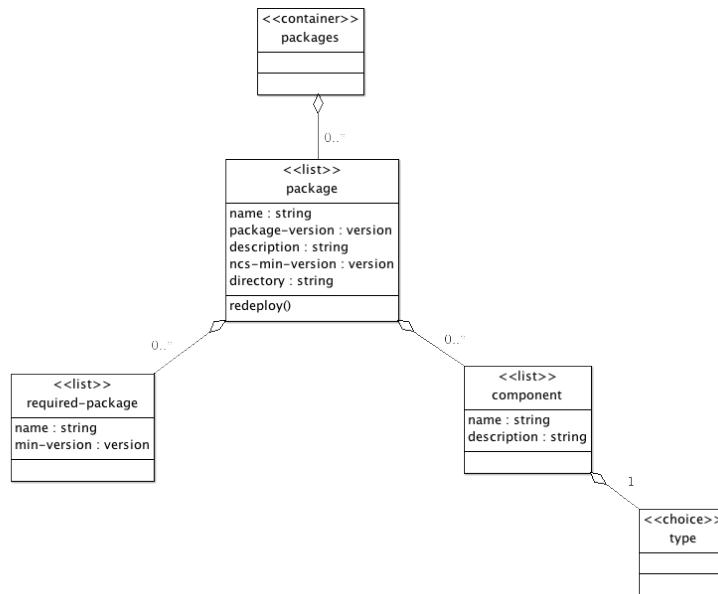
- [Package Overview, page 289](#)
- [An Example Package, page 291](#)
- [The package-meta-data.xml file, page 291](#)
- [Components, page 295](#)
- [Creating Packages , page 297](#)

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 138. 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 139. 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 ncs-max-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 supported-ned-id-match* string
        +-ro required-package* [name]
          | +-ro name              string
          | +-ro min-version?     version
          | +-ro max-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>
      <ncs-min-version>5.7</ncs-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>
        <ncs-min-version>5.7</ncs-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>
      </package>
    </packages>
  </config>

```

The package-meta-data.xml file

```

<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.
- ncs-max-version - the latest 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>
```
- supported-ned-id-match - the list of regular expressions for ned-ids supported by this package. Ned-ids in the system that matches at least one of the regular expressions in this list are added to the supported-ned-id list. The following example demonstrates how all minor versions with a major number of 1 of the router-nc NED can be added to a package's list of supported ned-ids:


```
<supported-ned-id-match>router-nc-1.\d+:router-nc-1.\d+</supported-ned-id-match>
```
- 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 [the section called “CLI 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: `$NCS_DIR/examples.ncs/getting-started/developing-with-ncs/4-rfs-service` See [Chapter 7, Implementing Services](#) for a thorough introduction to services.
- *ActionCallback* - to implement user defined tailf:actions or YANG RPCs. See example: `$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 <code>\$NCS_DIR/examples.ncs/getting-started/developing-with-ncs/1-cdb</code> 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 <code>\$NCS_DIR/examples.ncs/getting-started/developing-with-ncs/14-upgrade-service</code> illustrates user cdb upgrades using <i>upgrade</i> components.

Creating Packages

NSO ships with a tool *ncs-make-package* that can be used to create packages. [Chapter 18, 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 acmel authgroup southbound-bob address 10.2.3.4
> set devices device acmel 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 18

Package Development

- [Developing a Service Package, page 299](#)
- [ncs-setup, page 301](#)
- [The netsim Part of a NED Package, page 302](#)
- [Plug-and-play scripting, page 304](#)
- [Creating a Service Package, page 304](#)
- [Java Service Implementation, page 305](#)
- [Developing our First Service Application, page 305](#)
- [Tracing Within the NSO Service Manager, page 308](#)
- [Controlling error messages info level from Java, page 311](#)
- [Loading Packages, page 312](#)
- [Debugging the Service and Using Eclipse IDE, page 312](#)
- [Working with ncs-project, page 316](#)

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 17, 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:

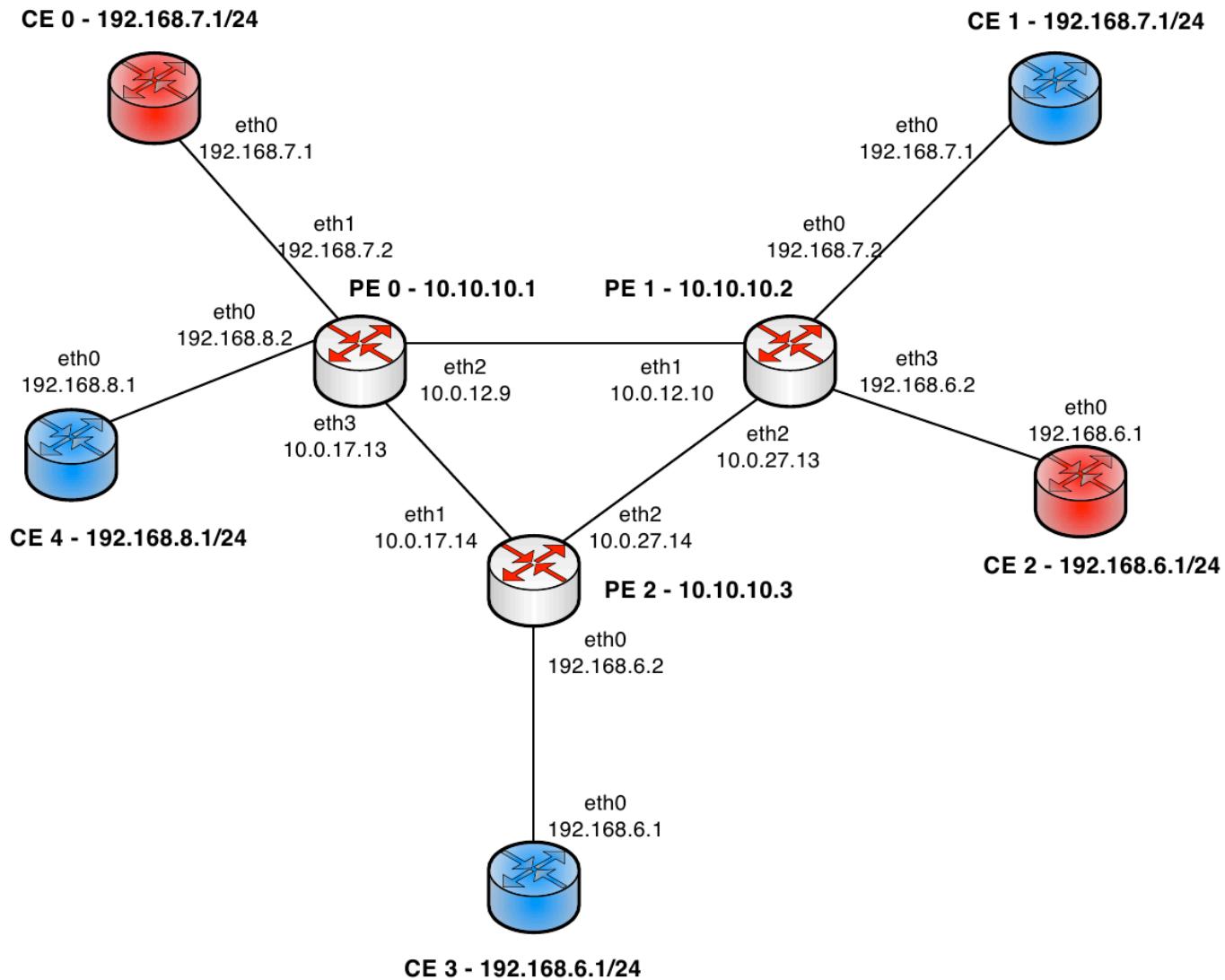
- A Makefile to build the source code of the package. The package contains source code and needs to be built.

- 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 140. 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 9, *Plug-and-play Scripting* in *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 7, Implementing 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 141. 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 142. 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 143. 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 144. 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 145. 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 146. 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 10, 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 147. 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 148. 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 149. 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 150. 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 151. 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 152. 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 156, “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 153. 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 154. 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 155. 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 156. 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 157. 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 19

Service Development Using Java

As using Java for service development may be somewhat more involved than Python, this chapter provides further examples and additional tips for setting up development environment for Java.

The two examples, a simple VLAN service and a Layer 3 MPLS VPN service, are more elaborate but show the same techniques as [Chapter 7, Implementing Services](#). If you or your team primarily focuses on services implemented in Python, feel free to skip or only skim through this chapter.

- [Simple VLAN service, page 325](#)
- [Simple VLAN service with templates, page 340](#)
- [Layer 3 MPLS VPN service, page 343](#)

Simple VLAN service

In this example, you will create a simple VLAN service in Java. In order to illustrate the concepts, the device configuration is simplified from a networking perspective and only uses one single device type (Cisco IOS).

Overview

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. |
-

These 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 new directory that will contain the files for this example, such as:

```
$ 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 -Fl
README.ncs
README.netsim
logs/
ncs-cdb/
ncs.conf
netsim/
packages/
scripts/
state/
```

Note the packages directory, **cd** to it:

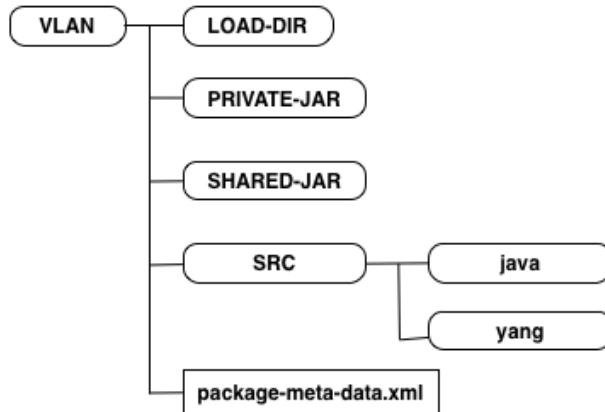
```
$ cd packages
$ ls -l
total 8
cisco-ios -> ../../packages/neds/cisco-ios
```

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 158. 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

So, if a user wants to create a new VLAN in the network what should the parameters be? 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.

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.

Make sure you keep the lines generated by the **ncs-make-package**:

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

The two lines tell NSO that this is a service. 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** (assumes you have the prerequisite 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
```

Now 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 a "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 10, 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 159. 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 160. 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. To do this, 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
```

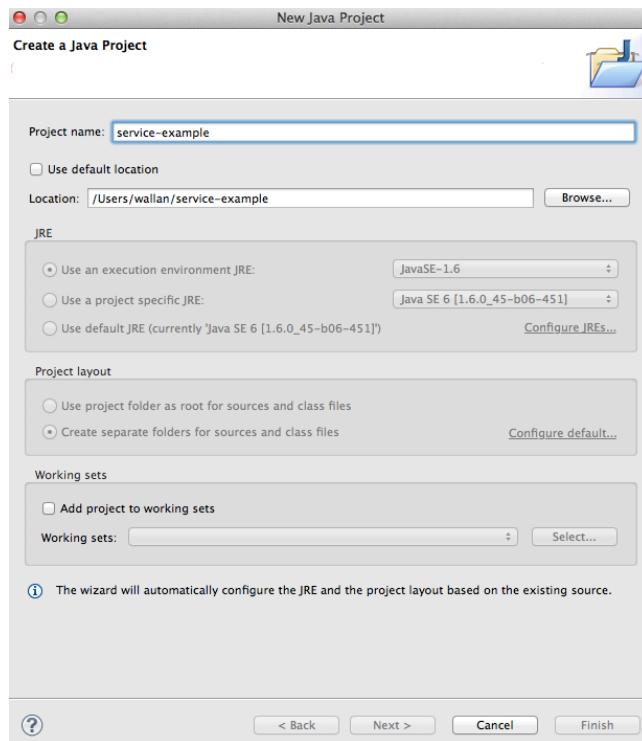
So modifying or creating a VLAN service will now have the "Hello World!" string show up in the shell. You can modify the package and reload/redeploy and see the output.

Using Eclipse

To use a GUI-based IDE Eclipse, first 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 are immediately ready to run this code in eclipse.

Figure 161. 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:

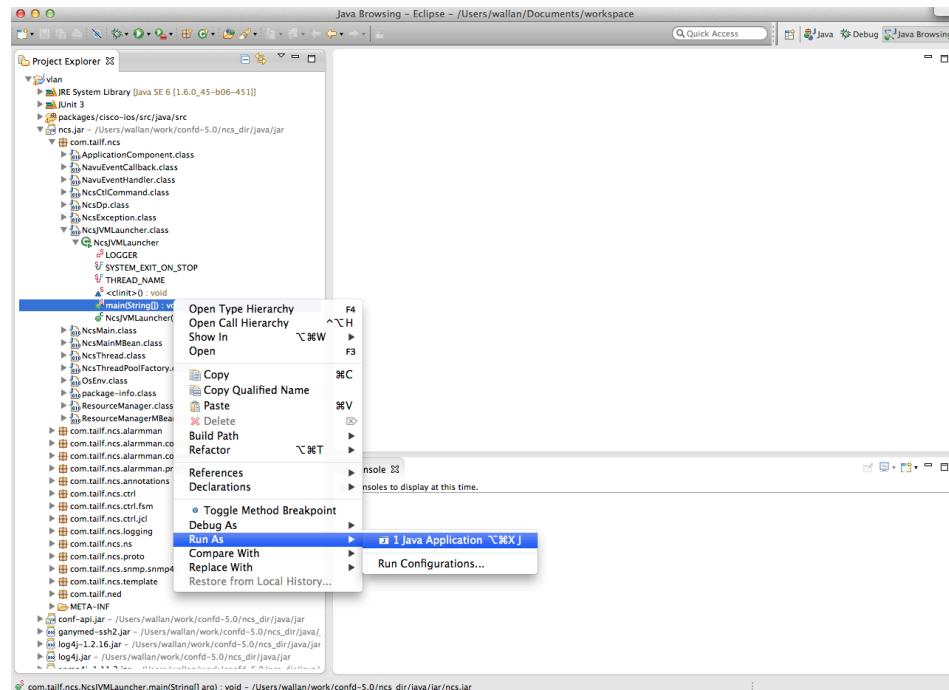
```
admin@ncs# show packages package vlan
packages package vlan
```

```

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:

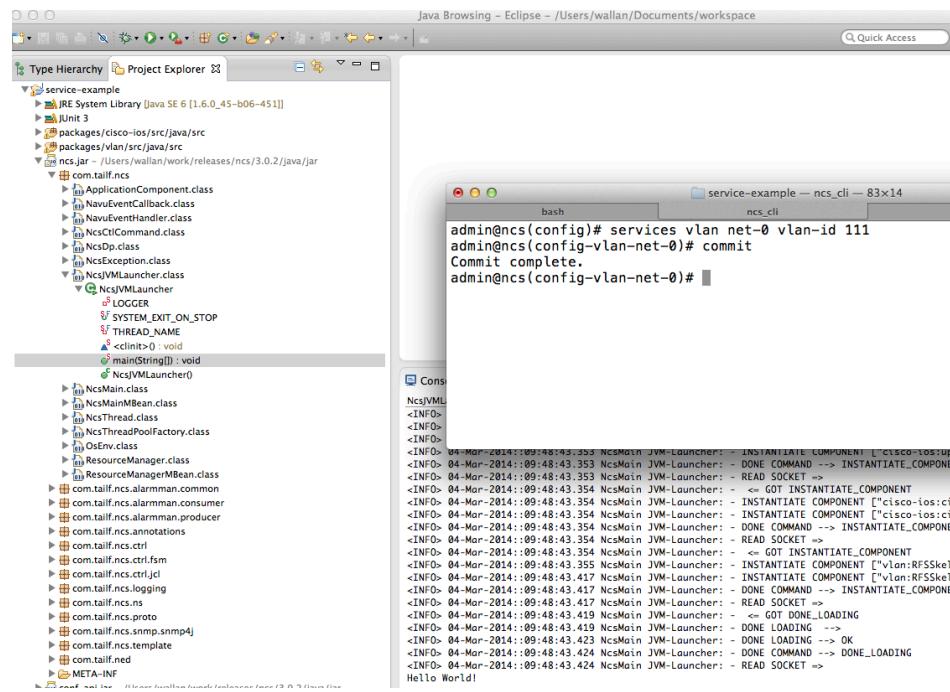
Figure 162. Starting the NSO JVM from Eclipse



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 163. Console output in Eclipse



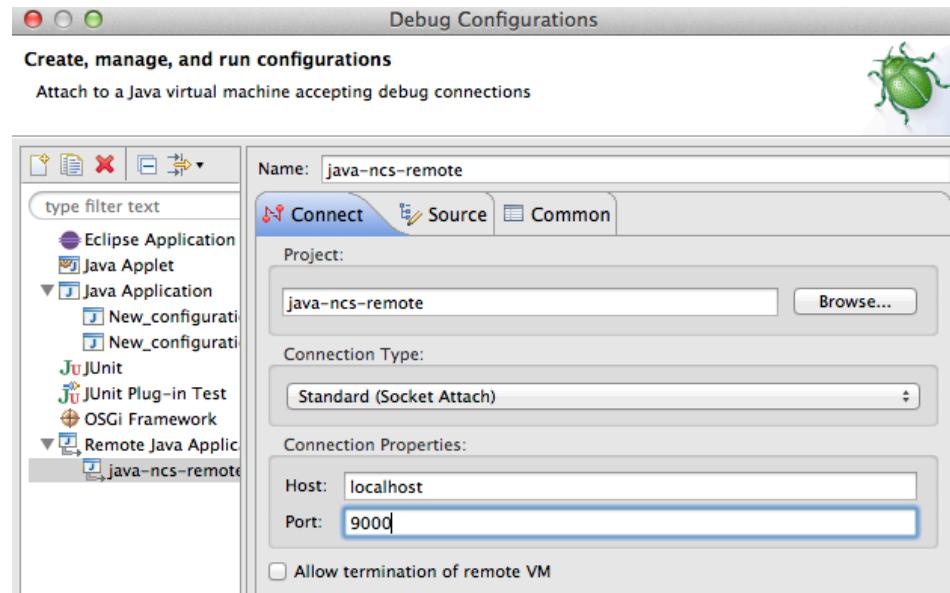
Another option is to have Eclipse connect to the running VM. Start the VM manually with the **-d** option.

```
$ ncs-start-java-vm -d
Listening for transport dt_socket at address: 9000
NCS JVM STARTING
...

```

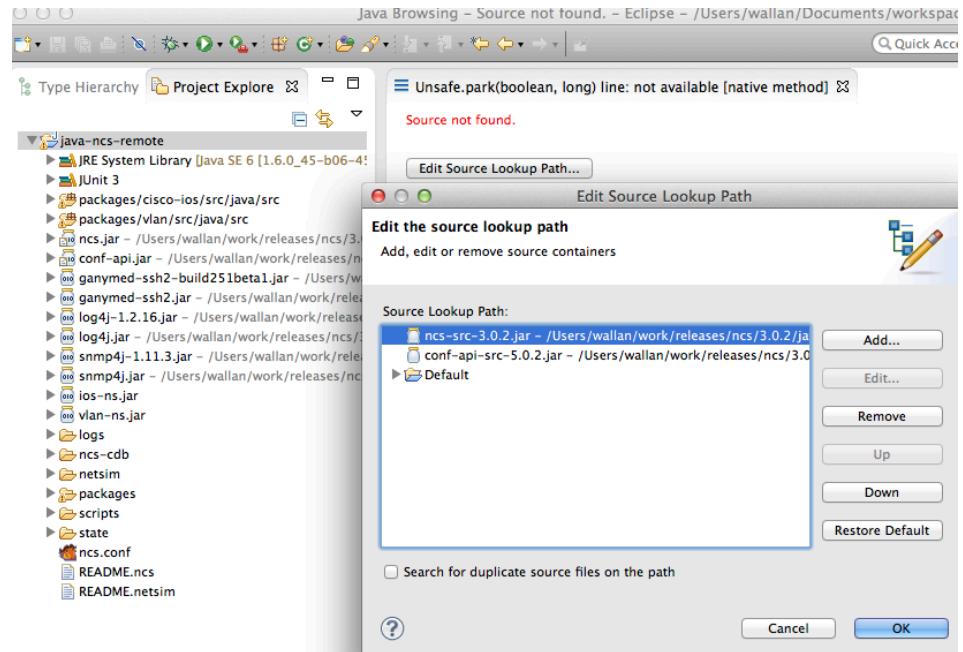
Then you can setup Eclipse to connect to the NSO Java VM:

Figure 164. Connecting to NSO Java VM Remote with Eclipse



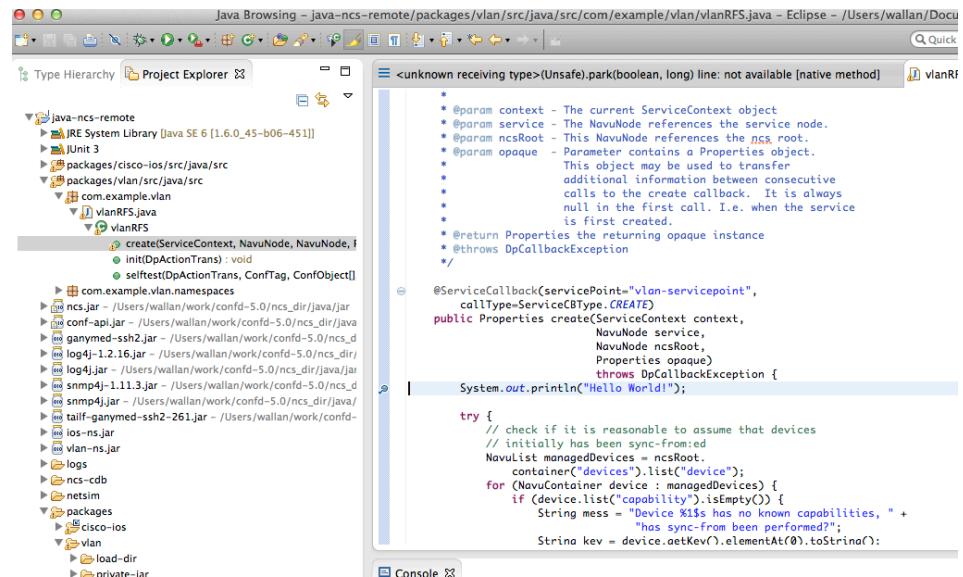
In order for Eclipse to show the NSO code when debugging add the NSO Source Jars, (add external Jar in Eclipse):

Figure 165. Adding the NSO source Jars



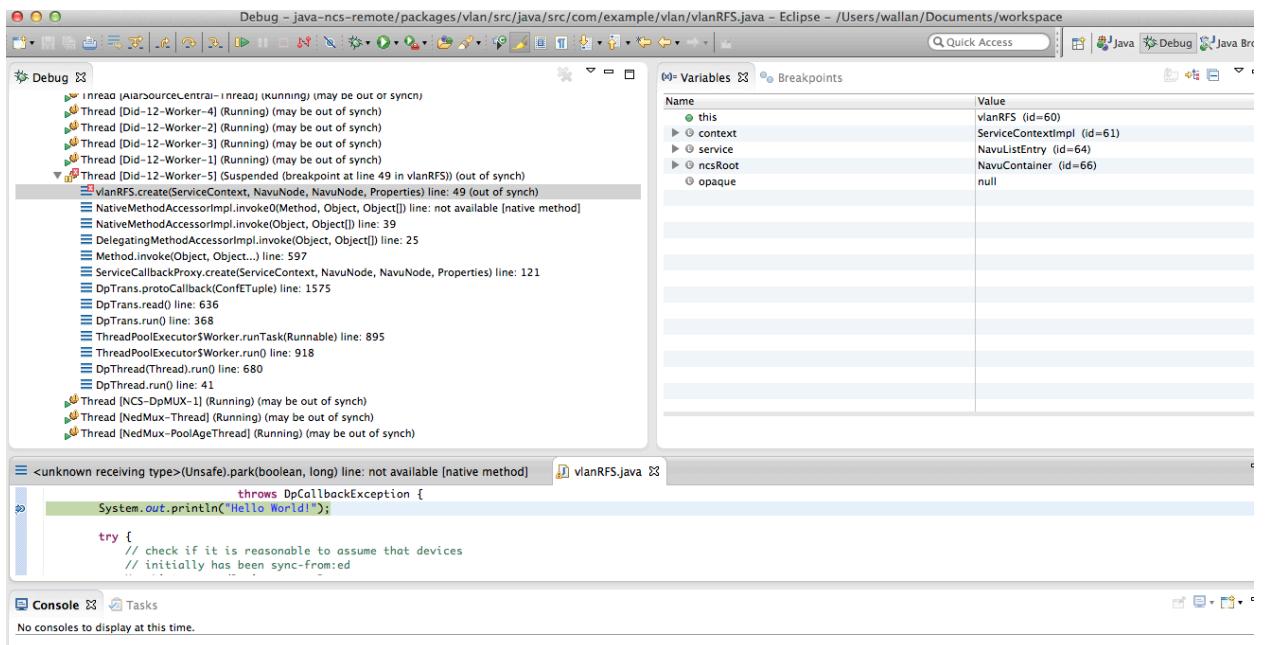
Navigate to the service create for the VLAN service and add a breakpoint:

Figure 166. Setting a break-point in Eclipse



Commit a change of a VLAN service instance and Eclipse will stop at the breakpoint:

Figure 167. 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 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-from: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 168. 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(NetconfContainer 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 169. 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 ConfUIInt32 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 170. 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 171. The Cisco IOS Model

Element	Schema	Type	Flags	Opts	Status	Path
tailf-ned-cisco-ios	module	string	config	?	current	/ios:version
version	leaf	string	config		current	/ios:service
service	container	config			current	/ios:platform
platform	container	config			current	/ios:hostname
hostname	leaf	string	config	?	current	/ios:enable
enable	container	config			current	/ios:archive
archive	container	config			current	/ios:username
username	list	config			current	/ios:controller
controller	list	config			current	/ios:vrf
vrf	container	config			current	/ios:ip
ip	container	config			current	/ios:vmpls
vmpls	container	config			current	/ios:ipv6
ipv6	container	config			current	/ios:vlan
vlan	container	config			current	/ios:interface
Interface	container	config			current	/ios:interface/ios:Embedded-Service-Engine
Embedded-Service-Engine	list	config			current	/ios:interface/ios:FastEthernet
FastEthernet	list	config			current	/ios:interface/ios:FastEthernet/ios:name
name	leaf	string	config		current	

Now, we replace the print statements with setting real configuration on the devices.

Figure 172. 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);

```

Note that the code uses the `sharedCreate()` functions instead of `create()`, as the shared variants are preferred and a best practice.

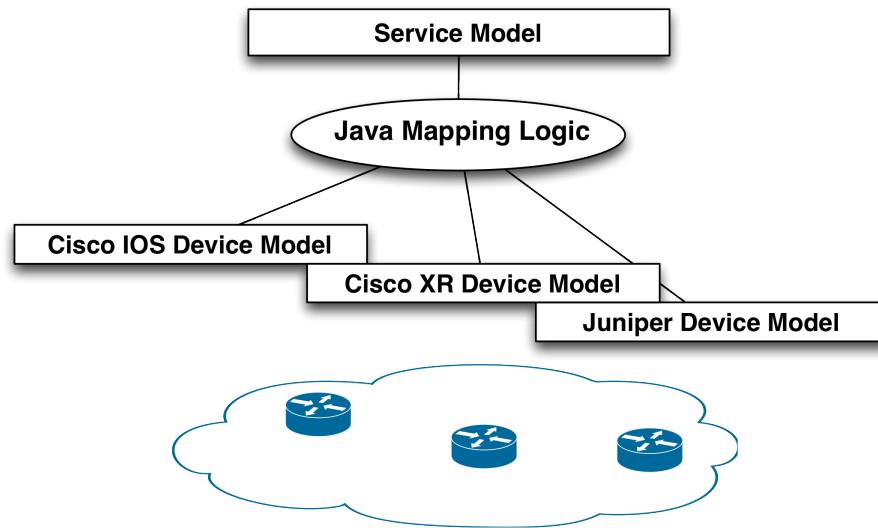
The above create method is all that is needed for create, read, update and delete. NSO will automatically handle any changes, like changing the VLAN ID, adding an interface to the VLAN service and deleting the service. This is handled by the FASTMAP engine, it renders any change based on the single definition of the create method.

Simple VLAN service with templates

Overview

The mapping strategy using only Java is illustrated in the following figure.

Figure 173. Flat mapping with Java

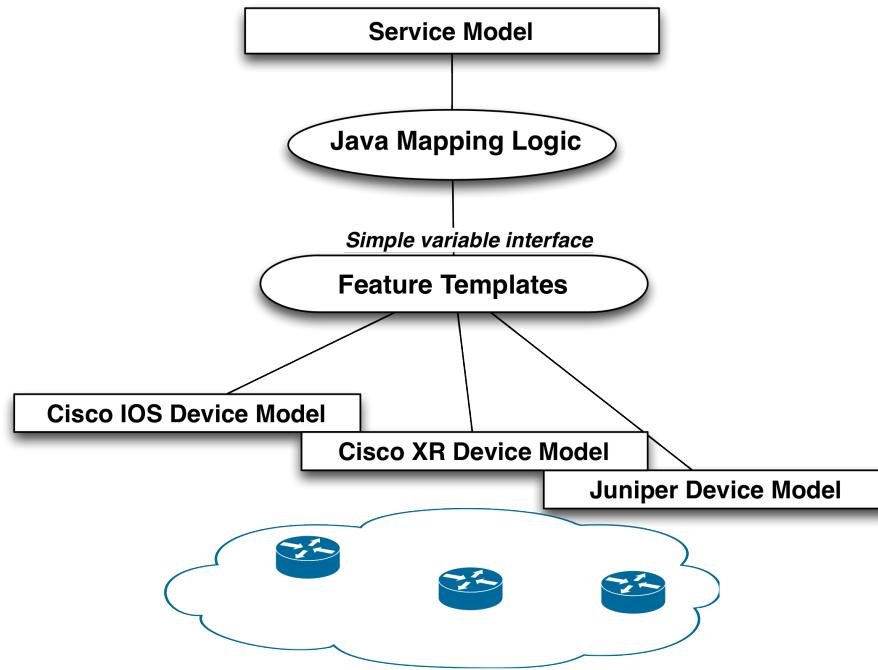


This strategy has some drawbacks:

- 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:

Figure 174. Two layered mapping using feature templates



In this model the Java layer focuses 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 take variables as input from the service logic, and the templates in turn transform these into concrete device configuration. Introduction 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 175. 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

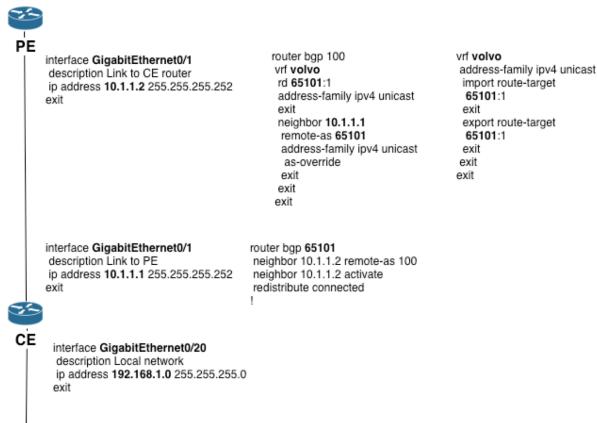
Layer 3 MPLS VPN service

This service shows a more elaborate service mapping. It is based on the `examples.ncs/service-provider/mpls-vpn` example.

MPLS VPNs are a type of a Virtual Private Network (VPN) that achieves segmentation of network traffic using Multiprotocol Label Switching (MPLS), often found in Service Provider (SP) networks. The Layer 3 variant uses BGP to connect and distribute routes between sites of the VPN.

The figure below illustrates an example configuration for one leg of the VPN. Configuration items in bold are variables that are generated from the service inputs.

Figure 176. Example L3 VPN Device Configuration



Auxiliary Service Data

Sometimes the input parameters are enough to generate the corresponding device configurations. But in many cases this is not enough. 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. Nano services are 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

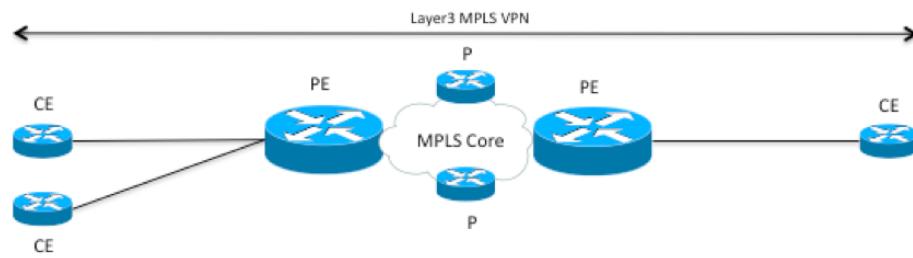
for that service in the network NSO treats topologies in the same way. It is of course common that you need to reference topology information in the service but it is highly desirable to have a decoupled and self-sufficient service that only uses the part of the topology that is interesting/needed for 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.

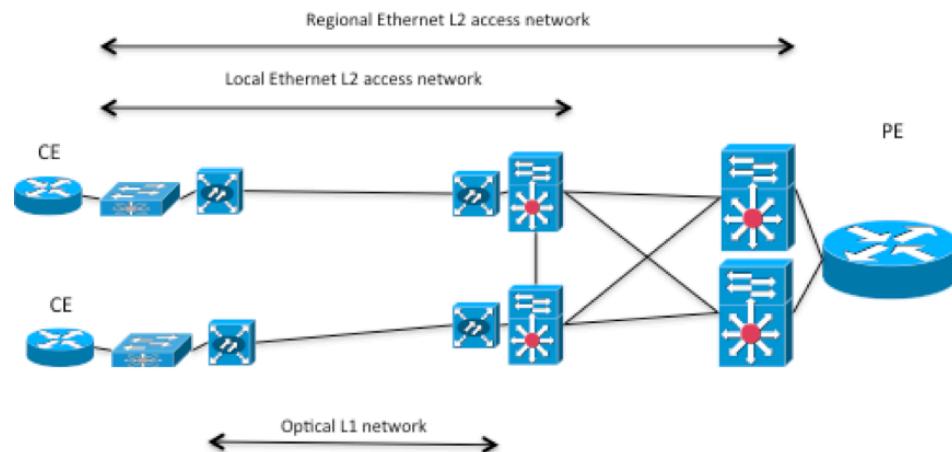
To illustrate this, lets look at a Layer 3 MPLS VPN service. A logical overview of an MPLS VPN with three endpoints could look something like this. CE routers connecting to PE routers, that are connected to an MPLS core network. In the MPLS core network there are a number of P routers.

Figure 177. 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 178. L1-L2 Topology



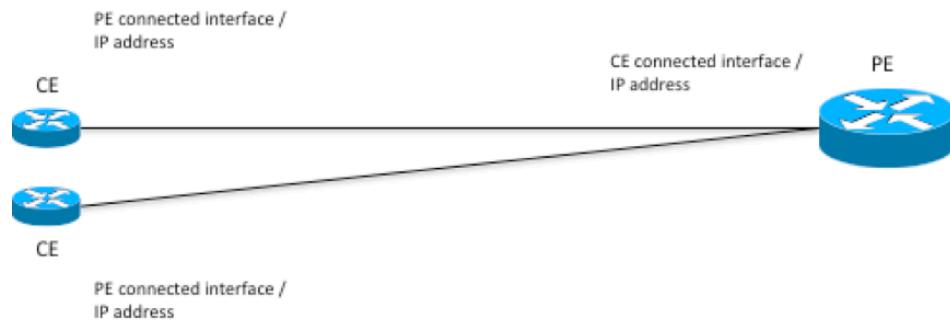
In pretty much all networks there is an access network between the CE and PE router. In the picture above the CE routers are connected to local Ethernet switches connected to a local Ethernet access network,

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 179. 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:

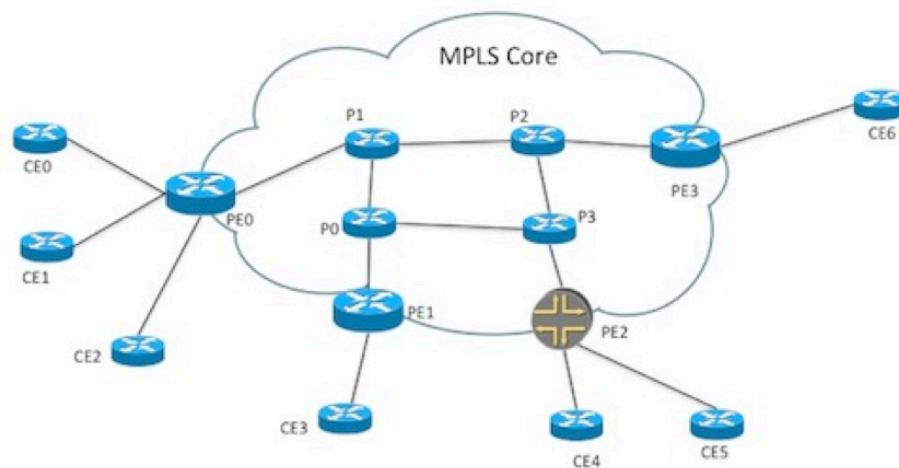
- Interface on the CE router that is connected to the PE router, and IP address of that interface.
- Interface on the PE router that is connected to the CE router, and IP address to the interface.

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 180. 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 181. 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 182. PE Router Config

```

vrf volvo
  address-family ipv4 unicast
    import route-target
      65101:1
    exit
    export route-target
      65101:1
    exit
  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 / ce1 - 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.

Defining the Mapping

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:

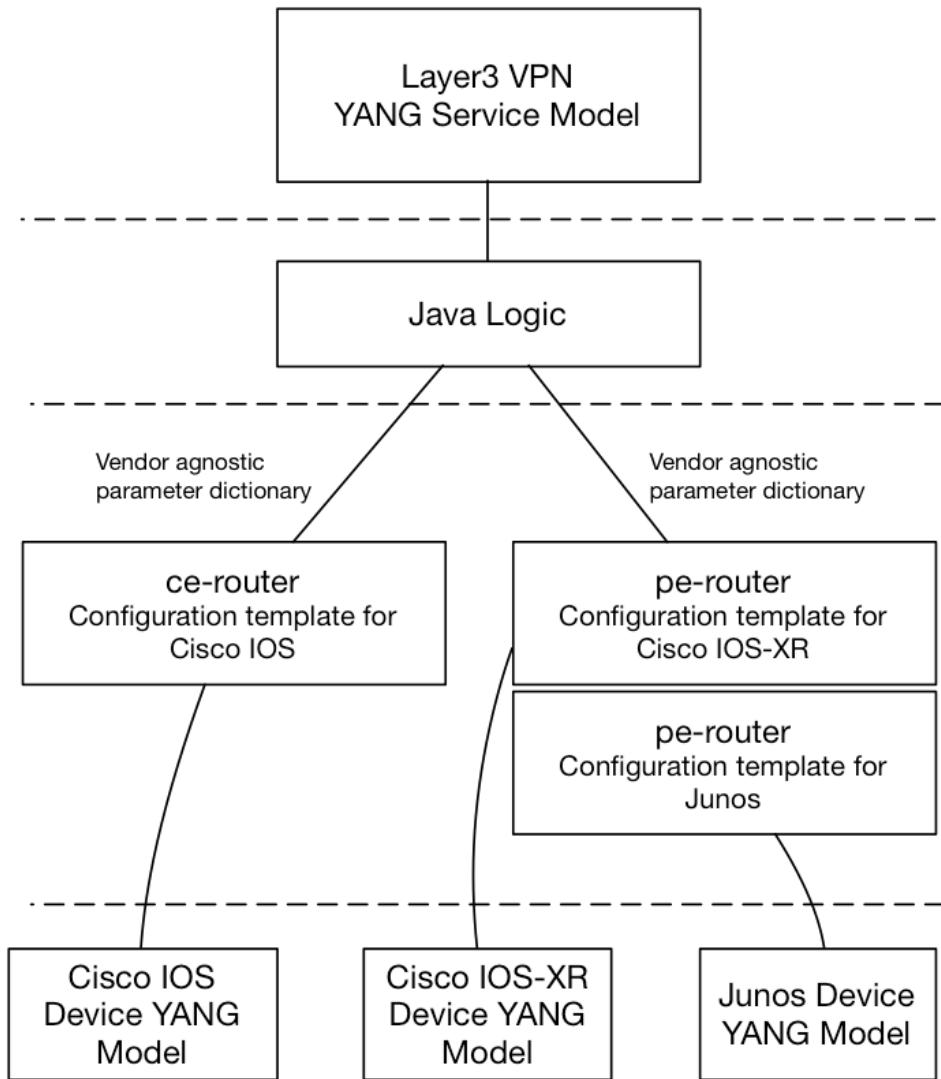
- CE router

- 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 183. 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
  
```

Defining the Mapping

```

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, "l3vpn-ce");
NavuList endpoints = service.list("endpoint");
NavuContainer topology = ncsRoot.getParent().
    container("http://com/example/l3vpn").
    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_ADDR}</id>
              <remote-as>100</remote-as>
              <activate/>
            </neighbor>
            <network>
              <number>{$LOCAL_CE_NET}</number>
            </network>
          </bgp>
        </router>
      </config>
    </device>
  </devices>
</config-template>
```

Defining the Mapping

```
</config>
</device>
</devices>
</config-template>
```



CHAPTER 20

NED Development

- [Creating a NED, page 355](#)
- [Types of NED Packages, page 355](#)
- [Dumb Versus Capable Devices, page 356](#)
- [NETCONF NED Development, page 358](#)
- [CLI NED Development, page 370](#)
- [Generic NED Development, page 428](#)
- [The SNMP NED, page 432](#)
- [Statistics, page 439](#)
- [Making the NED handle default values properly, page 442](#)
- [Dry-run considerations, page 445](#)
- [NED identification, page 446](#)
- [Migrating to the `juniper-junos_nc-gen` NED, page 447](#)

Creating a NED

A Network Element Driver (NED) represents a key NSO component that allows NSO to communicate southbound with network devices. 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. The YANG models can cover only a tiny subset of the device or all of the device.

Typically, the YANG models contained in a NED represent the subset of the device's configuration data, state data, Remote Procedure Calls, and notifications to be managed using NSO.

This guide provides information on NED development, focusing on building your own NED package. For a general introduction to NEDs, Cisco-provided NEDs, and NED administration, refer to the Chapter 14, *NED Administration* in *Administration Guide*.

Types of NED Packages

A NED package allows NSO to manage a network device of a specific type. NEDs typically contain YANG models and the code, specifying how NSO should configure and retrieve status. When developing your own NED, there are four categories supported by NSO.

- A NETCONF NED is used with the NSO's built-in NETCONF client and requires no code. Only YANG models. This NED is suitable for devices that strictly follow the specification for the NETCONF protocol and YANG mappings to NETCONF targeting a standardized machine-to-machine interface.
- CLI NED targeted devices that use a Cisco-style CLI as a human-to-machine configuration interface. Various YANG extensions are used to annotate the YANG model representation of the device together with code converting data between NSO and device formats.
- A generic NED is typically used to communicate with non-CLI devices, such as devices using protocols like REST, TL1, Corba, SOAP, RESTCONF, or gNMI as a configuration interface. Even NETCONF-enabled devices often require a generic NED to function properly with NSO.
- NSO's built-in SNMP client can manage SNMP devices by supplying NSO with the MIBs, with some additional declarative annotations and code to handle the communication to the device. Usually, this legacy protocol is used to read state data. Albeit limited, NSO has support for configuring devices using SNMP.

In summary, the NETCONF and SNMP NEDs use built-in NSO clients; the CLI NED is model-driven, whereas the generic NED requires a Java program to translate operations toward the device.

Dumb Versus Capable Devices

NSO differentiates between managed devices that can handle transactions and devices that can not. This discussion applies regardless of NED type, i.e., NETCONF, SNMP, CLI, or Generic.

NEDs for devices that cannot handle abort must indicate so in the reply of the `newConnection()` method indicating that the NED wants a reverse diff in case of an abort. Thus, NSO has two different ways to abort a transaction towards a NED, invoke the `abort()` method with or without a generated reverse diff.

For non-transactional devices, we have no other way of trying out a proposed configuration change than to send the change to the device and see what happens.

The table below shows the seven different data-related callbacks that could or must be implemented by all NEDs. It also differentiates between 4 different types of devices and what the NED must do in each callback for the different types of devices.

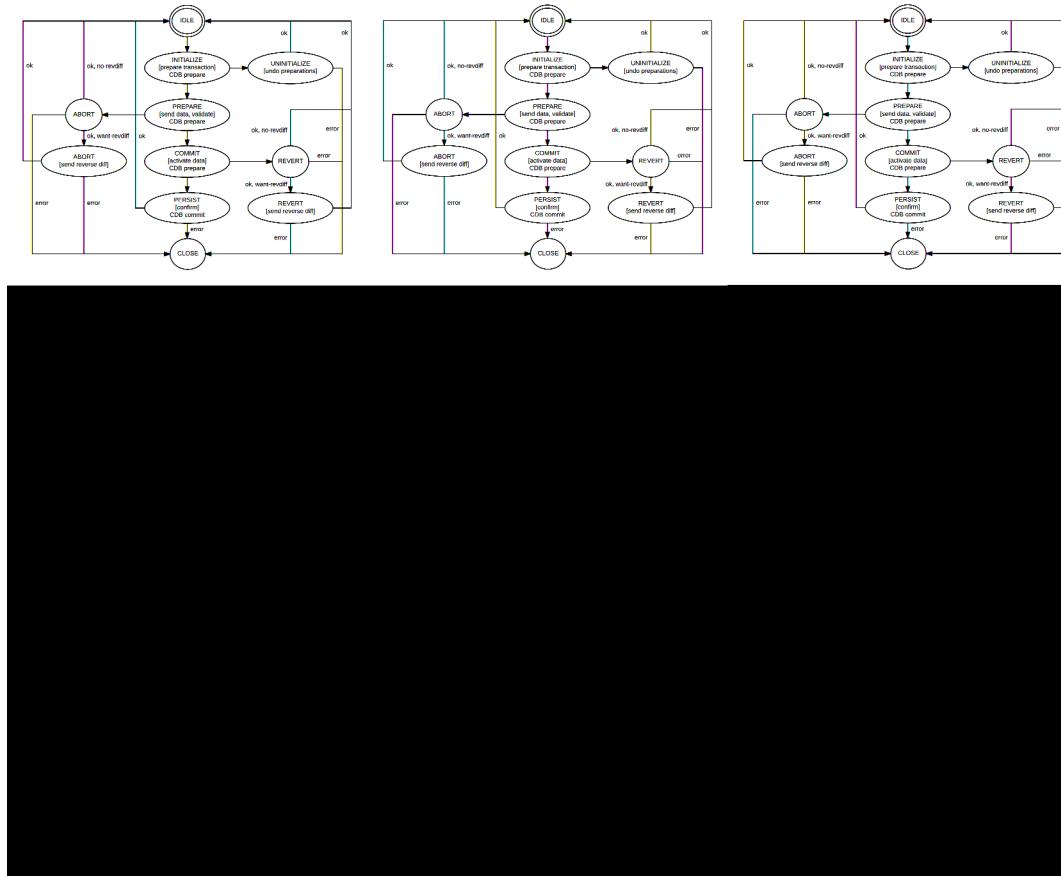
Table 184. Device types

Non transactional devices	Transactional devices	Transactional devices with confirmed commit	Fully capable NETCONF server
SNMP, Cisco IOS, NETCONF devices with startup+running	Devices that can abort, NETCONF devices without confirmed commit	Cisco XR type of devices	ConfD, Junos
<i>INITIALIZE</i> The initialize phase is used to initialize an transaction. For instance if locking or other transaction preparations are necessary they should be performed here. This callback is not mandatory to implement if no NED specific transaction preparations are needed.			
initialize(). NED code shall make the device go into config mode (if applicable) and lock (if applicable).	initialize(). NED code shall start a transaction on the device.	initialize(). NED code shall do the equivalent of configure exclusive.	Built in, NSO will lock.

Non transactional devices	Transactional devices	Transactional devices with confirmed commit	Fully capable NETCONF server
<i>UNINITIALIZE</i> If the transaction is not completed and the NED has done INITIALIZE this method is called to undo the transaction preparations, that is restoring the NED to the state before INITIALIZE. This callback is not mandatory to implement if no NED specific preparations was performed in INITIALIZE.			
uninitialize(). NED code shall unlock (if applicable).	uninitialize(). NED code shall abort the transaction.	uninitialize(). NED code shall abort the transaction.	Built in, NSO will unlock.
<i>PREPARE</i> In the prepare phase, the NEDs get exposed to all the changes that are destined for each managed device handled by each NED. It is the responsibility of the NED to determine the outcome here. If the NED replies successfully from the prepare phase, NSO assumes the device will be able to go through with the proposed configuration change.			
prepare(Data). NED code shall send all data to the device.	prepare(Data). NED code shall add Data to the transaction and validate.	prepare(Data). NED code shall add Data to the transaction and validate.	Built in, NSO will edit-config towards the candidate, validate and commit confirmed with a timeout.
<i>ABORT</i> If any participants in the transaction reject the proposed changes, all NEDs will be invoked in the abort() method for each managed device the NED handles. It is the responsibility of the NED to make sure that whatever was done in the PREPARE phase is undone. For NEDs that indicate as reply in newConnection() that they want the reverse diff, they will get the reverse data as a parameter here.			
abort(ReverseData null) Either do the equivalent of copy startup to running, or apply the ReverseData to the device.	abort(ReverseData null). Abort the transaction	abort(ReverseData null). Abort the transaction	Built in, discard-changes and close.
<i>COMMIT</i> Once all NEDs that get invoked in commit(Timeout) reply ok, the transaction is permanently committed to the system. The NED may still reject the change in COMMIT. If any NED reject the COMMIT, all participants will be invoked in REVERT, NEDs that support confirmed commit with a timeout, Cisco XR, may choose to use the provided timeout to make REVERT easy to implement.			
commit(Timeout). Do nothing	commit(Timeout). Commit the transaction.	commit(Timeout). Execute commit confirmed [Timeout] on the device.	Built in, commit confirmed with the timeout.
<i>REVERT</i> This state is reached if any NED reports failure in the COMMIT phase. Similar to the ABORT state, the reverse diff is supplied to the NED if the NED has asked for that.			
revert(ReverseData null) Either do the equivalent of copy startup to running, or apply the ReverseData to the device.	revert(ReverseData null) Either do the equivalent of copy startup to running, or apply the ReverseData to the device.	revert(ReverseData null). discard-changes	Built in, discard-changes and close.
<i>PERSIST</i> This state is reached at the end of a successful transaction. Here it's responsibility of the NED to make sure that if the device reboots, the changes are still there.			

Non transactional devices	Transactional devices	Transactional devices with confirmed commit	Fully capable NETCONF server
persist() Either do the equivalent of copy running to startup or nothing.	persist() Either do the equivalent of copy running to startup or nothing.	persist(). confirm.	Built in, commit confirm.

The following state diagram depicts the different states the NED code goes through in the life of a transaction.



NED transaction states

NETCONF NED Development

Creating and installing a NETCONF NED consists of the following steps:

- Make the device YANG data models available to NSO
- Build the NED package from the YANG data models using NSO tools
- Install the NED with NSO
- Configure the device connection and notification events in NSO

Creating a NETCONF NED that uses the built-in NSO NETCONF client can be a pleasant experience with devices and nodes that strictly follow the specification for the NETCONF protocol and YANG mappings to NETCONF. If the device does not, the smooth sailing will quickly come to a halt, and you are

recommended to visit the Chapter 14, *NED Administration* in *Administration Guide* and get help from the Cisco NSO NED team who can diagnose, develop and maintain NEDs that bypass misbehaving devices special quirks.

Tools for NETCONF NED Development

Before NSO can manage a NETCONF-capable device, a corresponding NETCONF NED needs to be loaded. While no code needs to be written for such NED, it must contain YANG data models for this kind of device. While in some cases, the YANG models may be provided by the device's vendor, devices that implement RFC 6022 YANG Module for NETCONF Monitoring can provide their YANG models using the functionality described in this RFC.

The NSO example under `$NCS_DIR/examples.ncs/development-guide/ned-development/netconf-ned` implements two shell scripts that use different tools to build a NETCONF NED from a simulated hardware chassis system controller device.

The netconf-console and ncs-make-package Tools

The **netconf-console** NETCONF client tool is a Python script that can be used for test, debug, and simple client duties. For example, making the device YANG models available to NSO using the NETCONF IETF RFC 6022 `get-schema` operation to download YANG modules and the RFC 6241`get` operation, where the device implements the RFC 7895 YANG module library to provide information about all the YANG modules used by the NETCONF server. Type **netconf-console -h** for documentation.

Once the required YANG models are downloaded or copied from the device, the **ncs-make-package** bash script tool can be used to create and build, for example, the NETCONF NED package. See `ncs-make-package(1)` in *Manual Pages* and **ncs-make-package -h** for documentation.

The `demo.sh` script in the `netconf-ned` example uses the **netconf-console** and **ncs-make-package** combination to create, build, and install the NETCONF NED. When you know beforehand which models you need from the device, you often begin with this approach when encountering a new NETCONF device.

The NETCONF NED Builder Tool

The NETCONF NED builder uses the functionality of the two previous tools to assist the NSO developer onboard NETCONF devices by fetching the YANG models from a device and building a NETCONF NED using CLI commands as a frontend.

The `demo_nb.sh` script in the `netconf-ned` example uses the NSO CLI NETCONF NED builder commands to create, build, and install the NETCONF NED. This tool can be beneficial for a device where the YANG models are required to cover the dependencies of the must-have models. Also, devices known to have behaved well with previous versions can benefit from using this tool and its selection profile and production packaging features.

Using the netconf-console and ncs-make-package Combination

For a demo of the steps below, see `README` in the `$NCS_DIR/examples.ncs/development-guide/ned-development/netconf-ned` example and run the `demo.sh` script.

Make the Device YANG Data Models Available to NSO

List the YANG version 1.0 models the device supports using NETCONF `hello` message.

```
$ netconf-console --port $DEVICE_NETCONF_PORT --hello | grep "module="
<capability>http://tail-f.com/ns/aaa/1.1?module=tailf-aaa&revision=2023-04-13</capability>
<capability>http://tail-f.com/ns/common/query?module=tailf-common-query&revision=2017-12-15</capability>
```

Using the netconf-console and ncs-make-package Combination

```
<capability>http://tail-f.com/ns/confd-progress?module=tailf-confd-progress&revision=2020-06
...
<capability>urn:ietf:params:xml:ns:yang:ietf-yang-metadata?module=ietf-yang-metadata&revision=
<capability>urn:ietf:params:xml:ns:yang:ietf-yang-types?module=ietf-yang-types&revision=2013
```

List the YANG version 1.1 models supported by the device from the device yang-library.

```
$ netconf-console --port=$DEVICE_NETCONF_PORT --get -x /yang-library/module-set/module/name
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
  <data>
    <yang-library xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library">
      <module-set>
        <name>common</name>
        <module>
          <name>iana-crypt-hash</name>
        </module>
        <module>
          <name>ietf-hardware</name>
        </module>
        <module>
          <name>ietf-netconf</name>
        </module>
        <module>
          <name>ietf-netconf-acm</name>
        </module>
        <module>
          ...
        </module>
        <module>
          <name>tailf-yang-patch</name>
        </module>
        <module>
          <name>timestamp-hardware</name>
        </module>
      </module-set>
    </yang-library>
  </data>
</rpc-reply>
```

The ietf-hardware.yang model is of interest to manage the device hardware. Use the **netconf-console** NETCONF get-schema operation to get the ietf-hardware.yang model.

```
$ netconf-console --port=$DEVICE_NETCONF_PORT \
  --get-schema=ietf-hardware > dev-yang/ietf-hardware.yang
```

The ietf-hardware.yang import a few YANG models.

```
$ cat dev-yang/ietf-hardware.yang | grep import
<import ietf-inet-types {
import ietf-yang-types {
import iana-hardware {
```

Two of the imported YANG models are shipped with NSO.

```
$ find ${NCS_DIR} \
  \(-name "ietf-inet-types.yang" -o -name "ietf-yang-types.yang" -o -name "iana-hardware.yang"
  /path/to/nso/src/ncs/builtin_yang/ietf-inet-types.yang
  /path/to/nso/src/ncs/builtin_yang/ietf-yang-types.yang
```

Use the **netconf-console** NETCONF get-schema operation to get the iana-hardware.yang module.

```
$ netconf-console --port=$DEVICE_NETCONF_PORT --get-schema=iana-hardware > \
  dev-yang/iana-hardware.yang
```

The `timestamp-hardware.yang` module augments a node onto the `ietf-hardware.yang` model. This is not visible in the YANG library. Therefore, information on the augment dependency must be available, or all YANG models must be downloaded and checked for imports and augments of the `ietf-hardware.yang` model to make use of the augmented node(s).

```
$ netconf-console --port=$DEVICE_NETCONF_PORT --get-schema=timestamp-hardware > \
dev-yang/timestamp-hardware.yang
```

Build the NED from the YANG Data Models

Create and build the NETCONF NED package from the device YANG models using the **ncs-make-package** script.

```
$ ncs-make-package --netconf-ned dev-yang --dest nso-rundir/packages/devsim --build \
--verbose --no-test --no-java --no-netsim --no-python --no-template --vendor "Tail-f" \
--package-version "1.0" devsim
```

If you make any changes to, for example, the YANG models after creating the package above, you can rebuild the package using **make -C nso-rundir/packages/devsim all**.

Configure the Device Connection

Start NSO. NSO will load the new package. If the package was loaded previously, use the **--with-package-reload** option. See `ncs(1)` in *Manual Pages* for details. If NSO is already running, use the **packages reload** CLI command.

```
$ ncs --cd ./nso-rundir
```

As communication with the devices being managed by NSO requires authentication, a custom authentication group will likely need to be created with mapping between the NSO user and the remote device username and password, SSH public-key authentication, or external authentication. The example used here has a 1-1 mapping between the NSO admin user and the ConfD-enabled simulated device admin user for both username and password.

In the example below, the device name is set to `hw0`, and as the device here runs on the same host as NSO, the NETCONF interface IP address is `127.0.0.1` while the port is set to `12022` to not collide with the NSO northbound NETCONF port. The standard NETCONF port, `830`, is used for production.

The `default` authentication group, as shown above, is used.

```
$ ncs_cli -u admin -C
# config
Entering configuration mode terminal
(config)# devices device hw0 address 127.0.0.1 port 12022 authgroup default
(config-device-hw0)# devices device hw0 trace pretty
(config-device-hw0)# state admin-state unlocked
(config-device-hw0)# device-type netconf ned-id devsim-nc-1.0
(config-device-hw0)# commit
Commit complete.
```

Fetch the public SSH host key from the device and sync the configuration covered by the `ietf-hardware.yang` from the device.

```
$ ncs_cli -u admin -C
# devices fetch-ssh-host-keys
fetch-result {
    device hw0
    result updated
    fingerprint {
        algorithm ssh-ed25519
        value 00:11:22:33:44:55:66:77:88:99:aa:bb:cc:dd:ee:ff
    }
}
```

```

}
# device device hw0 sync-from
result true

```

NSO can now configure the device, state data can be read, actions can be executed, and notifications can be received. See the `$NCS_DIR/examples.ncs/development-guide/ned-development/netconf-ned/demo.sh` example script for a demo.

Using the NETCONF NED Builder Tool

For a demo of the steps below, see README in the `$NCS_DIR/examples.ncs/development-guide/ned-development/netconf-ned` example and run the `demo_nb.sh` script.

Configure the Device Connection

As communication with the devices being managed by NSO requires authentication, a custom authentication group will likely need to be created with mapping between the NSO user and the remote device username and password, SSH public-key authentication, or external authentication.

The example used here has a 1-1 mapping between the NSO admin user and the ConfD-enabled simulated device admin user for both username and password.

```

admin@ncs# show running-config devices authgroups group
devices authgroups group default
  umap admin
    remote-name      admin
    remote-password $9$xr1xtyI/819xm9GxPqwzcEbQ6oaK7k5RHm96Hkgysg=
  !
  umap oper
    remote-name      oper
    remote-password $9$Pr2BRIHRSWOW2v85PvRGvU7DNehWL1hcP3t1+cIgaoE=
  !
!
```

In the example below, the device name is set to hw0, and as the device here runs on the same host as NSO, the NETCONF interface IP address is 127.0.0.1 while the port is set to 12022 to not collide with the NSO northbound NETCONF port. The standard NETCONF port, 830, is used for production.

The default authentication group, as shown above, is used.

```

# config
Entering configuration mode terminal
(config)# devices device hw0 address 127.0.0.1 port 12022 authgroup default
(config-device-hw0)# devices device hw0 trace pretty
(config-device-hw0)# state admin-state unlocked
(config-device-hw0)# device-type netconf ned-id netconf
(config-device-hw0)# commit

```



Note

A temporary NED identity is configured to `netconf` as the NED package has not yet been built. It will be changed to match the NETCONF NED package NED ID once the package is installed. The generic `netconf ned-id` allows NSO to connect to the device for basic NETCONF operations, such as `get` and `get-schema` for listing and downloading YANG models from the device.

Make the Device YANG Data Models Available to NSO

Create a NETCONF NED Builder project called `hardware` for the device, here named `hw0`.

```

# devtools true
# config

```

```
(config)# netconf-ned-builder project hardware 1.0 device hw0 local-user admin vendor Tail-f
(config)# commit
(config)# end
# show netconf-ned-builder project hardware
netconf-ned-builder project hardware 1.0
download-cache-path /path/to/ns0/examples.ncs/development-guide/ned-development/netconf-ned/
state/netconf-ned-builder/cache/hardware-nc-1.0
ned-directory-path /path/to/ns0/examples.ncs/development-guide/ned-development/netconf-ned/
state/netconf-ned-builder/hardware-nc-1.0
```

The NETCONF NED Builder is a developer tool that must be enabled first through the **devtools true** command. The NETCONF NED Builder feature is not expected to be used by the end users of NSO.

The cache directory above is where additional YANG and YANG annotation files can be added in addition to the ones download from the device. Files added need to be configured with the NED builder to be included with the project, as described below.

The project argument for the **netconf-ned-builder** command requires both the project name and a version number for the NED being built. A version number often picked is the version number of the device software version to match the NED to the device software it is tested with. NSO uses the project name and version number to create the NED name, here **hardware-nc-1.0**. The device's name is linked to the device name configured for the device connection.

Copying Manually to the Cache Directory



Note

This step is not required if the device supports the NETCONF **get-schema** operation and all YANG modules can be retrieved from the device.. Otherwise, you copy the YANG models to the **state/netconf-ned-builder/cache/hardware-nc-1.0** directory for use with the device.

After downloading the YANG data models and before building the NED with the NED builder, you need to register the YANG module with the NSO NED builder. For example, if you want to include a **dummy.yang** module with the NED, you first copy it to the cache directory and then, for example, create an XML file for use with the **ncs_load** command to update the NSO CDB operational datastore:

```
$ cp dummy.yang $NCS_DIR/examples.ncs/development-guide/ned-development/netconf-ned/\
nso-rundir/state/netconf-ned-builder/cache/hardware-nc-1.0/
$ cat dummy.xml
<config xmlns="http://tail-f.com/ns/config/1.0">
<netconf-ned-builder xmlns="http://tail-f.com/ns/ncs/netconf-ned-builder">
<project>
<family-name>hardware</family-name>
<major-version>1.0</major-version>
<module>
<name>dummy</name>
<revision>2023-11-10</revision>
<location>NETCONF</location>
<status>selected downloaded</status>
</module>
</project>
</netconf-ned-builder>
</config>
$ ncs_load -O -m -l dummy.xml
$ ncs_cli -u admin -C
# devtools true
# show netconf-ned-builder project hardware 1.0 module dummy 2023-11-10
SELECT    BUILD      BUILD
NAME      REVISION      NAMESPACE      FEATURE      LOCATION      STATUS
-----
```

```
dummy 2023-11-10 - - [ NETCONF ] selected,downloaded
```

Adding YANG Annotation Files

In some situations, you want to annotate the YANG data models that were downloaded from the device. For example, when an encrypted string is stored on the device, the encrypted value that is stored on the device will differ from the value stored in NSO if the two initialization vectors differ.

Say you have a YANG data model:

```
module dummy {
    namespace "urn:dummy";
    prefix dummy;

    revision 2023-11-10 {
        description
            "Initial revision.";
    }

    grouping my-grouping {
        container my-container {
            leaf my-encrypted-password {
                type tailf:aes-cfb-128-encrypted-string;
            }
        }
    }
}
```

And create a YANG annotation module:

```
module dummy-ann {
    namespace "urn:dummy-ann";
    prefix dummy-ann;

    import tailf-common {
        prefix tailf;
    }
    tailf:annotate-module "dummy" {
        tailf:annotate-statement "grouping[name='my-grouping']" {
            tailf:annotate-statement "container[name='my-container']" {
                tailf:annotate-statement "leaf[name=' my-encrypted-password']" {
                    tailf:ned-ignore-compare-config;
                }
            }
        }
    }
}
```

After downloading the YANG data models and before building the NED with the NED builder, you need to register the `dummy-ann.yang` annotation module, as was done above with the XML file for the `dummy.yang` module.

Using NETCONF get-schema with the NED Builder

If the device supports `get-schema` requests, the device can be contacted directly to download the YANG data models. The hardware system example returns the below YANG source files when the NETCONF `get-schema` operation is issued to the device from NSO. Only a subset of the list is shown.

```
$ ncs_cli -u admin -C
# devtools true
# devices fetch-ssh-host-keys
fetch-result {
    device hw0
```

```

        result updated
        fingerprint {
            algorithm ssh-ed25519
            value 00:11:22:33:44:55:66:77:88:99:aa:bb:cc:dd:ee:ff
        }
    }
# netconf-ned-builder project hardware 1.0 fetch-module-list
# show netconf-ned-builder project hardware 1.0 module
module iana-crypt-hash 2014-08-06
    namespace urn:ietf:params:xml:ns:yang:iana-crypt-hash
    feature [ crypt-hash-md5 crypt-hash-sha-256 crypt-hash-sha-512 ]
    location [ NETCONF ]
module iana-hardware 2018-03-13
    namespace urn:ietf:params:xml:ns:yang:iana-hardware
    location [ NETCONF ]
module ietf-datastores 2018-02-14
    namespace urn:ietf:params:xml:ns:yang:ietf-datastores
    location [ NETCONF ]
module ietf-hardware 2018-03-13
    namespace urn:ietf:params:xml:ns:yang:ietf-hardware
    location [ NETCONF ]
module ietf-inet-types 2013-07-15
    namespace urn:ietf:params:xml:ns:yang:ietf-inet-types
    location [ NETCONF ]
module ietf-interfaces 2018-02-20
    namespace urn:ietf:params:xml:ns:yang:ietf-interfaces
    feature [ arbitrary-names if-mib pre-provisioning ]
    location [ NETCONF ]
module ietf-ip 2018-02-22
    namespace urn:ietf:params:xml:ns:yang:ietf-ip
    feature [ ipv4-non-contiguous-netmasks ipv6-privacy-autoconf ]
    location [ NETCONF ]
module ietf-netconf 2011-06-01
    namespace urn:ietf:params:xml:ns:netconf:base:1.0
    feature [ candidate confirmed-commit rollback-on-error validate xpath ]
    location [ NETCONF ]
module ietf-netconf-acm 2018-02-14
    namespace urn:ietf:params:xml:ns:yang:ietf-netconf-acm
    location [ NETCONF ]
module ietf-netconf-monitoring 2010-10-04
    namespace urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring
    location [ NETCONF ]
...
module ietf-yang-types 2013-07-15
    namespace urn:ietf:params:xml:ns:yang:ietf-yang-types
    location [ NETCONF ]
module tailf-aaa 2023-04-13
    namespace http://tail-f.com/ns/aaa/1.1
    location [ NETCONF ]
module tailf-acm 2013-03-07
    namespace http://tail-f.com/yang/acm
    location [ NETCONF ]
module tailf-common 2023-10-16
    namespace http://tail-f.com/yang/common
    location [ NETCONF ]
...
module timestamp-hardware 2023-11-10
    namespace urn:example:timestamp-hardware
    location [ NETCONF ]

```

The **fetch-ssh-host-key** command fetches the public SSH host key from the device to set up NETCONF over SSH. The **fetch-module-list** command will look for existing YANG modules in the download-

cache-path folder, YANG version 1.0 models in the device NETCONF hello message, and issue a **get** operation to look for YANG version 1.1 models in the device yang-library. The **get-schema** operation fetches the YANG modules over NETCONF and puts them in the download-cache-path folder.

After the list of YANG modules is fetched, the retrieved list of modules can be shown. Select the ones you want to download and include in the NETCONF NED.

When you select a module with dependencies on other modules, the modules dependent on are automatically selected, such as those listed below for the `ietf-hardware` module including `iana-hardware`, `ietf-inet-types` and `ietf-yang-types`. To select all available modules, use the wild card for both fields. Use the **deselect** command to exclude modules previously included from the build.

```
$ ncs_cli -u admin -C
# devtools true
# netconf-ned-builder project hardware 1.0 module ietf-hardware 2018-03-13 select
# netconf-ned-builder project hardware 1.0 module timestamp-hardware 2023-11-10 select
# show netconf-ned-builder project hardware 1.0 module status
NAME          REVISION      STATUS
-----
iana-hardware 2018-03-13  selected,downloaded
ietf-hardware 2018-03-13  selected,downloaded
ietf-inet-types 2013-07-15 selected,pending
ietf-yang-types 2013-07-15 selected,pending
timestamp-hardware 2023-11-10 selected,pending

Waiting for NSO to download the selected YANG models (see demo-nb.sh for details)

NAME          REVISION      STATUS
-----
iana-hardware 2018-03-13  selected,downloaded
ietf-hardware 2018-03-13  selected,downloaded
ietf-inet-types 2013-07-15 selected,downloaded
ietf-yang-types 2013-07-15 selected,downloaded
timestamp-hardware 2023-11-10 selected,downloaded
```

Principles of Selecting the YANG Modules

Before diving into more details, the principles of selecting the modules for inclusion in the NED are crucial steps in building the NED and deserve to be highlighted.

The best practice recommendation is to select only the modules necessary to perform the tasks for the given NSO deployment to reduce memory consumption, for example, for the **sync-from** command, and improve upgrade wall-clock performance.

For example, suppose the aim of the NSO installation is exclusively to manage BGP on the device, and the necessary configuration is defined in a separate module. In that case, only this module and its dependencies need to be selected. If several services are running within the NSO deployment, it will be necessary to include more data models in the single NED that may serve one or many devices. However, if the NSO installation is used to, for example, take a full backup of the device's configuration, all device modules need to be included with the NED.

Selecting a module will also require selecting the module's dependencies, namely, modules imported by the selected modules, modules that augment the selected modules with the required functionality, and modules known to deviate from the selected module in the device's implementation.

Avoid selecting YANG modules that overlap where, for example, configuring one leaf will update another. Including both will cause NSO to get out-of-sync with the device after a NETCONF **edit-config** operation, forcing time-consuming sync operations.

Build the NED from the YANG Data Models

An NSO NED is a package containing the device YANG data models. The NED package must first be built, then installed with NSO, and finally, the package must be loaded for NSO to communicate with the device via NETCONF using the device YANG data models as the schema for what to configure, state to read, etc.

After the files have been downloaded from the device, they must be built before being used. The following example shows how to build a NED for the hw0 device.

```
# devtools true
# netconf-ned-builder project hardware 1.0 build-ned
# show netconf-ned-builder project hardware 1.0 build-status
build-status success
# show netconf-ned-builder project hardware 1.0 module build-warning
% No entries found.
# show netconf-ned-builder project hardware 1.0 module build-error
% No entries found.
# unhide debug
# show netconf-ned-builder project hardware 1.0 compiler-output
% No entries found.
```



Note

Build errors can be found in the build-error leaf under the module list entry. If there are errors in the build, resolve the issues in the YANG models, update them and their revision on the device, and download them from the device or place the YANG models in the cache as described earlier.

Warnings after building the NED can be found in the build-warning leaf under the module list entry. It is good practice to clean up build warnings in your YANG models.

A build error example:

```
# netconf-ned-builder project cisco-iosxr 6.6 build-ned
Error: Failed to compile NED bundle
# show netconf-ned-builder project cisco-iosxr 6.6 build-status
build-status error
# show netconf-ned-builder project cisco-iosxr 6.6 module build-error
module openconfig-telemetry 2016-02-04
  build-error at line 700: <error message>
```

The full compiler output for debugging purposes can be found in the compiler-output leaf under the project list entry. The compiler-output leaf is hidden by hide-group debug and may be accessed in the CLI using the **unhide debug** command if the hide-group is configured in ncs.conf. Example ncs.conf config:

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

For the ncs.conf configuration change to take effect, it must be either reloaded or NSO restarted. A reload using the **ncs_cmd** tool:

```
$ ncs_cmd -c reload
```

As the compilation will halt if an error is found in a YANG data model, it can be helpful to first check all YANG data models at once using a shell script plus the NSO yanger tool.

```
$ ls -1
```

```

check.sh
yang    # directory with my YANG modules
$ cat check.sh
#!/bin/sh
for f in yang/*.yang
do
    $NCS_DIR/bin/yanger -p yang $f
done

```

As an alternative to debugging the NED building issues inside an NSO CLI session, the **make-development-ned** action creates a development version of NED, which can be used to debug and fix the issue in the YANG module.

```

$ ncs_cli -u admin -C
# devtools true
(config)# netconf-ned-builder project hardware 1.0 make-development-ned in-directory /tmp
ned-path /tmp/hardware-nc-1.0
(config)# end
# exit
$ cd /tmp/hardware-nc-1.0/src
$ make clean all

```

YANG data models that do not compile due to YANG RFC compliance issues can either be updated in the cache folder directly or in the device and re-uploaded again through `get-schema` operation by removing them from the cache folder and repeating the previous process to rebuild the NED. The YANG modules can be deselected from the build if they are not needed for your use case.



Note Having device vendors update their YANG models to comply with the NETCONF and YANG standards can be time-consuming. Visit the Chapter 14, *NED Administration* in *Administration Guide* and get help from the Cisco NSO NED team, who can diagnose, develop and maintain NEDs that bypass misbehaving device's special quirks.

Export the NED package and Load

A successfully built NED may be exported as a `tar` file using the **export-ned** action. The `tar` file name is constructed according to the below naming convention.

```
ncs-<ncs-version>-<ned-family>-nc-<ned-version>.tar.gz
```

The user chooses the directory the file needs to be created in. The user *must* have write access to the directory. I.e., configure the NSO user with the same uid (id -u) as the non-root user:

```

$ id -u
501
$ ncs_cli -u admin -C
# devtools true
# config
(config)# aaa authentication users user admin uid 501
(config-user-admin)# commit
Commit complete.
(config-user-admin)# end
# netconf-ned-builder project hardware 1.0 export-ned to-directory \
    /path/to/ns0/examples.ncs/development-guide/ned-development/netconf-ned/ns0-rundir/packages
tar-file /path/to/ns0/examples.ncs/development-guide/ned-development/netconf-ned/
    ns0-rundir/packages/ncs-6.2-hardware-nc-1.0.tar.gz

```

When the NED package has been copied to the NSO run-time packages directory, the NED package can be loaded by NSO.

```

# 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 hardware-nc-1.0
    result true
}
# show packages | nomore
packages package hardware-nc-1.0
package-version 1.0
description      "Generated by NETCONF NED builder"
ncs-min-version [ 6.2 ]
directory        ./state/packages-in-use/1/hardware-nc-1.0
component hardware
ned netconf ned-id hardware-nc-1.0
ned device vendor Tail-f
oper-status up

```

Update the ned-id for the hw0 Device

When the NETCONF NED has been built for the hw0 device, the ned-id for hw0 needs to be updated before the NED can be used to manage the device.

```

$ ncs_cli -u admin -C
# show packages package hardware-nc-1.0 component hardware ned netconf ned-id
ned netconf ned-id hardware-nc-1.0
# config
(config)# devices device hw0 device-type netconf ned-id hardware-nc-1.0
(config-device-hw0)# commit
Commit complete.
(config-device-hw0)# end
# devices device hw0 sync-from
result true
# show running-config devices device hw0 config | nomore
devices device hw0
config
hardware component carbon
    class          module
    parent         slot-1-4-1
    parent-rel-pos 1040100
    alias          dummy
    asset-id       dummy
    uri            [ urn:dummy ]
!
hardware component carbon-port-4
    class          port
    parent         carbon
    parent-rel-pos 1040104
    alias          dummy-port
    asset-id       dummy
    uri            [ urn:dummy ]
!
...

```

NSO can now configure the device, state data can be read, actions can be executed, and notifications can be received. See the \$NCS_DIR/examples.ncs/development-guide/ned-development/netconf-ned/demo-nb.sh example script for a demo.

Remove a NED from NSO

Installed NED packages can be removed from NSO by deleting them from the NSO project's packages folder and then deleting the device and the NETCONF NED project through the NSO CLI. To uninstall a NED built for the device hw0:

```
$ ncs_cli -C -u admin
# devtools true
# config
(config)# no netconf-ned-builder project hardware 1.0
(config)# commit
Commit complete.
(config)# end
# packages reload
Error: The following modules will be deleted by upgrade:
hardware-nc-1.0: iana-hardware
hardware-nc-1.0: ietf-hardware
hardware-nc-1.0: hardware-nc
hardware-nc-1.0: hardware-nc-1.0
If this is intended, proceed with 'force' parameter.
# packages reload force

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

CLI NED Development

The CLI NED is a model-driven way to CLI script towards all Cisco like device. Some Java code is necessary for handling the corner cases a human-to-machine interface presents. The NSO CLI NED southbound of NSO shares a Cisco-style CLI engine with the northbound NSO CLI interface, and the CLI engine can thus run in both directions, producing CLI southbound and interpreting CLI data coming from southbound while presenting a CLI interface northbound. It is helpful to keep this in mind when learning and working with CLI NEDs.

- A sequence of Cisco CLI commands can be turned into the equivalent manipulation of the internal XML tree that represents the configuration inside NSO.
A YANG model, annotated appropriately, will produce a Cisco CLI. The user can enter Cisco commands, and NSO will parse the Cisco CLI commands using the annotated YANG model and change the internal XML tree accordingly. Thus, this is the CLI parser and interpreter. Model-driven.
- The reverse operation is also possible. Given two different XML trees, each representing a configuration state, in the netsim/ConfD case and NSO's northbound CLI interface, it represents the configuration of a single device, i.e., the device using ConfD as a management framework. In contrast, the NSO case represents the entire network configuration and can generate the list of Cisco commands going from one XML tree to another.
NSO uses this technology to generate CLI commands southbound when we manage Cisco-like devices.

It will become clear later in the examples how the CLI engine runs in forward and reverse mode. The key point though, is that the Cisco CLI NED Java programmer doesn't have to understand and parse the structure of the CLI, this is entirely done by the NSO CLI engine.

To implement a CLI NED, the following components are required:

- A YANG data model that describes the CLI. An important development tool here is netsim (ConfD), the Tail-f on-device management toolkit. For NSO to manage a CLI device, it needs a YANG file

with exactly the right annotations to produce precisely the managed device's CLI. A few examples exist in the NSO NED evaluation collection with annotated YANG models that render different Cisco CLI variants.

See, for example, `$NCS_DIR/packages/neds/dell-ftos` and `$NCS_DIR/packages/neds/cisco-nx`. Look for `tailf:cli-*` extensions in the NED `src/yang` directory YANG models.

Thus, to create annotated YANG files for a device with a Cisco-like CLI, the work procedure is to run netsim (ConfD) and write a YANG file which renders the correct CLI.

Furthermore, this YANG model must declare an identity with `ned:cli-ned-id` as a base.

- It is important to note that a NED only needs to cover certain aspects of the device. To have NSO manage a device with a Cisco-like CLI you do not have to model the entire device, only the commands intended to be used need be covered. When the `show()` callback issues its "show running-config [toptag]" command, and the device replies with data that is fed to NSO, NSO will ignore all command dump output that the loaded YANG models do not cover.

Thus, whichever Cisco-like device we wish to manage, we must first have YANG models from NSO that cover all aspects of the device we want to use. Once we have a YANG model, we load it into NSO and modify the example CLI NED class to return the NedCapability list of the device.

- The NED code gets to see all data from and to the device. If it's impossible or too hard to get the YANG model exactly right for all commands, a last resort is to let the NED code modify the data inline.
- The next thing required is a Java class that implements the NED. This is typically not a lot of code, and the existing example NED Java classes are easily extended and modified to fit other needs. The most important point of the Java NED class code is that the code can be oblivious to the CLI commands sent and received.

Java CLI NED code must implement the `CliNed` interface.

- `NedConnectionBase.java`. See `$NCS_DIR/java/jar/ncs-src.jar`. Use `jar xf ncs-src.jar` to extract the JAR file. Look for `src/com/tailf/ned/NedConnectionBase.java`.
- `NedCliBase.java`. See `$NCS_DIR/java/jar/ncs-src.jar`. Use `jar xf ncs-src.jar` to extract the JAR file. Look for `src/com/tailf/ned/NedCliBase.java`.

Thus, the Java NED class has the following responsibilities.

- It must implement the identification callbacks, i.e `modules()`, `type()`, and `identity()`
- It must implement the connection related callback methods `newConnection()`, `isConnection()` and `reconnect()`

NSO will invoke the `newConnection()` when it requires a connection to a managed device. The `newConnection()` method is responsible for connecting to the device, figuring out exactly what type of device it is, and returning an array of `NedCapability` objects.

```
public class NedCapability {
    public String str;
    public String uri;
    public String module;
    public String features;
    public String revision;
    public String deviations;
    ....
```

This is very much in line with how a NETCONF connect works and how the NETCONF client and server exchange hello messages.

- Finally the NED code must implement a series of data methods. For example the method `void prepare(NedWorker w, String data)` get a `String` object which is the set of Cisco CLI commands it shall send to the device.

In the other direction, when NSO wants to collect data from the device, it will invoke `void show(NedWorker w, String toptag)` for each tag found at the top of data model(s) loaded for that device. For example if the NED gets invoked with `show(w, "interface")` its responsibility is to invoke the relevant show configuration command for "interface", i.e `show running-config interface` over the connection to the device, and then dumbly reply with all data the device replies with. NSO will parse the output data and feed it into its internal XML trees.

NSO can order the `showPartial()` to collect part of the data if the NED announces the capability `http://tail-f.com/ns/ncs-ned/show-partial?path-format=FORMAT` in which `FORMAT` is of the followings:

- key-path: support regular instance keypath format
- top-tag: support top tags under the `/devices/device/config` tree
- cmd-path-full: support Cisco's CLI edit path with instances
- path-modes-only: support Cisco CLI mode path
- cmd-path-modes-only-existing: same as `path-mode-only` but NSO only supplies the path mode of existing nodes.

Writing a data model for a CLI NED

The idea is to write a YANG data model and feed that into the NSO CLI engine such that the resulting CLI mimics that of the device to manage. This is fairly straightforward once you have understood how the different constructs in YANG are mapped into CLI commands. The data model usually needs to be annotated with specific Tail-f CLI extension to tailor exactly how the CLI is rendered.

This chapter will describe how the general principles work and give a number of cook book style examples of how certain CLI constructs are modeled.

The CLI NED is primarily designed to be used with devices that has a CLI that is similar to the CLIs on a typical Cisco box (i.e. IOS, XR, NX-OS etc). However, if the CLI follows the same principles but with a slightly different syntax, it may still be possible to use a CLI NED if some of the differences are handled by the Java part of the CLI NED. This chapter will describe how this can be done.

Lets start with the basic data model to CLI mapping. YANG consists of three major elements: containers, lists, and leaves. For example

```
container interface {
list ethernet {
    key id;

    leaf id {
        type uint16 {
            range "0..66";
        }
    }

    leaf description {
        type string {
            length "1..80";
        }
    }
}
```

```
leaf mtu {
    type uint16 {
        range "64..18000";
    }
}
}
```

The basic rendering of the constructs are as follows. Containers are rendered as command prefixes which can be stacked at any depth. Leaves are rendered as commands that takes one parameter. Lists are rendered as submodes, where the key of the list is rendered as a submode parameter. The example above would result in the command

```
interface ethernet ID
```

for entering the interface ethernet submode. Interface is a container and is rendered as a prefix, ethernet is a list and is rendered as a submode. Two additional commands would be available in the submode

description WORD
mtu INTEGER<64-18000>

A typical configuration with two interfaces could look like this:

```
interface ethernet 0
description "customer a"
mtu 1400
!
interface ethernet 1
description "customer b"
mtu 1500
!
```

Note that it makes sense to add help texts to the data model since these texts will be visible in the NSO and help the user see the mapping between the J-style CLI in the NSO and the CLI on the target device. The data model above may look like the following with proper help texts.

```
container interface {
tailf:info "Configure interfaces";

list ethernet {
    tailf:info "FastEthernet IEEE 802.3";
    key id;

    leaf id {
        type uint16 {
            range "0..66";
            tailf:info "<0-66>;FastEthernet interface number";
        }
    }

    leaf description {
        type string {
            length "1..80";
            tailf:info "LINE;;Up to 80 characters describing this interface";
        }
    }

    leaf mtu {
        type uint16 {
            range "64..18000";
            tailf:info "<64-18000>;MTU size in bytes";
        }
    }
}
```

```
}
```

I will generally not include the help texts in the examples below to save some space but they should be present in a production data model.

Tweaking the basic rendering scheme

The basic rendering suffice in many cases but is also not enough in many situations. What follows is a list of ways to annotate the data model in order to make the CLI engine mimic a device.

Suppressing submodes

Sometimes you want a number of instances (a list) but do not want a submode. For example

```
container dns {
    leaf domain {
        type string;
    }
    list server {
        ordered-by user;
        tailf:cli-suppress-mode;
        key ip;

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

The above would result in the commands

```
dns domain WORD
dns server IPAddress
```

A typical show-config output may look like:

```
dns domain tail-f.com
dns server 192.168.1.42
dns server 8.8.8.8
```

Adding a submode

Sometimes you want a submode to be created without having a list instance, for example a submode called **aaa** where all aaa configuration is located.

This is done by using the tailf:cli-add-mode extension. For example:

```
container aaa {
    tailf:info "AAA view";
    tailf:cli-add-mode;
    tailf:cli-full-command;

    ...
}
```

This would result in the command **aaa** for entering the container. However, sometimes the CLI requires that a certain set of elements are also set when entering the submode, but without being a list. For example the police rules inside a policy map in the Cisco 7200.

```
container police {
    // To cover also the syntax where cir, bc and be
    // doesn't have to be explicitly specified
```

```

tailf:info "Police";
tailf:cli-add-mode;
tailf:cli-mode-name "config-pmap-c-police";
tailf:cli-incomplete-command;
tailf:cli-compact-syntax;
tailf:cli-sequence-commands {
    tailf:cli-reset-siblings;
}
leaf cir {
    tailf:info "Committed information rate";
    tailf:cli-hide-in-submode;
    type uint32 {
        range "8000..2000000000";
        tailf:info "<8000-2000000000>;Bits per second";
    }
}
leaf bc {
    tailf:info "Conform burst";
    tailf:cli-hide-in-submode;
    type uint32 {
        range "1000..512000000";
        tailf:info "<1000-512000000>;Burst bytes";
    }
}
leaf be {
    tailf:info "Excess burst";
    tailf:cli-hide-in-submode;
    type uint32 {
        range "1000..512000000";
        tailf:info "<1000-512000000>;Burst bytes";
    }
}
leaf conform-action {
    tailf:cli-break-sequence-commands;
    tailf:info "action when rate is less than conform burst";
    type police-action-type;
}
leaf exceed-action {
    tailf:info "action when rate is within conform and "+
        "conform + exceed burst";
    type police-action-type;
}
leaf violate-action {
    tailf:info "action when rate is greater than conform + "+
        "exceed burst";
    type police-action-type;
}
}

```

Here the leaves with the annotation tailf:cli-hide-in-submode is not present as commands once the submode has been entered, but are instead only available as options the police command when entering the police submode.

Commands with multiple parameters

Often a command is defined as taking multiple parameters in a typical Cisco CLI. This is achieved in the data model by using the annotations tailf:cli-sequence-commands, tailf:cli-compact-syntax, tailf:cli-drop-node-name and possibly tailf:cli-reset-siblings.

For example:

```
container udld-timeout {
```

```

tailf:info "LACP unidirectional-detection timer";
tailf:cli-sequence-commands {
    tailf:cli-reset-all-siblings;
}
tailf:cli-compact-syntax;
leaf "timeout-type" {
    tailf:cli-drop-node-name;
    type enumeration {
        enum fast {
            tailf:info "in unit of milli-seconds";
        }
        enum slow {
            tailf:info "in unit of seconds";
        }
    }
}
leaf "milli" {
    tailf:cli-drop-node-name;
    when "../timeout-type = 'fast'" {
        tailf:dependency "../timeout-type";
    }
    type uint16 {
        range "100..1000";
        tailf:info "<100-1000>;timeout in unit of "
            +"milli-seconds";
    }
}
leaf "secs" {
    tailf:cli-drop-node-name;
    when "../timeout-type = 'slow'" {
        tailf:dependency "../timeout-type";
    }
    type uint16 {
        range "1..60";
        tailf:info "<1-60>;timeout in unit of seconds";
    }
}
}

```

This results in the command

```
udld-timeout [fast <millisecs> | slow <secs> ]
```

The tailf:cli-sequence-commands annotation tells the CLI engine to process the leaves in sequence. The tailf:cli-reset-siblings tells the CLI to reset all leaves in the container if one is set. This is necessary in order to ensure that no lingering config remains from a previous invocation of the command where more parameters were configured. The tailf:cli-drop-node-name tells the CLI that the leaf name shouldn't be specified. The tailf:cli-compact-syntax annotation tells the CLI that the leaves should be formatted on one line, i.e. as

```
udld-timeout fast 1000
```

as opposed to

```
uldl-timeout fast
uldl-timeout 1000
```

without the annotation. When constructs are used to control if the numerical value should be the milli or the secs leaf.

This command could also be written using a choice construct as

```
container udld-timeout {
    tailf:cli-sequence-command;
```

```

choice udld-timeout-choice {
    case fast-case {
        leaf fast {
            tailf:info "in unit of milli-seconds";
            type empty;
        }
        leaf milli {
            tailf:cli-drop-node-name;
            must ".../fast" { tailf:dependency ".../fast"; }
            type uint16 {
                range "100..1000";
                tailf:info "<100-1000>;timeout in unit of "
                    +"milli-seconds";
            }
            mandatory true;
        }
    }
    case slow-case {
        leaf slow {
            tailf:info "in unit of milli-seconds";
            type empty;
        }
        leaf "secs" {
            must ".../slow" { tailf:dependency ".../slow"; }
            tailf:cli-drop-node-name;
            type uint16 {
                range "1..60";
                tailf:info "<1-60>;timeout in unit of seconds";
            }
            mandatory true;
        }
    }
}

```

Sometimes the tailf:cli-incomplete-command is used to ensure that all parameters are configured. The cli-incomplete-command only applies to the C- and I-style CLI. To ensure that prior leaves in a container is also configured when the configuration is written using J-style or Netconf proper 'must' declarations should be used.

Another example is this where tailf:cli-optional-in-sequence is used

```

list pool {
    tailf:cli-remove-before-change;
    tailf:cli-suppress-mode;
    tailf:cli-sequence-commands {
        tailf:cli-reset-all-siblings;
    }
    tailf:cli-compact-syntax;
    tailf:cli-incomplete-command;
    key name;
    leaf name {
        type string {
            length "1..31";
            tailf:info "WORD<length:1-31> Pool Name or Pool Group";
        }
    }
    leaf ipstart {
        mandatory true;
        tailf:cli-incomplete-command;
        tailf:cli-drop-node-name;
        type inet:ipv4-address {
    }
}

```

```

        tailf:info "A.B.C.D;;Start IP Address of NAT pool";
    }
}
leaf ipend {
    mandatory true;
    tailf:cli-incomplete-command;
    tailf:cli-drop-node-name;
    type inet:ipv4-address {
        tailf:info "A.B.C.D;;End IP Address of NAT pool";
    }
}
leaf netmask {
    mandatory true;
    tailf:info "Configure Mask for Pool";
    type string {
        tailf:info "/nn or A.B.C.D;;Configure Mask for Pool";
    }
}
leaf gateway {
    tailf:info "Gateway IP";
    tailf:cli-optional-in-sequence;
    type inet:ipv4-address {
        tailf:info "A.B.C.D;;Gateway IP";
    }
}
leaf ha-group-ip {
    tailf:info "HA Group ID";
    tailf:cli-optional-in-sequence;
    type uint16 {
        range "1..31";
        tailf:info "<1-31>;HA Group ID 1 to 31";
    }
}
leaf ha-use-all-ports {
    tailf:info "Specify this if services using this NAT pool "
        +"are transaction based (immediate aging)";
    tailf:cli-optional-in-sequence;
    type empty;
    when ".../ha-group-ip" {
        tailf:dependency ".../ha-group-ip";
    }
}
leaf vrid {
    tailf:info "VRRP vrid";
    tailf:cli-optional-in-sequence;
    when "not(.../ha-group-ip)" {
        tailf:dependency ".../ha-group-ip";
    }
    type uint16 {
        range "1..31";
        tailf:info "<1-31>;VRRP vrid 1 to 31";
    }
}
leaf ip-rr {
    tailf:info "Use IP address round-robin behavior";
    type empty;
}
}

```

The tailf:cli-optional-in-sequence means that the parameters should be processed in sequence but a parameter can be skipped. However, if a parameter is specified then only parameters later in the container can follow it.

It is also possible to have some parameters in sequence initially in the container, and then the rest in any order. This is indicated by the tailf:cli-break-sequence command. For example:

```
list address {
    key ip;
    tailf:cli-suppress-mode;
    tailf:info "Set the IP address of an interface";
    tailf:cli-sequence-commands {
        tailf:cli-reset-all-siblings;
    }
    tailf:cli-compact-syntax;
    leaf ip {
        tailf:cli-drop-node-name;
        type inet:ipv6-prefix;
    }
    leaf link-local {
        type empty;
        tailf:info "Configure an IPv6 link local address";
        tailf:cli-break-sequence-commands;
    }
    leaf anycast {
        type empty;
        tailf:info "Configure an IPv6 anycast address";
        tailf:cli-break-sequence-commands;
    }
}
```

where it is possible to write

```
ip 1.1.1.1 link-local anycast
```

as well as

```
ip 1.1.1.1 anycast link-local
```

Leaf values not really part of the key

Sometimes a command for entering a submode has parameters that are not really key values, i.e. not part of the instance identifier, but still needs to be given when entering the submode. For example

```
list service-group {
    tailf:info "Service Group";
    tailf:cli-remove-before-change;
    key "name";
    leaf name {
        type string {
            length "1..63";
            tailf:info "NAME<length:1-63>;SLB Service Name";
        }
    }
    leaf tcpudp {
        mandatory true;
        tailf:cli-drop-node-name;
        tailf:cli-hide-in-submode;
        type enumeration {
            enum tcp { tailf:info "TCP LB service"; }
            enum udp { tailf:info "UDP LB service"; }
        }
    }
}
```

Tweaking the basic rendering scheme

```

leaf backup-server-event-log {
    tailf:info "Send log info on back up server events";
    tailf:cli-full-command;
    type empty;
}
leaf extended-stats {
    tailf:info "Send log info on back up server events";
    tailf:cli-full-command;
    type empty;
}
...
}

```

In this case the `tcpudp` is a non-key leaf that needs to be specified as a parameter when entering the `service-group` submode. Once in the submode the commands `backup-server-event-log` and `extended-stats` are present. Leaves with the `tailf:cli-hide-in-submode` attribute are given after the last key, in the sequence they appear in the list.

It is also possible to allow leaf values to be entered in between key elements. For example:

```

list community {
    tailf:info "Define a community who can access the SNMP engine";
    key "read remote";
    tailf:cli-suppress-mode;
    tailf:cli-compact-syntax;
    tailf:cli-reset-container;
    leaf read {
        tailf:cli-expose-key-name;
        tailf:info "read only community";
        type string {
            length "1..31";
            tailf:info "WORD<length:1-31>;SNMPv1/v2c community string";
        }
    }
    leaf remote {
        tailf:cli-expose-key-name;
        tailf:info "Specify a remote SNMP entity to which the user belongs";
        type string {
            length "1..31";
            tailf:info "Hostname or A.B.C.D;;IP address of remote SNMP "
                +"entity(length: 1-31)";
        }
    }
}

leaf oid {
    tailf:info "specific the oid"; // SIC
    tailf:cli-prefix-key {
        tailf:cli-before-key 2;
    }
    type string {
        length "1..31";
        tailf:info "WORD<length:1-31>;The oid qvalue";
    }
}

leaf mask {
    tailf:cli-drop-node-name;
    type string {
        tailf:info "/nn or A.B.C.D;;The mask";
    }
}

```

```
}
```

Here we have a list that is not mapped to a submode. It has two keys, read and remote, an an optional oid that can be specified before the remote key. Finally after the last key an optional mask parameter can be specified. The use of the tailf:cli-expose-key-name means that the key names should be part of the command, which they are not by default. The above construct results in the commands

```
community read WORD [oid WORD] remote HOSTNAME [/nn or A.B.C.D]
```

The tailf:cli-reset-container attribute means that all leaves in the container will be reset if any leaf is given.

Change controlling annotations

Some devices requires that a setting is removed before it can be changed, for example the service-group list above. This is indicated with the tailf:cli-remove-before-change annotation. It can be used both on lists and on leaves. A leaf example:

```
leaf source-ip {
    tailf:cli-remove-before-change;
    tailf:cli-no-value-on-delete;
    tailf:cli-full-command;
    type inet:ipv6-address {
        tailf:info "X:X::X:X;;Source IPv6 address used by DNS";
    }
}
```

This means that the diff sent to the device will contain first a no source-ip command, followed by a new source-ip command to set the new value.

The data model also use the tailf:cli-no-value-on-delete annotation which means that the leaf value should not be present in the no command. With the annotation a diff to modify the source ip from 1.1.1.1 to 2.2.2.2 would look like

```
no source-ip
source-ip 2.2.2.2
```

and without the annotation as

```
no source-ip 1.1.1.1
source-ip 2.2.2.2
```

Ordered-by user lists

By default a diff for an ordered-by user list contains information about where a new item should be inserted. This is typically not supported by the device. Instead the commands (diff) to send the device needs to remove all items following the new item, and then reinsert the items in the proper order. This behavior is controlled using the tailf:cli-long-obu-diff annotation. For example

```
list access-list {
    tailf:info "Configure Access List";
    tailf:cli-suppress-mode;
    key id;
    leaf id {
        type uint16 {
            range "1..199";
        }
    }
    list rules {
        ordered-by user;
        tailf:cli-suppress-mode;
        tailf:cli-drop-node-name;
        tailf:cli-show-long-obu-diffs;
    }
}
```

```

        key "txt";
leaf txt {
    tailf:cli-multi-word-key;
    type string;
}
}
}
}

```

Suppose we have the access list

```
access-list 90 permit host 10.34.97.124
access-list 90 permit host 172.16.4.224
```

and we want to change this to

```
access-list 90 permit host 10.34.97.124
access-list 90 permit host 10.34.94.109
access-list 90 permit host 172.16.4.224
```

we would generate the diff

```
no access-list 90 permit host 172.16.4.224
access-list 90 permit host 10.34.94.109
access-list 90 permit host 172.16.4.224
```

with the tailf:cli-long-obu-diff. Without the annotation the diff would be

```
# after permit host 10.34.97.124
access-list 90 permit host 10.34.94.109
```

Default values

Often in a config when a leaf is set to its default value it is not displayed by the 'show running-config' command, but we still need to set it explicitly. Suppose we have the leaf 'state'. By default the value is 'active'.

```

leaf state {
    tailf:info "Activate/Block the user(s)";
    type enumeration {
        enum active {
            tailf:info "Activate/Block the user(s)";
        }
        enum block {
            tailf:info "Activate/Block the user(s)";
        }
    }
    default "active";
}

```

If the device state is 'block' and we cant to set it to 'active', i.e. the default value. The default behavior is to send

```
no state block
```

to the device. This will not work. The correct command sequence should be

```
state active
```

The way to achieve this is to do the following.

```

leaf state {
    tailf:info "Activate/Block the user(s)";
    type enumeration {
        enum active {

```

```

        tailf:info "Activate/Block the user(s)";
    }
    enum block {
        tailf:info "Activate/Block the user(s)";
    }
}
default "active";
tailf:cli-trim-default;
tailf:cli-show-with-default;
}

```

This way a value for 'state' will always be generated. This may seem unintuitive but the reason this works comes from how the diff is calculated. When generating the diff the target configuration and the desired configuration is compared (per line). The target config will be

state block

and the desired config will be

state active

This will be interpreted as a leaf value change and the resulting diff will be to set the new value, i.e. active.

However, without the 'cli-show-with-default' option the desired config will be an empty line, i.e. no value set. When we compare the two lines we get

(current config)

state block

(desired config)

<empty>

This will result in the command to remove the configured leaf, i.e.

state block

which does not work.

Understanding how the diffs are generated

What you see in the C-style CLI when you do 'show configuration' is the commands needed to go from the running config to the configuration you have in your current session. It usually corresponds to the command you have just issued in your CLI session, but not always.

The output is actually generated by comparing the two configurations, i.e. the running config and your current uncommitted configuration. It is done by running 'show running-config' on both the running config and your uncommitted config, and then comparing the output line by line. Each line is complemented by some meta information which makes it possible to generate a better diff.

For example, if you modify a leaf value, say set the mtu to 1400 and the previous value was 1500. The two configs will then be

```

interface FastEthernet0/0/1      interface FastEthernet0/0/1
mtu 1500                         mtu 1400
!

```

When we compare these configs the first line are the same -> no action but we remember that we have entered the FastEthernet0/0/1 submode. The second line differ in the value (the meta information associated with the lines have the path and the value). When we analyze the two lines we determine that a value_set has occurred. The default action when the value has been changed is to output the command for

setting the new value, i.e. mtu 1500. However, we also need to reposition to the current submode. If this is the first line we are outputting in the submode we need to issue the command

```
interface FastEthernet0/0/1
```

before issuing the mtu 1500 command.

Similarly, suppose a value has been removed, i.e. mtu used to be set but it is no longer present

```
interface FastEthernet0/0/1      interface FastEthernet0/0/1
!                                mtu 1400
!
```

As before, the first lines are equivalent, but the second line has ! in the new config, and mtu 1400 in the running config. This is analyzed as being a delete and the commands

```
interface FastEthernet0/0/1
no mtu 1400
```

are generated.

There are tweaks to this behavior. For example, some machines does not like the no command to include the old value but want instead the command

```
no mtu
```

We can instruct the CLI diff engine to behave in this way by using the YANG annotation tailf:cli-no-value-on-delete;

```
leaf mtu {
    tailf:cli-no-value-on-delete;
    type uint16;
}
```

It is also possible to tell the CLI engine to not include the element name in the delete operation. For example the command

```
aaa local-user password cipher "C>9=UF*^V/ 'Q=^Q`MAF4<1!!"
```

but the command to delete the password is

```
no aaa local-user password
```

The data model for this would be

```
// aaa local-user
container password {
    tailf:info "Set password";
    tailf:cli-flatten-container;
    leaf cipher {
        tailf:cli-no-value-on-delete;
        tailf:cli-no-name-on-delete;
        type string {
            tailf:info "STRING<1-16>/<24>;The UNENCRYPTED/"
            +"ENCRYPTED password string";
        }
    }
}
```

Modifying the Java part of the CLI NED

It is often necessary to do some minor modifications to the Java part of a CLI NED. There are mainly four functions that needs to be modified: connect, show, applyConfig, enter/exit config mode.

Connecting to a device

The CLI NED code should do a few things when the connect callback is invoked.

- Set up a connection to the device (usually ssh).
- If necessary send a secondary password to enter exec mode. Typically a Cisco IOS like CLI requires the user to give the **enable** command followed by a password.
- Verify that it is the right kind of device and respond to NSO with a list of capabilities. This is usually done by running the **show version** command, or equivalent, and parsing the output.
- Configure the CLI session on the device to not use pagination. This is normally done by setting the screen length to 0 (or infinity or disable). Optionally it may also fiddle with the idle time.

Some modifications may be needed in this section if the commands for the above differ from the Cisco IOS style.

Displaying the configuration of a device

The NSO will invoke the show() callback multiple times, one time for each top-level tag in the data model. Some devices have support for displaying just parts of the configuration, others do not.

For a device that cannot display only parts of a config the recommended strategy is to wait for a show() invocation with a well known top tag and send the entire config at that point. If, if you know that the data model has a top tag called **interface** then you can use code like:

```
public void show(NedWorker worker, String toptag)
    throws NedException, IOException {
    session.setTracer(worker);
    try {
        int i;

        if (toptag.equals("interface")) {
            session.print("show running-config | exclude able-management\n");
            ...
        } else {
            worker.showCliResponse("");
        }
    } catch (...) { ... }
}
```

From the point of NSO it is perfectly ok to send the entire config as a response to one of the requested toptags, and to send an empty response otherwise.

Often some filtering is required of the output from the device. For example, perhaps part of the configuration should not be sent to NSO, or some keywords replaced with other. Here follows some examples:

Stripping sections, headers and footers

Some devices start the output from **show running-config** with a short header, and some add a footer. Common headers are **Current configuration:** and a footer may be **end** or **return**. In the example below we strip out a header and remove a footer.

```
if (toptag.equals("interface")) {
    session.print("show running-config | exclude able-management\n");
    session.expect("show running-config | exclude able-management");

    String res = session.expect(".*#");
```

```

        i = res.indexOf("Current configuration :");
        if (i >= 0) {
            int n = res.indexOf("\n", i);
            res = res.substring(n+1);
        }

        i = res.lastIndexOf("\nend");
        if (i >= 0) {
            res = res.substring(0,i);
        }

        worker.showCliResponse(res);
    } else {
        // only respond to first toptag since the A10
        // cannot show different parts of the config.
        worker.showCliResponse("");
    }
}

```

Also, you may choose to only model part of a device configuration in which case you can strip out the parts that you have not modelled. For example stripping out the snmp configuration:

```

if (toptag.equals("context")) {
    session.print("show configuration\n");
    session.expect("show configuration");

    String res = session.expect(".*\n.*#\n");

    snmp = res.indexOf("\nsnmp");
    home = res.indexOf("\nsession-home");
    port = res.indexOf("\nport");
    tunnel = res.indexOf("\ntunnel");

    if (snmp >= 0) {
        res = res.substring(0,snmp)+res.substring(home,port)+res.substring(tunnel);
    } else if (port >= 0) {
        res = res.substring(0,port)+res.substring(tunnel);
    }

    worker.showCliResponse(res);
} else {
    // only respond to first toptag since the STOKEOS
    // cannot show different parts of the config.
    worker.showCliResponse("");
}

```

Removing keywords

Sometimes a device generates non-parsable commands in the output from **show running-config**. For example, some A10 devices adds a keyword **cpu-process** at the end of the ip route command, i.e.

```
ip route 10.40.0.0 /14 10.16.156.65 cpu-process
```

but it does not accept this keyword when a route is configured. The solution is to simply strip the keyword before sending the config to NSO, and to not include the keyword in the data model for the device. Code to do this may look like:

```

if (toptag.equals("interface")) {
    session.print("show running-config | exclude able-management\n");
    session.expect("show running-config | exclude able-management");

    String res = session.expect(".*#");
}

```

```

        // look for the string cpu-process and remove it
        i = res.indexOf(" cpu-process");
        while (i >= 0) {
            res = res.substring(0,i)+res.substring(i+12);
            i = res.indexOf(" cpu-process");
        }

        worker.showCliResponse(res);
    } else {
        // only respond to first toptag since the A10
        // cannot show different parts of the config.
        worker.showCliResponse("");
    }
}

```

Replacing keywords

Sometimes a device has some other names for delete than the standard **no** command found in a typical Cisco CLI. NSO will only generate **no** commands when, for example, an element does not exist (i.e. **no shutdown** for an interface), but the device may need **undo** instead. This can be dealt with as a simple transformation of the configuration before sending it to NSO. For example:

```

if (toptag.equals("aaa")) {
    session.print("display current-config\n");
    session.expect("display current-config");

    String res = session.expect("return");

    session.expect(".*>");

    // split into lines, and process each line
    lines = res.split("\n");

    for(i=0 ; i < lines.length ; i++) {
        int c;
        // delete the version information, not really config
        if (lines[i].indexOf("version ") == 1) {
            lines[i] = "";
        }
        else if (lines[i].indexOf("undo ") >= 0) {
            lines[i] = lines[i].replaceAll("undo ", "no ");
        }
    }

    worker.showCliResponse(join(lines, "\n"));
} else {
    // only respond to first toptag since the H3C
    // cannot show different parts of the config.
    // (well almost)
    worker.showCliResponse("");
}

```

Another example is the following situation. A device has a configuration for **port trunk permit vlan 1-3** and may at the same time have disallow some vlans using the command **no port trunk permit vlan 4-6**. Since we cannot use a **no** container in the config, we instead add a **disallow** container, and then rely on the Java-code to do some processing, e.g.:

```

container disallow {
    container port {
        tailf:info "The port of mux-vlan";
        container trunk {
            tailf:info "Specify current Trunk port's "

```

```
        +"characteristics";
    container permit {
        tailf:info "allowed VLANs";
        leaf-list vlan {
            tailf:info "allowed VLAN";
            tailf:cli-range-list-syntax;
            type uint16 {
                range "1..4094";
            }
        }
    }
}
```

And in the Java show() code:

```
if (toptag.equals("aaa")) {
    session.print("display current-config\n");
    session.expect("display current-config");

    String res = session.expect("return");

    session.expect(".*>");

    // process each line
    lines = res.split("\n");

    for(i=0 ; i < lines.length ; i++) {
        int c;
        if (lines[i].indexOf("no port") >= 0) {
            lines[i] = lines[i].replaceAll("no ", "disallow ");
        }
    }

    worker.showCliResponse(join(lines, "\n"));
} else {
    // only respond to first toptag since the H3C
    // cannot show different parts of the config.
    // (well almost)
    worker.showCliResponse("");
}
```

A similar transformation needs to take place when the NSO sends a configuration change to the device. A more detailed discussion about apply config modifications follow later but the corresponding code would in this case be:

```
lines = data.split("\n");
for (i=0 ; i < lines.length ; i++) {
    if (lines[i].indexOf("disallow port ") == 0) {
        lines[i] = lines[i].replace("disallow ", "undo ");
    }
}
```

Different quoting practices

If the way a device quotes strings differ from the way it can be modelled in NSO it can be handled in the Java code. For example, one device does not quote encrypted password strings which may contain odd characters like the command character !. Java code to deal with this may look like:

```
if (toptag.equals("aaa")) {  
    session.print("display current-config\n");
```

```

        session.expect("display current-config");

        String res = session.expect("return");

        session.expect(".*>");

        // process each line
        lines = res.split("\n");
        for(i=0 ; i < lines.length ; i++) {
            if ((c=lines[i].indexOf("cipher ")) >= 0) {
                String line = lines[i];
                String pass = line.substring(c+7);
                String rest;
                int s = pass.indexOf(" ");
                if (s >= 0) {
                    rest = pass.substring(s);
                    pass = pass.substring(0,s);
                } else {
                    s = pass.indexOf("\r");
                    if (s >= 0) {
                        rest = pass.substring(s);
                        pass = pass.substring(0,s);
                    }
                    else {
                        rest = "";
                    }
                }
                // find cipher string and quote it
                lines[i] = line.substring(0,c+7)+quote(pass)+rest;
            }
        }

        worker.showCliResponse(join(lines, "\n"));
    } else {
        worker.showCliResponse(" ");
    }
}

```

And similarly dequoting when applying a configuration.

```

lines = data.split("\n");
for (i=0 ; i < lines.length ; i++) {
    if ((c=lines[i].indexOf("cipher ")) >= 0) {
        String line = lines[i];
        String pass = line.substring(c+7);
        String rest;
        int s = pass.indexOf(" ");
        if (s >= 0) {
            rest = pass.substring(s);
            pass = pass.substring(0,s);
        } else {
            s = pass.indexOf("\r");
            if (s >= 0) {
                rest = pass.substring(s);
                pass = pass.substring(0,s);
            }
            else {
                rest = "";
            }
        }
        // find cipher string and quote it
        lines[i] = line.substring(0,c+7)+dequote(pass)+rest;
    }
}

```

Applying a config

NSO will send configuration to the device in three different callbacks: prepare(), abort(), and revert(). The Java code should issue these commands to the device but some processing of the commands may be necessary. Also, the ongoing CLI session needs to enter configure mode, issue the commands, and then exit configure mode. Some processing may be needed if the device has different keywords, or different quoting, as described under the "Displaying the configuration of a device" section above.

For example, if a device uses **undo** in place of **no** then the code may look like this, where **data** is the string of commands received from NSO:

```
lines = data.split("\n");
for (i=0 ; i < lines.length ; i++) {
    if (lines[i].indexOf("no ") == 0) {
        lines[i] = lines[i].replace("no ", "undo ");
    }
}
```

This relies on the fact that NSO will not have any indentation in the commands sent to the device (as opposed to the indentation usually present in the output from **show running-config**).

Tail-f CLI NED Annotations

The typical Cisco CLI has two major modes, operational mode and configure mode. In addition, the configure mode has submodes. For example, interfaces are configured in a submode that is entered by giving the command **interface <InterfaceType> <Number>**. Exiting a submode, i.e. giving the **exit** command, leaves you in the parent mode. Submodes can also be embedded in other submodes.

In a typical Cisco CLI, you do not necessary have to exit a submode in order to execute a command in a parent mode. In fact, the output of the command **show running-config** hardly contains any exit commands. Instead there is an exclamation mark, **!**, to indicate that a submode is done, which is only a comment. The config is formatted to rely on the fact that if a command isn't found in the current submode, the CLI engine searches for the command in its parent mode.

Another interesting mapping problem is how to interpret the **no** command when multiple leaves are given on a command line. Consider the model:

```
container foo {
    tailf:cli-compact-syntax;
    tailf:cli-sequence-commands;
    presence true;
    leaf a {
        type string;
    }
    leaf b {
        type string;
    }
    leaf c {
        type string;
    }
}
```

It corresponds to the command syntax **foo [a <word> [b <word> [c <word>]]]**, i.e. the following commands are valid:

```
foo
foo a <word>
foo a <word> b <word>
foo a <word> b <word> c <word>
```

Now what does it mean to write **no foo a <word> b <word> c <word>**? It could mean that only the **c** leaf should be removed, or it could mean that all leaves should be removed, and it may also mean that the **foo** container should be removed.

There is no clear principle here and no one right solution. The annotations are therefore necessary to help the diff engine figure out what to actually send to the device.

Annotations

The full set of annotations can be found in `tailf_yang_cli_extensions(5)` in *Manual Pages*. All annotation YANG extension are not applicable in an NSO context, but most are. The most commonly used annotations are (in alphabetical order):

tailf:cli-add-mode

Used for adding a submode in a container. The default rendering engine maps a container as a command prefix, and a list node as a submode. However, sometimes entering a submode does not require the user to give a specific instance. In these cases you can use the **tailf:cli-add-mode** on a container:

```
container system {
    tailf:info "For system events.";
    container "default" {
        tailf:cli-add-mode;
        tailf:cli-mode-name "cfg-acct-mlist";
        tailf:cli-delete-when-empty;
        presence true;
        container start-stop {
            tailf:info "Record start and stop without waiting";
            leaf group {
                tailf:info "Use Server-group";
                type aaa-group-type;
            }
        }
    }
}
```

In this example, the **tailf:cli-add-mode** annotations tells the CLI engine to render the **default** container as a submode, in other words there will be a command **system default** for entering the default container as a submode. All further commands will use that context as base. In the example above, the **default** container will only contain one command **start-stop group**, rendered from the **start-stop** container (rendered as a prefix) and the **group** leaf.

tailf:cli-allow-join-with-key

Tells the parser that the list name is allowed to be joined together with the first key, i.e. written without space in between. This is used to render, for example, the **interface FastEthernet** command where the list is **FastEthernet** and the key is the interface name. In a typical Cisco CLI they are allowed to be written both as **interface FastEthernet 1** and as **interface FastEthernet1**.

```
list FastEthernet {
    tailf:info "FastEthernet IEEE 802.3";
    tailf:cli-allow-join-with-key {
        tailf:cli-display-joined;
    }
    tailf:cli-mode-name "config-if";
    key name;
    leaf name {
        type string {
            pattern "[0-9]+.*";
        }
        tailf:info "<0-66>/<0-128>;FastEthernet interface number";
    }
}
```

```
}
```

In the above example, the **tailf:cli-display-joined** substatement is used to tell the command renderer that it should display a list item using the format without space.

tailf:cli-allow-join-with-value

This tells the parser that a leaf value is allowed to be written without space between the leaf name and the value. This is typically the case when referring to an interface. For example:

```
leaf FastEthernet {
    tailf:info "FastEthernet IEEE 802.3";
    tailf:cli-allow-join-with-value {
        tailf:cli-display-joined;
    }
    type string;
    tailf:non-strict-leafref {
        path "/ios:interface/ios:FastEthernet/ios:name";
    }
}
```

In the example above, a leaf **FastEthernet** is used to point to an existing interface. The command is allowed to be written both as **FastEthernet 1** and as **FastEthernet1**, when referring to FastEthernet interface 1. The substatements say which is the preferred format when rendering the command.

tailf:cli-prefix-key and tailf:cli-before-key

Normally, keys come before other leaves when a list command is used, and this is required in YANG. However, this is not always the case in Cisco style CLIs. For example the **route-map** command where the name and sequence numbers are the keys, but the leaf operation (permit or deny) is given in between the first and the second key. The **tailf:cli-prefix-key** annotation tells the parser to expect a given leaf before the keys, but the substatement **tailf:cli-before-key <N>** can be used to specify that the leaf should occur in between two keys. For example:

```
list route-map {
    tailf:info "Route map tag";
    tailf:cli-mode-name "config-route-map";
    tailf:cli-compact-syntax;
    tailf:cli-full-command;
    key "name sequence";
    leaf name {
        type string {
            tailf:info "WORD;;Route map tag";
        }
    }
    // route-map * #
    leaf sequence {
        tailf:cli-drop-node-name;
        type uint16 {
            tailf:info "<0-65535>;Sequence to insert to/delete from "
                +"existing route-map entry";
            range "0..65535";
        }
    }
    // route-map * permit
    // route-map * deny
    leaf operation {
        tailf:cli-drop-node-name;
        tailf:cli-prefix-key {
            tailf:cli-before-key 2;
        }
    }
}
```

```

    }
    type enumeration {
        enum deny {
            tailf:code-name "op_deny";
            tailf:info "Route map denies set operations";
        }
        enum permit {
            tailf:code-name "op_internet";
            tailf:info "Route map permits set operations";
        }
        default permit;
    }
}

```

A lot of things are going on in the example above, in addition to the **tailf:cli-prefix-key** and **tailf:cli-before-key** annotations. The **tailf:cli-drop-node-name** annotation tells the parser to ignore the name of the leaf (to not accept that as input, or render it when displaying the configuration).

tailf:cli-boolean-no

This tells the parser to render a leaf of type boolean as **no <leaf>** and **<leaf>** instead of the default **<leaf> false** and **<leaf> true**. The other alternative to this is to use a leaf of type empty and the **tailf:cli-show-no** annotation. The difference is subtle. A leaf with **tailf:cli-boolean-no** would not be displayed unless explicitly configured to either true or false, whereas a type empty leaf with **tailf:cli-show-no** would always be displayed if not set. For example:

```

leaf keepalive {
    tailf:info "Enable keepalive";
    tailf:cli-boolean-no;
    type boolean;
}

```

In the above example the **keepalive** leaf is set to true when the command **keepalive** is given, and to false when **no keepalive** is given. The well known **shutdown** command, on the other hand, is modeled as a type empty leaf with the **tailf:cli-show-no** annotation:

```

leaf shutdown {
    // Note: default to "no shutdown" in order to be able to bring if up.
    tailf:info "Shutdown the selected interface";
    tailf:cli-full-command;
    tailf:cli-show-no;
    type empty;
}

```

tailf:cli-sequence-commands and tailf:cli-break-sequence-commands

These annotations are used to tell the CLI to only accept leaves in a container in the same order as they appears in the data model. This is typically required when the leaf names are hidden using the **tailf:cli-drop-node-name** annotation. It is very common in the Cisco CLI that commands accept multiple parameters, and such commands must be mapped to setting of multiple leaves in the data model. For example the **aggregate-address** command in the router bgp submode:

```

// router bgp * / aggregate-address
container aggregate-address {
    tailf:info "Configure BGP aggregate entries";
    tailf:cli-compact-syntax;
    tailf:cli-sequence-commands {
        tailf:cli-reset-all-siblings;
    }
    leaf address {

```

```

tailf:cli-drop-node-name;
type inet:ipv4-address {
    tailf:info "A.B.C.D; ;Aggregate address";
}
leaf mask {
    tailf:cli-drop-node-name;
    type inet:ipv4-address {
        tailf:info "A.B.C.D; ;Aggregate mask";
    }
}
leaf advertise-map {
    tailf:cli-break-sequence-commands;
    tailf:info "Set condition to advertise attribute";
    type string {
        tailf:info "WORD; ;Route map to control attribute "
        +"advertisement";
    }
}
leaf as-set {
    tailf:info "Generate AS set path information";
    type empty;
}
leaf attribute-map {
    type string {
        tailf:info "WORD; ;Route map for parameter control";
    }
}
leaf as-override {
    tailf:info "Override matching AS-number while sending update";
    type empty;
}
leaf route-map {
    type string {
        tailf:info "WORD; ;Route map for parameter control";
    }
}
leaf summary-only {
    tailf:info "Filter more specific routes from updates";
    type empty;
}
leaf suppress-map {
    tailf:info "Conditionally filter more specific routes from "
    +"updates";
    type string {
        tailf:info "WORD; ;Route map for suppression";
    }
}
}

```

In the above example the **tailf:cli-sequence-commands** annotation tells the parser to require the leaves in the **aggregate-address** container to be entered in the same order as in the data model, i.e. first address then mask. Since these leaves also have the **tailf:cli-drop-node-name** annotation, it would be impossible for the parser to know which leaf to map the values to, unless the order of appearance was used. The **tailf:cli-break-sequence-commands** annotation on the advertise-map leaf tells the parser that from that leaf and onward the ordering is no longer important and the leaves can be entered in any order (and leaves can be skipped).

Two other annotations are often used in combination with **tailf:cli-sequence-commands**; **tailf:cli-reset-all-siblings** and **tailf:cli-compact-syntax**. The first tells the parser that all leaves should be reset when any leaf is entered, i.e. if the user first gives the command:

```
aggregate-address 1.1.1.1 255.255.255.0 as-set summary-only
```

This would result in the leaves address, mask, as-set, and summary-only being set in the configuration. However, if the user then entered:

```
aggregate-address 1.1.1.1 255.255.255.0 as-set
```

The assumed result of this is that summary-only is no longer configured, ie that all leaves in the container is zeroed out when the command is entered again. The **tailf:cli-compact-syntax** annotation tells the CLI engine to render all leaves in the rendered on a separate line.

```
aggregate-address 1.1.1.1
aggregate-address 255.255.255.0
aggregate-address as-set
aggregate-address summary-only
```

The above will be rendered on one line (compact-syntax) as:

```
aggregate-address 1.1.1.1 255.255.255.0 as-set summary-only
```

tailf:cli-case-insensitive

Tells the parser that this particular leaf should be allowed to be entered in case insensitive format. The reason this is needed is that some devices display a command in one case, and other display the same command in a different case. Normally command parsing is case sensitive. For example:

```
leaf dhcp {
    tailf:info "Default Gateway obtained from DHCP";
    tailf:cli-case-insensitive;
    type empty;
}
```

tailf:cli-compact-syntax

This annotation tells the CLI engine to render all leaves in the container on one command line, i.e. instead of the default rendering where each leaf is rendered on a separate line

```
aggregate-address 1.1.1.1
aggregate-address 255.255.255.0
aggregate-address as-set
aggregate-address summary-only
```

it should be rendered on one line (compact-syntax) as

```
aggregate-address 1.1.1.1 255.255.255.0 as-set summary-only
```

tailf:cli-delete-container-on-delete

Deleting items in the data base is tricky when using the Cisco CLI syntax. The reason is that **no <command>** is open to multiple interpretations in many cases, for example when multiple leaves are set in one command, or a presence container is set in addition to a leaf. For example:

```
container dampening {
    tailf:info "Enable event dampening";
    presence "true";
    leaf dampening-time {
        tailf:cli-drop-node-name;
        tailf:cli-delete-container-on-delete;
        tailf:info "<1-30>;Half-life time for penalty";
        type uint16 {
            range 1..30;
        }
    }
}
```

```

    }
}
```

This data model allows both the **dampening** command and the command **dampening 10**. When the command **no dampening 10** is issued, should both the dampening container and the leaf be removed, or only the leaf? The **tailf:cli-delete-container-on-delete** tells the CLI engine to also delete the container when the leaf is removed.

tailf:cli-delete-when-empty

This annotation tells the CLI engine to remove a list entry or a presence container when all content of the container or list instance has been removed. For example:

```

container access-class {
    tailf:info "Filter connections based on an IP access list";
    tailf:cli-compact-syntax;
    tailf:cli-sequence-commands;
    tailf:cli-reset-container;
    tailf:cli-flatten-container;
    list access-list {
        tailf:cli-drop-node-name;
        tailf:cli-compact-syntax;
        tailf:cli-reset-container;
        tailf:cli-suppress-mode;
        tailf:cli-delete-when-empty;
        key direction;
        leaf direction {
            type enumeration {
                enum "in" {
                    tailf:info "Filter incoming connections";
                }
                enum "out" {
                    tailf:info "Filter outgoing connections";
                }
            }
        }
        leaf access-list {
            tailf:cli-drop-node-name;
            tailf:cli-prefix-key;
            type exp-ip-acl-type;
            mandatory true;
        }
        leaf vrf-also {
            tailf:info "Same access list is applied for all VRFs";
            type empty;
        }
    }
}
```

In this case, the **tailf:cli-delete-when-empty** annotation tells the CLI engine to remove an access-list instance when it doesn't have neither an access-list nor a vrf-also child.

tailf:cli-diff-dependency

This annotations tells the CLI engine that there is a dependency between the current account when generating diff commands to send to the device, or when rendering the **show configuration** command output. It can have two different substatements: **tailf:cli-trigger-on-set** and **tailf:cli-trigger-on-all**.

Without substatements, it should be thought of as similar to a leaf-ref, i.e. if the dependency target is delete, first perform any modifications to this leaf. For example the redistribute ospf submode in router bgp:

```
// router bgp * / redistribute ospf *
list ospf {
    tailf:info "Open Shortest Path First (OSPF)";
    tailf:cli-suppress-mode;
    tailf:cli-delete-when-empty;
    tailf:cli-compact-syntax;
    key id;
    leaf id {
        type uint16 {
            tailf:info "<1-65535>;Process ID";
            range "1..65535";
        }
    }
    list vrf {
        tailf:info "VPN Routing/Forwarding Instance";
        tailf:cli-suppress-mode;
        tailf:cli-delete-when-empty;
        tailf:cli-compact-syntax;
        tailf:cli-diff-dependency "/ios:ip/ios:vrf";
        tailf:cli-diff-dependency "/ios:vrf/ios:definition";
        key name;
        leaf name {
            type string {
                tailf:info "WORD;;VPN Routing/Forwarding Instance (VRF) name";
            }
        }
    }
}
```

The **tailf:cli-diff-dependency "/ios:ip/ios:vrf"** tells the engine that if the **ip vrf** part of the configuration is deleted, then first display any changes to this part. This can be used when the device requires a certain ordering of the commands.

If the **tailf:cli-trigger-on-all** substatement is used, then it means that the target will always be displayed before the current node. Normally the order in the YANG file is used, but and it might not even be possible if they are embedded in a container.

The **tailf:cli-trigger-on-set** tells the engine that the ordering should be taken into account when this leaf is set and some other leaf is deleted. The other leaf should then be deleted before this is set. Suppose you have this data model:

```
list b {
    key "id";
    leaf id {
        type string;
    }
    leaf name {
        type string;
    }
    leaf y {
        type string;
    }
}
list a {
    key id;
    leaf id {
        tailf:cli-diff-dependency "/c[id=current()../id]" {
            tailf:cli-trigger-on-set;
        }
        tailf:cli-diff-dependency "/b[id=current()../id]";
        type string;
    }
}
```

```

}
list c {
    key id;
    leaf id {
        tailf:cli-diff-dependency "/a[id=current()../id]" {
            tailf:cli-trigger-on-set;
        }
        tailf:cli-diff-dependency "/b[id=current()../id]";
        type string;
    }
}

```

Then the **tailf:cli-diff-dependency "/b[id=current()../id]"** tells the CLI that before b list instance is deleted, the c instance with the same name needs to be changed.

```

tailf:cli-diff-dependency "/a[id=current()../id]" {
    tailf:cli-trigger-on-set;
}

```

This annotation, on the other hand, says that before this instance is created any changes to the a instance with the same name needs to be displayed.

Suppose you have the configuration:

```

b foo
!
a foo
!

```

Then created **c foo** and deleted **a foo**, it should be displayed as:

```

no a foo
c foo

```

If you then deleted **c foo** and created **a foo**, it should be rendered as:

```

no c foo
a foo

```

That is, in the reverse order.

tailf:cli-disallow-value

This annotation is used to disambiguate parsing. This is sometimes necessary when **tailf:cli-drop-node-name** is used. For example:

```

container authentication {
    tailf:info "Authentication";
    choice auth {
        leaf word {
            tailf:cli-drop-node-name;
            tailf:cli-disallow-value "md5|text";
            type string {
                tailf:info "WORD;;Plain text authentication string "
                +"(8 chars max)";
            }
        }
        container md5 {
            tailf:info "Use MD5 authentication";
            leaf key-chain {
                tailf:info "Set key chain";
                type string {

```

```
        tailf:info "WORD; ;Name of key-chain";  
    }  
}  
}  
}  
}
```

when the command **authentication md5...** is entered the CLI parser cannot determine if the leaf **word** should be set to the value "md5" or if the leaf **md5** should be set. By adding the **tailf:cli-disallow-value** annotation you can tell the CLI parser that certain regular expressions are not valid values. An alternative would be to add a restriction to the string type of **word** but this is much more difficult since restrictions can only be used to specify allowed values, not disallowed values.

tailf:cli-display-joined

See the description of `tailf:cli-allow-join-with-value` and `tailf:cli-allow-join-with-key`.

tailf:cli-display-separated

This annotation can be used on a presence container and tells the CLI engine that the container should be displayed as a separate command, even when a leaf in the container is set. The default rendering does not do this. For example:

```
container ntp {
    tailf:info "Configure NTP";
    // interface * / ntp broadcast
    container broadcast {
        tailf:info "Configure NTP broadcast service";
        //tailf:cli-display-separated;
        presence true;
        container client {
            tailf:info "Listen to NTP broadcasts";
            tailf:cli-full-command;
            presence true;
        }
    }
}
```

If both **broadcast** and **client** are created in the configuration then this will be displayed as:

```
ntp broadcast  
ntp broadcast client
```

When the **tailf:cli-display-separated** annotation is used. If the annotation isn't present then it would only be displayed as:

ntp broadcast client

The creation of the broadcast container would be implied.

tailf:cli-drop-node-name

This might be the most used annotation of them all. It can be used for multiple purposes. Primarily it tells the CLI engine that the node name should be ignored, which is typically needed when there is no corresponding leaf name in the command, typically when a command requires multiple parameters:

```
container exec-timeout {
    tailf:info "Set the EXEC timeout";
    tailf:cli-sequence-commands;
    tailf:cli-compact-syntax;
    leaf minutes {
```

```

        tailf:info "<0-35791>;Timeout in minutes";
        tailf:cli-drop-node-name;
        type uint32;
    }
leaf seconds {
    tailf:info "<0-2147483>;Timeout in seconds";
    tailf:cli-drop-node-name;
    type uint32;
}
}

```

However, it can also be used to introduce ambiguity, or a choice in the parse tree if you like. Suppose you need to support these commands:

```

// interface * / vrf forwarding
// interface * / ip vrf forwarding
choice vrf-choice {
    container ip-vrf {
        tailf:cli-no-keyword;
        tailf:cli-drop-node-name;
        container ip {
            container vrf {
                leaf forwarding {
                    tailf:info "Configure forwarding table";
                    type string {
                        tailf:info "WORD;;VRF name";
                    }
                    tailf:non-strict-leafref {
                        path "/ios:ip/ios:vrf/ios:name";
                    }
                }
            }
        }
    }
    container vrf {
        tailf:info "VPN Routing/Forwarding parameters on the interface";
        // interface * / vrf forwarding
        leaf forwarding {
            tailf:info "Configure forwarding table";
            type string {
                tailf:info "WORD;;VRF name";
            }
            tailf:non-strict-leafref {
                path "/ios:vrf/ios:definition/ios:name";
            }
        }
    }
}

// interface * / ip
container ip {
    tailf:info "Interface Internet Protocol config commands";
}

```

In the above case when the parser see the beginning of the command **ip**, it can interpret it as either entering the **interface */vrf-choice/ip-vrf/ip/vrf** config tree, or the **interface */ip** tree since the tokens consumed are the same in both branches. When the parser sees a **tailf:cli-drop-node-name** in the parse tree, it will try to match the current token stream to that parse tree, and if that fails backtrack and try other paths.

tailf:cli-exit-command

Tells the CLI engine to add an explicit exit command in the current submode. Normally, a submode does not have exit commands for leaving a submode, instead it is implied by the following command residing in

a parent mode. However, to avoid ambiguity it is sometimes necessary. For example, in the address-family submode:

```
container address-family {
    tailf:info "Enter Address Family command mode";
    container ipv6 {
        tailf:info "Address family";
        container unicast {
            tailf:cli-add-mode;
            tailf:cli-mode-name "config-router-af";
            tailf:info "Address Family Modifier";
            tailf:cli-full-command;
            tailf:cli-exit-command "exit-address-family" {
                tailf:info "Exit from Address Family configuration "
                +"mode";
            }
        }
    }
}
```

tailf:cli-explicit-exit

This tells the CLI engine to render explicit exit commands instead of the default ! when leaving a submode. The annotation is inherited by all submodes. For example:

```
container interface {
    tailf:info "Configure interfaces";
    tailf:cli-diff-dependency "/ios:vrf";
    tailf:cli-explicit-exit;
    // interface Loopback
    list Loopback {
        tailf:info "Loopback interface";
        tailf:cli-allow-join-with-key {
            tailf:cli-display-joined;
        }
        tailf:cli-mode-name "config-if";
        tailf:cli-suppress-key-abbreviation;
        // tailf:cli-full-command;
        key name;
        leaf name {
            type string {
                pattern "([0-9\\.])+";
                tailf:info "<0-2147483647>;Loopback interface number";
            }
        }
        uses interface-common-grouping;
    }
}
```

Without the **tailf:cli-explicit-exit** annotation, the edit sequences sent to the NED device will contain ! at the end of a mode, and rely on the next command to move from one submode to some other place in the CLI. This is the way the Cisco CLI usually works. However, it may cause problems if the next edit command is also a valid command in the current submode. Using **tailf:cli-explicit-exit** gets around this problem.

tailf:cli-expose-key-name

By default the key leaf names are not shown in the CLI, but sometimes you want them to be visible, for example:

```
// ip explicit-path name *
```

```

list explicit-path {
    tailf:info "Configure explicit-path";
    tailf:cli-mode-name "cfg-ip-expl-path";
    key name;
    leaf name {
        tailf:info "Specify explicit path by name";
        tailf:cli-expose-key-name;
        type string {
            tailf:info "WORD;;Enter name";
        }
    }
}

```

tailf:cli-flat-list-syntax

By default a leaf-list is rendered as a single line with the elements enclosed by [and]. If you want the values to be listed on one line this is the annotation to use. For example:

```

// class-map * / match cos
leaf-list cos {
    tailf:info "IEEE 802.1Q/ISL class of service/user priority values";
    tailf:cli-flat-list-syntax;
    type uint16 {
        range "0..7";
        tailf:info "<0-7>;Enter up to 4 class-of-service values"+
            " separated by white-spaces";
    }
}

```

tailf:cli-flatten-container

This annotation is a bit tricky. It tells the CLI engine that the container should be allowed to co-exist with leaves on the same command line, i.e. flattened. Normally, once the parser has entered a container it will not exit. However, if the container is flattened, the container will be exited once all leaves in the container have been entered. Also, a flattened container will be displayed together with sibling leaves on the same command line (provided the surrounding container has **tailf:cli-compact-syntax**).

Suppose you want to model the command **limit [inbound <int16> <int16>] [outbound <int16> <int16>] mtu <uint16>**. In other word the inbound and outbound settings are optional, but if you give inbound you have to specify two 16-bit integers, and you can always specify mtu.

```

container foo {
    tailf:cli-compact-syntax;
    container inbound {
        tailf:cli-compact-syntax;
        tailf:cli-sequence-commands;
        tailf:cli-flatten-container;
        leaf a {
            tailf:cli-drop-node-name;
            type uint16;
        }
        leaf b {
            tailf:cli-drop-node-name;
            type uint16;
        }
    }
    container outbound {
        tailf:cli-compact-syntax;
        tailf:cli-sequence-commands;
        tailf:cli-flatten-container;
        leaf a {
    
```

```

        tailf:cli-drop-node-name;
        type uint16;
    }
    leaf b {
        tailf:cli-drop-node-name;
        type uint16;
    }
}
leaf mtu {
    type uint16;
}
}

```

In the above example the **tailf:cli-flatten-container** tells the parser that it should exit the outbound/inbound container once both values have been entered. Without the annotation it would not be possible to exit the container once it has been entered. It would be possible to have the command **foo inbound 1 3** or **foo outbound 1 2** but not both at the same time, and not the final mtu leaf. The **tailf:cli-compact-syntax** annotation tells the renderer to display all leaves on the same line. If it wasn't used the line setting **foo inbound 1 2 outbound 3 4 mtu 1500** would be displayed as:

```

foo inbound 1
foo inbound 2
foo outbound 3
foo outbound 4
foo mtu 1500

```

The annotation **tailf:cli-sequence-commands** tells the CLI that the user has to enter the leaves inside the container in the specified order. Without this annotation it would not be possible to drop the names of the leaves and still have a deterministic parser. With the annotation, the parser knows that for the command **foo inbound 1 2**, leaf a should be assigned the value 1 and leaf b the value 2.

Another example:

```

container htest {
    tailf:cli-add-mode;
    container param {
        tailf:cli-hide-in-submode;
        tailf:cli-flatten-container;
        tailf:cli-compact-syntax;
        leaf a {
            type uint16;
        }
        leaf b {
            type uint16;
        }
    }
    leaf mtu {
        type uint16;
    }
}

```

The above model results in the command **htest param a <uint16> b <uint16>** for entering the submode. Once the submode has been entered, the command **mtu <uint16>** is available. Without the **tailf:cli-flatten-container** annotation it wouldn't be possible to use the **tailf:cli-hide-in-submode** annotation to attach the leaves to the command for entering the submode.

tailf:cli-full-command

This annotation tells the parser to not accept any more input beyond this element. By default the parser will allow setting of multiple leaves in the same command, and both enter a submode and set leaf values in the

submode. In most cases, it doesn't matter that the parser accepts commands that are not actually generated by the device in the output of **show running-config**. It is however needed to avoid ambiguity, or just to make the NSO CLI for the device more user friendly.

```
container transceiver {
    tailf:info "Select from transceiver configuration commands";
    container "type" {
        tailf:info "type keyword";
        // transceiver type all
        container all {
            tailf:cli-add-mode;
            tailf:cli-mode-name "config-xcvr-type";
            tailf:cli-full-command;
            // transceiver type all / monitoring
            container monitoring {
                tailf:info "Enable/disable monitoring";
                presence true;
                leaf interval {
                    tailf:info "Set interval for monitoring";
                    type uint16 {
                        tailf:info "<300-3600>;Time interval for monitoring "+ "transceiver in seconds";
                        range "300..3600";
                    }
                }
            }
        }
    }
}
```

In the above example it is possible to have the command **transceiver type all** for entering a submode, and then give the command **monitor [interval <300-3600>]**. If the **tailf:cli-full-command** annotation had not been used, the following would also have been a valid command: **transceiver type all monitor [interval <300-3600>]**. In the above example it doesn't make a difference as far as being able to parse the configuration on a device. The device will never show the oneline command syntax, but always display it as two lines, one for entering the submode and one for setting the monitor interval.

tailf:cli-full-no

This annotation tells the CLI parser that no further arguments should be accepted for this path when the path is traversed as an argument to the **no** command.

Example of use:

```
// event manager applet * / action * info
container info {
    tailf:info "Obtain system specific information";
    // event manager applet * / action info type
    container "type" {
        tailf:info "Type of information to obtain";
        tailf:cli-full-no;
        container snmp {
            tailf:info "SNMP information";
            // event manager applet * / action info type snmp var
            container var {
                tailf:info "Trap variable";
                tailf:cli-compact-syntax;
                tailf:cli-sequence-commands;
                tailf:cli-reset-container;
                leaf variable-name {
                    tailf:cli-drop-node-name;
                }
            }
        }
    }
}
```

tailf:cli-hide-in-submode

In some cases you need to give some parameters for entering a submode, but the submode cannot be modeled as a list, or the parameters should not be modeled as a key element of the list, but rather behaves as a leaf. In these cases you model the parameter as a leaf and use the **tailf:cli-hide-in-submode** annotation. It has two purposes, the leaf is displayed as part of the command for entering the submode when rendering the config, and the leaf is not available as a command in the submode.

For example:

```
// event manager applet *
list applet {
    tailf:info "Register an Event Manager applet";
    tailf:cli-mode-name "config-applet";
    tailf:cli-exit-command "exit" {
        tailf:info "Exit from Event Manager applet configuration submode";
    }
    key name;
    leaf name {
        type string {
            tailf:info "WORD;;Name of the Event Manager applet";
        }
    }
    // event manager applet * authorization
    leaf authorization {
        tailf:info "Specify an authorization type for the applet";
        tailf:cli-hide-in-submode;
        type enumeration {
            enum bypass {
                tailf:info "EEM aaa authorization type bypass";
            }
        }
    }
    // event manager applet * class
    leaf class {
        tailf:info "Specify a class for the applet";
        tailf:cli-hide-in-submode;
        type string {
            tailf:info "Class A-Z | default - default class";
            pattern "[A-Z]|default";
        }
    }
    // event manager applet * trap
    leaf trap {
        tailf:info "Generate an SNMP trap when applet is triggered.";
        tailf:cli-hide-in-submode;
        type empty;
    }
}
```

In the example above the key to the list is the **name** leaf, but in order to enter the submode the user may also give the arguments **event manager applet <name> [authorization bypass] [class <word>] [trap]**.

It is clear that these leaves are not keys to the list since giving the same name but different authorization, class or trap argument does not result in a new applet instance.

tailf:cli-incomplete-command

Tells the CLI that it should not be possible to hit **cr** after the current element. This is usually the case when a command takes multiple parameters, for example, given the following data model:

```
container foo {
    tailf:cli-compact-syntax;
    tailf:cli-sequence-commands;
    presence true;
    leaf a {
        type string;
    }
    leaf b {
        type string;
    }
    leaf c {
        type string;
    }
}
```

The valid commands are **foo [a <word> [b <word> [c <word>]]]**. If it however should be **foo a <word> b <word> [c <word>]**, i.e. the parameters a and b are mandatory, and c is optional, then the **tailf:cli-incomplete-command** annotation should be used as follows:

```
container foo {
    tailf:cli-compact-syntax;
    tailf:cli-sequence-commands;
    tailf:cli-incomplete-command;
    presence true;
    leaf a {
        tailf:cli-incomplete-command;
        type string;
    }
    leaf b {
        type string;
    }
    leaf c {
        type string;
    }
}
```

In other words, the command is incomplete after entering just **foo**, and also after entering **foo a <word>**, but not after **foo a <word> b <word>** or **foo a <word> b <word> c <word>**.

tailf:cli-incomplete-no

This annotation is similar to the **tailf:cli-incomplete-command** above, but applies to **no** commands. Sometimes you want to prevent the user from entering a generic **no** command. Suppose you have the data model:

```
container foo {
    tailf:cli-compact-syntax;
    tailf:cli-sequence-commands;
    tailf:cli-incomplete-command;
    presence true;
    leaf a {
        tailf:cli-incomplete-command;
        type string;
```

```

    }
leaf b {
    type string;
}
leaf c {
    type string;
}
}

```

Then it would be valid to write any of the following:

```

no foo
no foo a <word>
no foo a <word> b <word>
no foo a <word> b <word> c <word>

```

If you only want the last version of this to be a valid command, then you can use **tailf:cli-incomplete-no** to enforce this. For example:

```

container foo {
    tailf:cli-compact-syntax;
    tailf:cli-sequence-commands;
    tailf:cli-incomplete-command;
    tailf:cli-incomplete-no;
    presence true;
    leaf a {
        tailf:cli-incomplete-command;
        tailf:cli-incomplete-no;
        type string;
    }
    leaf b {
        tailf:cli-incomplete-no;
        type string;
    }
    leaf c {
        type string;
    }
}

```

tailf:cli-list-syntax

The default rendering of a leaf-list element is as a command taking a list of values enclosed in square brackets. Given the following element:

```

// class-map * / source-address
container source-address {
    tailf:info "Source address";
    leaf-list mac {
        tailf:info "MAC address";
        type string {
            tailf:info "H.H.H... H.H.H";
        }
    }
}

```

This would result in the command **source-address mac [H.H.H... H.H.H]**, instead of the desired **source-address mac H.H.H**. Given the configuration:

```

source-address {
    mac [ 1410.9fd8.8999 a110.9fd8.8999 bb10.9fd8.8999 ]
}

```

It should be rendered as:

```
source-address mac 1410.9fd8.8999
source-address mac a110.9fd8.8999
source-address mac bb10.9fd8.8999
```

This is achieved by adding the **tailf:cli-list-syntax** annotation. For example:

```
// class-map * / source-address
container source-address {
    tailf:info "Source address";
    leaf-list mac {
        tailf:info "MAC address";
        tailf:cli-list-syntax;
        type string {
            tailf:info "H.H.H; ;MAC address";
        }
    }
}
```

An alternative would be to model this as a list, i.e.:

```
// class-map * / source-address
container source-address {
    tailf:info "Source address";
    list mac {
        tailf:info "MAC address";
        tailf:cli-suppress-mode;
        key address;
        leaf address {
            type string {
                tailf:info "H.H.H; ;MAC address";
            }
        }
    }
}
```

In many cases, this may be the better choice. Notice how the **tailf:cli-suppress-mode** annotation is used to prevent the list from being rendered as a submode.

tailf:cli-mode-name

This annotation is not really needed when writing a NED. It is used to tell the CLI which prompt to use when in the submode. Without specific instructions, the CLI will invent a prompt based on the name of the submode container/list and the list instance. If a specific prompt is desired this annotation can be used. For example:

```
container transceiver {
    tailf:info "Select from transceiver configuration commands";
    container "type" {
        tailf:info "type keyword";
        // transceiver type all
        container all {
            tailf:cli-add-mode;
            tailf:cli-mode-name "config-xcvr-type";
            tailf:cli-full-command;
            // transceiver type all / monitoring
            container monitoring {
                tailf:info "Enable/disable monitoring";
                presence true;
                leaf interval {
                    tailf:info "Set interval for monitoring";
                    type uint16 {

```

```
        tailf:info "<300-3600>;;Time interval for monitoring "+  
        "transceiver in seconds";  
        range "300..3600";  
    }  
}  
}  
}  
}  
}
```

tailf:cli-multi-value

This annotation is used to indicate that a leaf should accept multiple tokens, and concatenate them. By default, only a single token is accepted as value to a leaf. If spaces are required then the value needs to be quoted. If this isn't desired the **tailf:cli-multi-value** annotation can be used to tell the parser that a leaf should accept multiple tokens. A common example of this is the description command. It is modeled as:

```
// event manager applet * / description
leaf "description" {
    tailf:info "Add or modify an applet description";
    tailf:cli-full-command;
    tailf:cli-multi-value;
    type string {
        tailf:info "LINE;;description";
    }
}
```

In the above example the `description` command will take all tokens to the end of the line, concatenate them with a space, and use that for leaf value. The `tailf:cli-full-command` annotation is used to tell the parser that no other command following this can be entered on the same command line. The parser would not be able to determine when the argument to this command ended and the next command commenced anyway.

tailf:cli-multi-word-key and tailf:cli-max-words

By default all key values consists of a single parser token, i.e. a string without spaces, or a quoted string. If multiple tokens should be accepted for a single key element, without quotes, then the **tailf:cli-multi-word-key** annotation can be used. The sub-annotation **tailf:cli-max-words** can be used to tell the parser that at most a fixed number of words should be allowed for the key. For example:

```
container permit {
    tailf:info "Specify community to accept";
    presence "Specify community to accept";
    list permit-list {
        tailf:cli-suppress-mode;
        tailf:cli-delete-when-empty;
        tailf:cli-drop-node-name;
        key expr;
        leaf expr {
            tailf:cli-multi-word-key {
                tailf:cli-max-words 10;
            }
            type string {
                tailf:info "LINE;;An ordered list as a regular-expression";
            }
        }
    }
}
```

The **tailf:cli-max-words** annotation can be used to allow more things to be entered on the same command line.

tailf:cli-no-name-on-delete and tailf:cli-no-value-on-delete

When generating delete commands towards the device, the default behaviour is to simply add "no" in front of the line you are trying to remove. However, this is not always allowed. In some cases only parts of the command is allowed. For example, suppose you have the data model:

```
container ospf {
    tailf:info "OSPF routes Administrative distance";
    leaf external {
        tailf:info "External routes";
        type uint32 {
            range "1.. 255";
            tailf:info "<1-255>;Distance for external routes";
        }
        tailf:cli-suppress-no;
        tailf:cli-no-value-on-delete;
        tailf:cli-no-name-on-delete;
    }
    leaf inter-area {
        tailf:info "Inter-area routes";
        type uint32 {
            range "1.. 255";
            tailf:info "<1-255>;Distance for inter-area routes";
        }
        tailf:cli-suppress-no;
        tailf:cli-no-name-on-delete;
        tailf:cli-no-value-on-delete;
    }
    leaf intra-area {
        tailf:info "Intra-area routes";
        type uint32 {
            range "1.. 255";
            tailf:info "<1-255>;Distance for intra-area routes";
        }
        tailf:cli-suppress-no;
        tailf:cli-no-name-on-delete;
        tailf:cli-no-value-on-delete;
    }
}
```

If the old configuration has the configuration **ospf external 3 inter-area 4 intra-area 1** then the default behaviour would be to send **no ospf external 3 inter-area 4 intra-area 1** but this would generate an error. Instead, the device simply wants **no ospf**. This is then achieved by adding **tailf:cli-no-name-on-delete** (telling the CLI engine to remove the element name from the no line), and **tailf:cli-no-value-on-delete** (telling the CLI engine to strip the leaf value from the command line to be sent).

tailf:cli-optional-in-sequence

This annotation is used in combination with **tailf:cli-sequence-commands**. It tells the parser that a leaf in the sequence isn't mandatory. Suppose you have the data model:

```
container foo {
    tailf:cli-compact-syntax;
    tailf:cli-sequence-commands;
    presence true;
    leaf a {
        tailf:cli-incomplete-command;
        type string;
    }
    leaf b {
        tailf:cli-incomplete-command;
    }
}
```

```

        type string;
    }
leaf c {
    type string;
}
}

```

If you want the command to behave as **foo a <word> [b <word>] c <word>**, it means that the leaves a and c are required and b is optional. If b is to be entered it must be entered after a and before c. This would be achieved by adding **tailf:cli-optional-in-sequence** in b.

```

container foo {
    tailf:cli-compact-syntax;
    tailf:cli-sequence-commands;
    presence true;
    leaf a {
        tailf:cli-incomplete-command;
        type string;
    }
    leaf b {
        tailf:cli-incomplete-command;
        tailf:cli-optional-in-sequence;
        type string;
    }
    leaf c {
        type string;
    }
}

```

A live example of this from the cisco-ios data model is:

```

// voice translation-rule * / rule *
list rule {
    tailf:info "Translation rule";
    tailf:cli-suppress-mode;
    tailf:cli-delete-when-empty;
    tailf:cli-incomplete-command;
    tailf:cli-compact-syntax;
    tailf:cli-sequence-commands {
        tailf:cli-reset-all-siblings;
    }
    ordered-by "user";
    key tag;
    leaf tag {
        type uint8 {
            tailf:info "<1-15>;Translation rule tag";
            range "1..15";
        }
    }
    leaf reject {
        tailf:info "Call block rule";
        tailf:cli-optional-in-sequence;
        type empty;
    }
    leaf "pattern" {
        tailf:cli-drop-node-name;
        tailf:cli-full-command;
        tailf:cli-multi-value;
        type string {
            tailf:info "WORD;;Matching pattern";
        }
    }
}

```

tailf:cli-prefix-key

This annotation is used when the key element of a list isn't the first value that you give when setting a list element (for example when entering a submode). This is similar to **tailf:cli-hide-in-submode**, except it allows the leaf values to be entered in between key elements. In the example below the match leaf is entered before giving the filter id.

```
container radius {
    tailf:info "RADIUS server configuration command";
    // radius filter *
    list filter {
        tailf:info "Packet filter configuration";
        key id;
        leaf id {
            type string {
                tailf:info "WORD;;Name of the filter (max 31 characters, longer will "
                           "+be rejected";
            }
        }
        leaf match {
            tailf:cli-drop-node-name;
            tailf:cli-prefix-key;
            type enumeration {
                enum match-all {
                    tailf:info "Filter if all of the attributes matches";
                }
                enum match-any {
                    tailf:info "Filter if any of the attributes matches";
                }
            }
        }
    }
}
```

It is also possible to have a sub-annotation to **tailf:cli-prefix-key** that specifies that the leaf should occur before a certain key position. For example:

```
list route-map {
    tailf:info "Route map tag";
    tailf:cli-mode-name "config-route-map";
    tailf:cli-compact-syntax;
    tailf:cli-full-command;
    key "name sequence";
    leaf name {
        type string {
            tailf:info "WORD;;Route map tag";
        }
    }
    // route-map * #
    leaf sequence {
        tailf:cli-drop-node-name;
        type uint16 {
            tailf:info "<0-65535>;Sequence to insert to/delete from "
                       +"existing route-map entry";
            range "0..65535";
        }
    }
    // route-map * permit
    // route-map * deny
    leaf operation {
        tailf:cli-drop-node-name;
        tailf:cli-prefix-key {
            tailf:cli-before-key 2;
```

```

    }
    type enumeration {
        enum deny {
            tailf:code-name "op_deny";
            tailf:info "Route map denies set operations";
        }
        enum permit {
            tailf:code-name "op_internet";
            tailf:info "Route map permits set operations";
        }
        default permit;
    }
    // route-map * / description
    leaf "description" {
        tailf:info "Route-map comment";
        tailf:cli-multi-value;
        type string {
            tailf:info "LINE;;Comment up to 100 characters";
            length "0..100";
        }
    }
}

```

The keys for this list are **name** and **sequence**, but in between you need to specify **deny** or **permit**. This is not a key since you cannot have two different list instances with the same name and sequence number, but differ in **deny** and **permit**.

tailf:cli-range-list-syntax

This annotation is used to group together list instances, or values in a leaf-list into ranges. The type of the value are not restricted to integer only. It works with a string also, and it is possible to have a value like this: 1-5, t1, t2.

```

// spanning-tree vlans-root
container vlans-root {
    tailf:cli-drop-node-name;
    list vlan {
        tailf:info "VLAN Switch Spanning Tree";
        tailf:cli-range-list-syntax;
        tailf:cli-suppress-mode;
        tailf:cli-delete-when-empty;
        key id;
        leaf id {
            type uint16 {
                tailf:info "WORD;;vlan range, example: 1,3-5,7,9-11";
                range "1..4096";
            }
        }
    }
}

```

What will exist in the database is separate instances, i.e. if the configuration is **vlan 1,3-5,7,9-11** this will result in the database having the instances 1,3,4,5,7,9,10, and 11. Similarly, to create these instances on the device, the command generated by NSO will be **vlan 1,3-5,7,9-11**. Without this annotation NSO would generate unique commands for each instance, i.e.:

```

vlan 1
vlan 2
vlan 3
vlan 5

```

```
vlan 7
...

```

Same thing for leaf-lists:

```
leaf-list vlan {
    tailf:info "Range of vlans to add to the instance mapping";
    tailf:cli-range-list-syntax;
    type uint16 {
        tailf:info "LINE;;vlan range ex: 1-65, 72, 300 -200";
    }
}
```

tailf:cli-remove-before-change

Some settings needs to be unset before they can be set. This can be accommodated by using the **tailf:cli-remove-before-change** annotation. An example of such a leaf is:

```
// ip vrf * / rd
leaf rd {
    tailf:info "Specify Route Distinguisher";
    tailf:cli-full-command;
    tailf:cli-remove-before-change;
    type rd-type;
}
```

You are not allowed to define a new route distinguisher before removing the old.

tailf:cli-replace-all

This annotation is used on leaf-lists to tell the CLI engine that the entire list should be written and not just the additions or subtractions, which is the default behaviour for leaf-lists. For example:

```
// controller * / channel-group
list channel-group {
    tailf:info "Specify the timeslots to channel-group "+ "mapping for an interface";
    tailf:cli-suppress-mode;
    tailf:cli-delete-when-empty;
    key number;
    leaf number {
        type uint8 {
            range "0..30";
        }
    }
    leaf-list timeslots {
        tailf:cli-replace-all;
        tailf:cli-range-list-syntax;
        type uint16;
    }
}
```

The **timeslots** leaf is changed by writing the entire range value. The default would be to generate commands for adding and deleting values from the range.

tailf:cli-reset-siblings and tailf:cli-reset-all-siblings

This annotation is a sub-annotation to **tailf:cli-sequence-commands**. The problem it addresses is what should happen when a command that takes multiple parameters is run a second time. Consider the data model:

```
container foo {
```

```

tailf:cli-compact-syntax;
tailf:cli-sequence-commands {
    tailf:cli-reset-siblings;
}
presence true;
leaf a {
    type string;
}
leaf b {
    type string;
}
leaf c {
    type string;
}
}

```

You are allowed to enter any of the below commands:

```

foo
foo a <word>
foo a <word> b <word>
foo a <word> b <word> c <word>

```

If you first enter the command **foo a 1 b 2 c 3**, what will be stored in the database is foo being present, the leaf a having the value 1, the leaf b having the value 2 and the leaf c having the value 3.

Now, if the command **foo a 3** is executed, it will set the value of leaf a to 3, but will leave leaf b and c as they were before. This is probably not the way the device works. In most cases it expects the leaves b and c to be unset. The annotation **tailf:cli-reset-siblings** tells the CLI engine that all siblings covered by the **tailf:cli-sequence-commands** should be reset.

Another similar case is when you have some leaves covered by the command sequencing, and some not. For example:

```

container foo {
    tailf:cli-compact-syntax;
    tailf:cli-sequence-commands {
        tailf:cli-reset-all-siblings;
    }
    presence true;
    leaf a {
        type string;
    }
    leaf b {
        tailf:cli-break-sequence-commands;
        type string;
    }
    leaf c {
        type string;
    }
}

```

The above model will allow the user to enter the b and c leaves in any order, as long as leaf a is entered first. The annotation **tailf:cli-reset-siblings** will reset the leaves up to the **tailf:cli-break-sequence-commands**. The **tailf:cli-reset-all-siblings** tells the CLI engine to reset all siblings, also those outside the command sequencing.

tailf:cli-reset-container

This annotation can be used on both containers/lists and on leaves, but has slightly different meaning. When used on a container it means that whenever the container is entered, all leaves in it are reset.

If used on a leaf it should be understood as whenever that leaf is set all other leaves in the container are reset. For example:

```
// license udi
container udi {
    tailf:cli-compact-syntax;
    tailf:cli-sequence-commands;
    tailf:cli-reset-container;
    leaf pid {
        type string;
    }
    leaf sn {
        type string;
    }
}
container ietf {
    tailf:info "IETF graceful restart";
    container helper {
        tailf:info "helper support";
        presence "helper support";
        leaf disable {
            tailf:cli-reset-container;
            tailf:cli-delete-container-on-delete;
            tailf:info "disable helper support";
            type empty;
        }
        leaf strict-lsa-checking {
            tailf:info "enable helper strict LSA checking";
            type empty;
        }
    }
}
```

tailf:cli-show-long-obu-diffs

Changes to lists that has the **ordered-by "user"** annotation are shown as insert, delete, and move operations. However, most devices do not support such operations on the lists. In these cases, if you want to insert an element in the middle of a list, you need to first delete all elements following the insertion point, add the new element, and then add all the elements you deleted. The **tailf:cli-show-long-obu-diffs** tells the CLI engine to do exactly this. For example:

```
list foo {
    ordered-by user;
    tailf:cli-show-long-obu-diffs;
    tailf:cli-suppress-mode;
    key id;
    leaf id {
        type string;
    }
}
```

If the old configuration is:

```
foo a
foo b
foo c
foo d
```

The desired configuration is:

```
foo a
foo b
```

```
foo e
foo c
foo d
```

NSO will send the following to the device:

```
no foo c
no foo d
foo e
foo c
foo d
```

An example from the cisco-ios model is:

```
// ip access-list extended *
container extended {
    tailf:info "Extended Access List";
    tailf:cli-incomplete-command;
    list ext-named-acl {
        tailf:cli-drop-node-name;
        tailf:cli-full-command;
        tailf:cli-mode-name "config-ext-nacl";
        key name;
        leaf name {
            type ext-acl-type;
        }
        list ext-access-list-rule {
            tailf:cli-suppress-mode;
            tailf:cli-delete-when-empty;
            tailf:cli-drop-node-name;
            tailf:cli-compact-syntax;
            tailf:cli-show-long-obu-diffs;
            ordered-by user;
            key rule;
            leaf rule {
                tailf:cli-drop-node-name;
                tailf:cli-multi-word-key;
                type string {
                    tailf:info "deny;;Specify packets to reject\n"+
                    "permit;;Specify packets to forwards\n"+
                    "remark;;Access list entry comment";
                    pattern "(permit.*)|(deny.*)|(no.*)|(remark.*)|([0-9]+.*)";
                }
            }
        }
    }
}
```

tailf:cli-show-no

One common CLI behaviour is to not only show when something is configured, but also when it isn't configured by displaying it as **no <command>**. You can tell the CLI engine that you want this behaviour by using the **tailf:cli-show-no** annotation. It can be used both on leaves and on presence containers. For example:

```
// ipv6 cef
container cef {
    tailf:info "Cisco Express Forwarding";
    tailf:cli-display-separated;
    tailf:cli-show-no;
    presence true;
}
```

and

```
// interface * / shutdown
leaf shutdown {
    // Note: default to "no shutdown" in order to be able to bring if up.
    tailf:info "Shutdown the selected interface";
    tailf:cli-full-command;
    tailf:cli-show-no;
    type empty;
}
```

However, this is a much more subtle behaviour than one may think and it is not obvious when the **tailf:cli-show-no** and the **tailf:cli-boolean-no** should be used. For example, it would also be possible to model the **shutdown** leaf a boolean value, i.e.:

```
// interface * / shutdown
leaf shutdown {
    tailf:cli-boolean-no;
    type boolean;
}
```

The problem with the above is that when a new interface is created, say a vlan interface, the **shutdown** leaf would not be set to anything and you would not send anything to the device. With the **cli-show-no** definition you would send **no shutdown** since the shutdown leaf would not be defined when a new interface vlan instance is created.

The boolean version can be tweaked to behave in a similar way using the **default** annotation and **tailf:cli-show-with-default**, i.e.:

```
// interface * / shutdown
leaf shutdown {
    tailf:cli-show-with-default;
    tailf:cli-boolean-no;
    type boolean;
    default "false";
}
```

The problem with this is that if you explicitly configure the leaf to false in NSO, you will send **no shutdown** to the device (which is fine), but if you then read the config from the device it will not display **no shutdown** since it now has its default setting. This will lead to an out-of-sync situation in NSO. NSO thinks the value should be set to false (which is different from the leaf not being set), whereas the device report the value as being unset.

The whole situation comes from the fact that NSO and the device treat default values differently. NSO considers a leaf as either being set or not set. If a leaf is set to its default value, it is still considered as set. A leaf must be explicitly deleted in order for it to become unset. Whereas a typical Cisco device considers a leaf unset if you set it to its default value.

tailf:cli-show-with-default

This tells the CLI engine to render a leaf not only when it is actually set, but also when it has its default value. For example:

```
leaf "input" {
    tailf:cli-boolean-no;
    tailf:cli-show-with-default;
    tailf:cli-full-command;
    type boolean;
    default true;
```

```
}
```

tailf:cli-suppress-list-no

Tells the CLI that it should not be possible to delete all lists instances, i.e. the command **no foo** is not allowed, it needs to be **no foo <instance>**. For example:

```
list class-map {
    tailf:info "Configure QoS Class Map";
    tailf:cli-mode-name "config-cmap";
    tailf:cli-suppress-list-no;
    tailf:cli-delete-when-empty;
    tailf:cli-no-key-completion;
    tailf:cli-sequence-commands;
    tailf:cli-full-command;
    // class-map *
    key name;
    leaf name {
        tailf:cli-disallow-value "type|match-any|match-all";
        type string {
            tailf:info "WORD;;class-map name";
        }
    }
}
```

tailf:cli-suppress-mode

By default all lists are rendered as submodes. This can be suppressed using the **tailf:cli-suppress-mode** annotation. For example, the data model:

```
list foo {
    key id;
    leaf id {
        type string;
    }
    leaf mtu {
        type uint16;
    }
}
```

If you have the configuration:

```
foo a {
    mtu 1400;
}
foo b {
    mtu 1500;
}
```

It would be rendered as:

```
foo a
mtu 1400
!
foo b
mtu 1500
!
```

However, if you add **tailf:cli-suppress-mode**:

```
list foo {
    tailf:cli-suppress-mode;
```

```

key id;
leaf id {
    type string;
}
leaf mtu {
    type uint16;
}
}

```

It will be rendered as:

```

foo a mtu 1400
foo b mtu 1500

```

tailf:cli-key-format

The format string is used when parsing a key value and when generating a key value for an existing configuration. The key items are numbered from 1-N and the format string should indicate how they are related by using \$(X) (where X is the key number). For example:

```

list interface {
    tailf:cli-key-format "$(1)/$(2)/$(3):$(4)";
    key "chassis slot subslot number";
    leaf chassis {
        type uint8 {
            range "1 .. 4";
        }
    }
    leaf slot {
        type uint8 {
            range "1 .. 16";
        }
    }
    leaf subslot {
        type uint8 {
            range "1 .. 48";
        }
    }
    leaf number {
        type uint8 {
            range "1 .. 255";
        }
    }
}

```

It will be rendered as:

```

interface 1/2/3:4

```

tailf:cli-recursive-delete

When generating configuration diffs delete all contents of a container or list before deleting the node. For example:

```

list foo {
    tailf:cli-recursive-delete;
    key "id";
    leaf id {
        type string;
    }
    leaf a {
        type uint8;
    }
}

```

```

    }
leaf b {
    type uint8;
}
leaf c {
    type uint8;
}
}

```

It will be rendered as:

```

# show full
foo bar
  a 1
  b 2
  c 3
!
# ex
# no foo bar
# show configuration
foo bar
  no a 1
  no b 2
  no c 3
!
no foo bar
#

```

tailf:cli-suppress-no

Specifies that the CLI should not auto-render 'no' commands for this element. An element with this annotation will not appear in the completion list to the 'no' command. For example:

```

list foo {
    tailf:cli-recursive-delete;
    key "id";
    leaf id {
        type string;
    }
    leaf a {
        type uint8;
    }
    leaf b {
        tailf:cli-suppress-no;
        type uint8;
    }
    leaf c {
        type uint8;
    }
}

```

It will be rendered as:

```

(config-foo-bar)# no ?
Possible completions:
  a
  c
  ---

```

The problem with the above is that the diff will still generate the **no**. To avoid it, you must use the **tailf:cli-no-value-on-delete** and **tailf:cli-no-name-on-delete**.

```
(config-foo-bar)# no ?
```

```
Possible completions:
  a
  c
  ---
  service    Modify use of network based services
(config-foo-bar)# ex
(config)# no foo bar
(config)# show config
foo bar
  no a 1
  no b 2
  no c 3
!
no foo bar
(config)#

```

tailf:cli-trim-default

Do not display value if it is same as default. Please note that this annotation works only in case of with-defaults basic-mode capability set to 'explicit' and the value is explicitly set by the user to the default value. For example:

```
list foo {
  key "id";
  leaf id {
    type string;
  }
  leaf a {
    type uint8;
    default 1;
  }
  leaf b {
    tailf:cli-trim-default;
    type uint8;
    default 2;
  }
}
```

It will be rendered as:

```
(config)# foo bar
(config-foo-bar)# a ?
Possible completions:
<unsignedByte>[1]
(config-foo-bar)# a 2 b ?
Possible completions:
<unsignedByte>[2]
(config-foo-bar)# a 2 b 3
(config-foo-bar)# commit
Commit complete.
(config-foo-bar)# show full
foo bar
  a 2
  b 3
!
(config-foo-bar)# a 1 b 2
(config-foo-bar)# commit
Commit complete.
(config-foo-bar)# show full
foo bar
  a 1
!
```

tailf:cli-embed-no-on-delete

Embed no in front of the element name instead of at the beginning of the line. For example:

```
list foo {
    key "id";
    leaf id {
        type string;
    }
    leaf a {
        type uint8;
    }
    container x {
        leaf b {
            type uint8;
            tailf:cli-embed-no-on-delete;
        }
    }
}
```

It will be rendered as:

```
(config-foo-bar)# show full
foo bar
  a 1
  x b 3
!
(config-foo-bar)# no x
(config-foo-bar)# show conf
foo bar
  x no b 3
!
```

tailf:cli-allow-range

Means that the non-integer key should allow range expressions. Can be used in key leafs only. The key must support a range format. The range applies only for matching existing instances. For example:

```
list interface {
    key name;
    leaf name {
        type string;
        tailf:cli-allow-range;
    }
    leaf number {
        type uint32;
    }
}
```

It will be rendered as:

```
(config)# interface eth0-100 number 90
Error: no matching instances found
(config)# interface
Possible completions:
  <name:string> eth0 eth1 eth2 eth3 eth4 eth5 range
(config)# interface eth0-3 number 100
(config-interface-eth0-3)# ex
(config)# interface eth4-5 number 200
(config-interface-eth4-5)# commit
Commit complete.
(config-interface-eth4-5)# ex
(config)# do show running-config interface
```

```

interface eth0
    number 100
!
interface eth1
    number 100
!
interface eth2
    number 100
!
interface eth3
    number 100
!
interface eth4
    number 200
!
interface eth5
    number 200
!
```

tailf:cli-case-sensitive

Specifies that this node is case-sensitive. If applied to a container or a list, any nodes below will also be case-sensitive. For example:

```

list foo {
    tailf:cli-case-sensitive;
    key "id";
    leaf id {
        type string;
    }
    leaf a {
        type string;
    }
}
```

It will be rendered as:

```

(config)# foo bar a test
(config-foo-bar)# ex
(config)# commit
Commit complete.
(config)# do show running-config foo
foo bar
    a test
!
(config)# foo bar a Test
(config-foo-bar)# ex
(config)# foo Bar a TEST
(config-foo-Bar)# commit
Commit complete.
(config-foo-Bar)# ex
(config)# do show running-config foo
foo Bar
    a TEST
!
foo bar
    a Test
!
```

tailf:cli-expose-ns-prefix

When used force the CLI to display namespace prefix of all children. For example:

```

list foo {
    tailf:cli-expose-ns-prefix;
    key "id";
    leaf id {
        type string;
    }
    leaf a {
        type uint8;
    }
    leaf b {
        type uint8;
    }
    leaf c {
        type uint8;
    }
}

```

It will be rendered as:

```

(config)# foo bar
(config-foo-bar)# ?
Possible completions:
example:a
example:b
example:c
---
```

tailf:cli-show-obu-comments

Enforces the CLI engine to generate 'insert' comments when displaying configuration changes of ordered-by user lists. Should not be used together with tailf:cli-show-long-obu-diffs. For example:

```

container policy {
    list policy-list {
        tailf:cli-drop-node-name;
        tailf:cli-show-obu-comments;
        ordered-by user;
        key policyid;
        leaf policyid {
            type uint32 {
                tailf:info "policyid;;Policy ID.";
            }
        }
        leaf-list srcintf {
            tailf:cli-flat-list-syntax {
                tailf:cli-replace-all;
            }
            type string;
        }
        leaf-list srcaddr {
            tailf:cli-flat-list-syntax {
                tailf:cli-replace-all;
            }
            type string;
        }
        leaf-list dstaddr {
            tailf:cli-flat-list-syntax {
                tailf:cli-replace-all;
            }
            type string;
        }
        leaf action {

```

```

type enumeration {
    enum accept {
        tailf:info "Action accept.";
    }
    enum deny {
        tailf:info "Action deny.";
    }
}

```

It will be rendered as:

```

admin@ncs(config-policy-4)# commit dry-run outformat cli
...
    policy {
        policy-list 1 {
            action accept;
            action deny;
        }
        # after policy-list 3
        policy-list 4 {
            srcintf aaa;
            srcaddr bbb;
            dstaddr ccc;
        }
    }
}
}
}

```

tailf:cli-multi-line-prompt

Tells the CLI to automatically enter multi-line mode when prompting the user for a value to this leaf. The user must type <CR> to enter in the multiline mode. For example:

```

leaf message {
    tailf:cli-multi-line-prompt;
    type string;
}

```

If configured on the same line, no prompt will appear and it will be rendered as:

```
(config)# message aaa
```

If <CR> typed, it will be rendered as:

```

(config)# message
(<string>) (aaa):
[Multiline mode, exit with ctrl-D.]
> Lorem ipsum dolor sit amet, consectetur adipiscing elit.
> Aenean commodo ligula eget dolor. Aenean massa.
> Cum sociis natoque penatibus et magnis dis parturient montes,
> nascetur ridiculus mus. Donec quam felis, ultricies nec,
> pellentesque eu, pretium quis, sem.
>
(config)# commit
Commit complete.

ubuntu(config)# do show running-config message
message "Lorem ipsum dolor sit amet, consectetur adipiscing elit. \nAenean
commodo ligula eget dolor. Aenean massa. \nCum sociis natoque penatibus et
magnis dis parturient montes, \nnascetur ridiculus mus. Donec quam felis,
ultricies nec,\n pellentesque eu, pretium quis, sem. \n"

```

```
(config)#
```

tailf:link target

This statement specifies that the data node should be implemented as a link to another data node, called the target data node. This means that whenever the node is modified, the system modifies the target data node instead, and whenever the data node is read, the system returns the value of target data node. Note that if the data node is a leaf, the target node MUST also be a leaf, and if the data node is a leaf-list, the target node MUST also be a leaf-list. The argument is an XPath absolute location path. If the target lies within lists, all keys must be specified. A key either has a value, or is a reference to a key in the path of the source node, using the function current() as starting point for an XPath location path. For example:

```
container foo {
    list bar {
        key id;
        leaf id {
            type uint32;
        }
        leaf a {
            type uint32;
        }
        leaf b {
            tailf:link "/example:foo/example:bar[id=current()../id]/example:a";
            type uint32;
        }
    }
}
```

It will be rendered as:

```
(config)# foo bar 1
ubuntu(config-bar-1)# ?
Possible completions:
  a
  b
  ---
  commit      Commit current set of changes
  describe    Display transparent command information
  exit        Exit from current mode
  help        Provide help information
  no          Negate a command or set its defaults
  pwd         Display current mode path
  top         Exit to top level and optionally run command
(config-bar-1)# b 100
(config-bar-1)# show config
foo bar 1
  b 100
!
(config-bar-1)# commit
Commit complete.
(config-bar-1)# show full
foo bar 1
  a 100
  b 100
!
(config-bar-1)# a 20
(config-bar-1)# commit
Commit complete.
(config-bar-1)# show full
foo bar 1
  a 20
```

```
b 20
!
```

Generic NED Development

As described in previous sections, the CLI NEDs are almost programming free. The NSO CLI engine takes care of parsing the stream of characters that come from "show running-config [toptag]" and also automatically produce the sequence of CLI commands required to take the system from one state to another.

A generic NED is required when we want to manage a device that neither speaks NETCONF or SNMP, nor can be modeled so that ConfD - loaded with those models - gets a CLI that looks almost/exactly like the CLI of the managed device. For example devices that have other proprietary CLIs, devices that can only be configured over other protocols such as REST, Corba, XML-RPC, SOAP, other proprietary XML solutions, etc.

In a manner similar to the CLI NED, the Generic NED needs to be able to connect to the device, return the capabilities, perform changes to the device and finally, grab the entire configuration of the device.

The interface that a Generic NED has to implement is very similar to the interface of a CLI NED. The main differences are:

- When NSO has calculated a diff for a specific managed device, it will for CLI NEDS also calculate the exact set of CLI commands to send to the device, according to the YANG models loaded for the device. In the case of a generic NED, NSO will instead send an array of operations to perform towards the device in the form of DOM manipulations. The generic NED class will receive an array of NedEditOp objects. Each NedEditOp object contains:
 - The operation to perform, i.e CREATED, DELETED, VALUE_SET etc.
 - The keypath to the object in case.
 - An optional value
- When NSO wants to sync the configuration from the device to NSO, the CLI NED only has to issue a series of "show running-config [toptag]" commands and reply with the output received from the device. A generic NED has to do more work. It is given a transaction handler, which it must attach to over the Maapi interface. Then the NED code must - by some means - retrieve the entire configuration and write into the supplied transaction, again using the Maapi interface.

Once the generic NED is implemented, all other functions in NSO work precisely in same manner as with NETCONF and CLI NED devices. NSO still has the capability to run network wide transactions. The caveat is that to abort a transaction towards a device that doesn't support transactions, we calculate the reverse diff and send it to the device, i.e. we automatically calculate the undo operations.

Another complication with generic NEDs is how the NED class shall authenticate towards the managed device. This depends entirely on the protocol between the NED class and the managed device. If SSH is used to a proprietary CLI, the existing authgroup structure in NSO can be used as is. However, if some other authentication data is needed, it is up the generic NED implementer to augment the authgroups in `tailf-ncs.yang` accordingly.

We must also configure a managed device, indicating that its configuration is handled by a specific generic NED. Below we see that the NED with identity "xmlrpc" is handling this device.

```
admin@ncs# show running-config devices device xl
address    127.0.0.1
port       12023
```

```

authgroup default
device-type generic ned-id xmlrpc
state admin-state unlocked
...

```

The example `examples.ncs/generic-ned/xmlrpc-device` in the NSO examples collection implements a generic NED that speaks XML-RPC to 3 HTTP servers. The HTTP servers run the apache XML-RPC server code and the NED code manipulates the 3 HTTP servers using a number of predefined XML RPC calls.

A good starting point when we wish to implement a new generic NED is the `ncs-make-package --generic-ned-skeleton ...` command, that be used to generate a skeleton package for a generic NED.

```

$ ncs-make-package --generic-ned-skeleton abc --build

$ ncs-setup --ned-package abc --dest ncs

$ cd ncs

$ ncs -c ncs.conf

$ ncs_cli -C -u admin

admin@ncs# show packages package abc
packages package abc
package-version 1.0
description      "Skeleton for a generic NED"
ncs-min-version [ 3.3 ]
component MyDevice
callback java-class-name [ com.example.abc.abcNed ]
ned generic ned-id abc
ned device vendor "Acme abc"
...
oper-status up

```

Getting started with a generic NED

A generic NED always requires more work than a CLI NED. The generic NED needs to know how to map arrays of `NedEditOp` objects into the equivalent reconfiguration operations on the device. Depending on the protocol and configuration capabilities of the device, this may be arbitrarily difficult.

Regardless of the device, we must always write a YANG model that describes the device. The array of `NedEditOp` objects that the generic NED code gets exposed to is relative the YANG model that we have written for the device. Again, this model doesn't necessarily have to cover all aspects of the device.

Often a useful technique with generic NEDs can be to write a pyang plugin to generate code for the generic NED. Again, depending on the device it may be possible to generate Java code from a pyang plugin that covers most or all aspects of mapping an array of `NedEditOp` objects into the equivalent reconfiguration commands for the device.

Pyang is an extensible and open source YANG parser (written by Tail-f) available at <http://www.yang-central.org>. pyang is also part of the NSO release. A number of plugins are shipped in the NSO release, for example `$NCS_DIR/lib/pyang/pyang/plugins/tree.py` is a good plugin to start with if we wish to write our own plugin.

`$NCS_DIR/examples.ncs/generic-ned/xmlrpc-device` is a good example to start with if we wish to write a generic NED. It manages a set of devices over the XML-RPC protocol. In this example we have:

- Defined a fictitious YANG model for the device.
- Implemented an XML-RPC server exporting a set of RPCs to manipulate that fictitious data model. The XML-RPC server runs the apache.org.apache.xmlrpc.server.XmlRpcServer Java package.
- Implemented a Generic NED which acts as an XML-RPC client speaking HTTP to the XML-RPC servers.

The example is self contained, and we can, using the NED code, manipulate these XML-RPC servers in a manner similar to all other managed devices.

```
$ cd $NCS_DIR/generic-ned/xmlrpc-device
$ make all start
$ ncs_cli -C -u admin
admin@ncs# devices sync-from

sync-result {
    device r1
    result true
}
sync-result {
    device r2
    result true
}
sync-result {
    device r3
    result true
}

admin@ncs# show running-config devices r1 config

ios:interface eth0
  macaddr      84:2b:2b:9e:af:0a
  ipv4-address 192.168.1.129
  ipv4-mask    255.255.255.0
  status       Up
  mtu         1500
  alias 0
    ipv4-address 192.168.1.130
    ipv4-mask    255.255.255.0
    !
  alias 1
    ipv4-address 192.168.1.131
    ipv4-mask    255.255.255.0
    !
  speed       100
  txqueuelen  1000
  !
```

Tweaking the order of NedEditOp objects

As it was mentioned earlier the NedEditOp objects are relative to the YANG model of the device, and they are to be translated into the equivalent reconfiguration operations on the device. Applying reconfiguration operations may only be valid in a certain order.

For Generic NEDs NSO provides a feature to ensure dependency rules being obeyed when generating a diff to commit. It controls the order of operations delivered in the NedEditOp array. The feature is activated by adding the following option to package-meta-data.xml:

```
<option>
  <name>ordered-diff</name>
</option>
```

When the `ordered-diff` flag is set the `NedEditOp` objects follow YANG schema order and consider dependencies *between leaf nodes*. Dependencies can be defined using `leafrefs` and the `tailf:cli-diff-after`, `tailf:cli-diff-create-after`, `tailf:cli-diff-modify-after`, `tailf:cli-diff-set-after`, `tailf:cli-diff-delete-after` YANG extensions. Read more about the above YANG extensions in the Tail-f CLI YANG extensions man page.

NED commands

A device we wish to manage using a NED usually has a not just configuration data that we wish to manipulate from NSO, but the device usually has a set of commands that do not relate to configuration.

The commands on the device we wish to be able to invoke from NSO must be modelled as actions. We model this as actions, and compile it using a special `ncsc` command to compile NED data models that do not directly relate to configuration data on the device.

The NSO example `$NCS_DIR/examples.ncs/generic-ned/xmlrpc-device` contains an example where the managed device, a fictitious XML-RPC device contains a YANG snippet :

```
container commands {
    tailf:action idle-timeout {
        tailf:actionpoint ncsinternal {
            tailf:internal;
        }
        input {
            leaf time {
                type int32;
            }
        }
        output {
            leaf result {
                type string;
            }
        }
    }
}
```

When that action YANG is imported into NSO it ends up under the managed device. We can invoke the action *on* the device as:

```
admin@ncs# devices device r1 config ios:commands idle-timeout time 55
result OK
```

The NED code is obviously involved here. All NEDs must always implement:

```
void command(NedWorker w, String cmdName, ConfXMLParam[] params)
throws NedException, IOException;
```

The `command()` method gets invoked in the NED, the code must then execute the command. The input parameters in the `params` parameter correspond to the data provided in the action. The `command()` method must reply with another array of `ConfXMLParam` objects.

```
public void command(NedWorker worker, String cmdname, ConfXMLParam[] p)
throws NedException, IOException {
    session.setTracer(worker);
```

```

        if (cmdname.compareTo("idle-timeout") == 0) {
            worker.commandResponse(new ConfXMLParam[] {
                new ConfXMLParamValue(new interfaces(),
                    "result",
                    new ConfBuf("OK"))
            });
        }
    }
}

```

The above code is fake, on a real device, the job of the `command()` method is to establish a connection to the device, invoke the command, parse the output and finally reply with an `ConfXMLParam` array.

The purpose of implementing NED commands is usually that we want to expose device commands to the programmatic APIs in the NSO DOM tree.

The SNMP NED

Introduction

NSO can use SNMP to configure a managed device, under certain circumstances. SNMP in general is not suitable for configuration, and it is important to understand why:

- In SNMP, the size of a SET request, which is used to write to a device, is limited to what fits into one UDP packet. This means that a large configuration change must be split into many packets. Each such packet contains some parameters to set, and each such packet is applied on its own by the device. If one SET request out of many fails, there is no abort command to undo the already applied changes, meaning that rollback is very difficult.
- The data modelling language used in SNMP, SMIv2, does not distinguish between configuration objects and other writable objects. This means that it is not possible to retrieve only the configuration from a device without explicit, exact knowledge of all objects in all MIBs supported by the device.
- SNMP supports only two basic operations, read and write. There is no protocol support for creating or deleting data. Such operations must be modeled in the MIBs, explicitly.
- SMIv2 has limited support for semantic constraints in the data model. This means that it is difficult to know if a certain configuration will apply cleanly on a device. If it doesn't, rollback is tricky, as explained above.
- Because of all of the above, ordering of SET requests becomes very important. If a device refuses to create some object A before another B, an SNMP manager must make sure to create B before creating A. It is also common that objects cannot be modified without first making them disabled or inactive. There is no standard way to do this, so again, different data models do this in different ways.

Despite all this, if a device can be configured over SNMP, NSO can use its built-in multilingual SNMP manager to communicate with the device. However, in order to solve the problems mentioned above, the MIBs supported by the device need to be carefully annotated with some additional information that instruct NSO on how to write configuration data to the device. This additional information is described in detail below.

Overview

To add a device, the following steps need to be followed. They are described in more details in the following sections.

- Collect (a subset of) the MIBs supported by the device.
- Optionally annotate the MIBs with annotations to instruct NSO on how to talk to the device, for example ordering dependencies that are not explicitly modeled in the MIB. This step is not required.

- Compile the MIBs and load them into NSO.
- Configure NSO with the address and authentication parameter for the SNMP devices.
- Optionally configure a named MIB group in NSO with the MIBs supported by the device, and configure the managed device in NSO to use this MIB group. If this step is not done, NSO assumes the device implements all MIBs known to NSO.

Compiling and loading MIBs

(See the Makefile `snmp-ned/basic/packages/ex-snmp-ned/src/Makefile`, for an example of the below description.) Make sure that you have all MIBs available, including import dependencies and that they contain no errors.

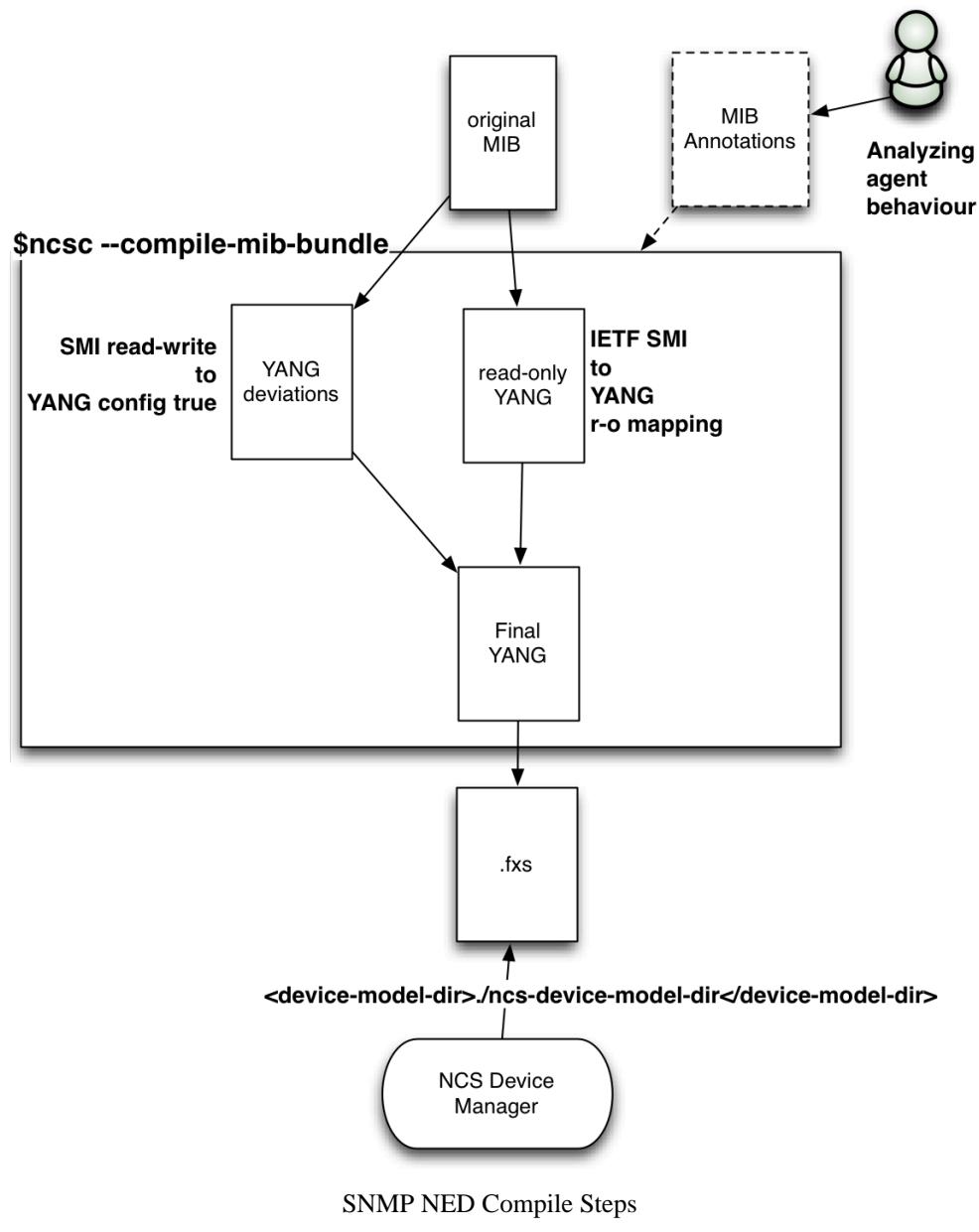
The **ncsc --ncs-compile-mib-bundle** compiler is used to compile MIBs and MIB annotation files into NSO load files. Assuming a directory with input MIB files (and optional MIB annotation files) exist, the following command compiles all the MIBs in `device-models` and writes the output to `ncs-device-model-dir`.

```
$ ncsc --ncs-compile-mib-bundle device-models \
--ncs-device-dir ./ncs-device-model-dir
```

The compilation steps performed by the **ncsc --ncs-compile-mib-bundle** are elaborated below::

- 1 Transform the MIBs into YANG according to the IETF standardized mapping (<https://www.ietf.org/rfc/rfc6643.txt>). The IETF defined mapping makes all MIB objects read-only over NETCONF.
- 2 Generate YANG deviations from the MIB, this basically makes SMIv2 `read-write` objects YANG `config` true as a YANG deviation.
- 3 Include the optional MIB annotations.
- 4 Merge the read-only YANG from step 1 with the read-write deviation from step 2.
- 5 Compile the merged YANG files into NSO load format.

These steps are illustrated in the Figure below:



Finally make sure that the NSO configuration file points to the correct device model directory:

```
<device-model-dir>./ncs-device-model-dir</device-model-dir>
```

Configuring NSO to speak SNMP southbound

Each managed device is configured with a name, IP address and port (161 by default), and the SNMP version to use (v1, v2c, or v3).

```
admin@host# show running-config devices device r3
```

```
address 127.0.0.1
port    2503
device-type snmp version v3 snmp-authgroup my-authgroup
state admin-state unlocked
```

In order to minimize the necessary configuration, the authentication group concept (see the section called "Authentication Groups" in *User Guide*) is used also for SNMP. A configured managed device of type `snmp` refers to an SNMP authgroup. An SNMP authgroup contains community strings for SNMP v1 and v2c, and USM parameters for SNMP v3.

```
admin@host# show running-config devices authgroups snmp-group my-authgroup
devices authgroups snmp-group my-authgroup
default-map community-name public
umap admin
usm remote-name admin
usm security-level auth-priv
usm auth md5 remote-password $4$wIo7Yd068FRwhYYI0d4IDw==
usm priv des remote-password $4$wIo7Yd068FRwhYYI0d4IDw==
!
!
```

In the example above, when NSO needs to speak to the device "r3", it sees that the device is of type `snmp`, and that SNMP v3 should be used, with authentication parameters from the SNMP authgroup "my-authgroup". This authgroup maps the local NSO user "admin" to the USM user "admin", with explicit remote passwords given. These passwords will be localized for each SNMP engine that NSO communicates with. While the passwords above are shown encrypted, when you *enter* them in the CLI you write them in clear-text. Note also that the remote engine id is not configured; NSO performs a discovery process to find it automatically.

No NSO user other than "admin" is mapped by the authgroup "my-authgroup" for SNMP v3.

Configure MIB groups

With SNMP, there is no standardized, generic way for an SNMP manager to learn which MIBs an SNMP agent implements. By default, NSO assumes that an SNMP device implements all MIBs known to NSO, i.e., all MIBs that have been compiled with the `ncsc --ncs-compile-mib-bundle` command. This works just fine if all SNMP devices NSO manages are of the same type, and implement the same set of MIBs. But if NSO is configured to manage many different SNMP devices, some other mechanism is needed.

In NSO, this problem is solved by using *MIB groups*. A MIB group is a named collection of MIB module names. A managed SNMP device can refer to one or more MIB groups. For example, below two MIB groups are defined:

```
admin@ncs# show running-config devices mib-group
devices mib-group basic
mib-module [ BASIC-CONFIG-MIB BASIC-TC ]
!
devices mib-group snmp
mib-module [ SNMP* ]
!
```

The wildcard '*' can be used only at the end of a string; it is thus used to define a prefix of a MIB module name. So the string "SNMP*" matches all loaded standard SNMP modules, such as SNMPv2-MIB, SNMP-TARGET-MIB etc.

An SNMP device can then be configured to refer to one or more of the MIB groups:

```
admin@ncs# show running-config devices device r3 device-type snmp
devices device r3
device-type snmp version v3
```

```
device-type snmp snmp-authgroup default
device-type snmp mib-group [ basic snmp ]
!
```

Annotations for MIB objects

Most annotations for MIB objects are used to instruct NSO on how to split a large transaction into suitable SNMP SET requests. This step is not necessary for a default integration. But when for example ordering dependencies in the MIB is discovered it is better to add this as annotations and let NSO handle the ordering rather than leaving it to the CLI user or Java programmer.

In some cases, NSO can automatically understand when rows in a table must be created or deleted before rows in some other table. Specifically, NSO understands that if a table B has an INDEX object in table A (i.e., B sparsely augments A), then rows in table B must be created after rows in table A, and vice versa for deletions. NSO also understand that if table B AUGMENTS table A, then a row in table A must be created before any column in B is modified.

However, in some MIBs, table dependencies cannot be detected automatically. In this case, these tables must be annotated with a `sort-priority`. By default, all rows have sort-priority 0. If table A has a lower sort-priority than table B, then rows in table A are created before rows in table B.

In some tables, existing rows cannot be modified unless the row is inactivated. Once inactive, the row can be modified, and then activated again. Unfortunately, there is no formal way to declare this is SMIv2, so these tables must be annotated with two statements; `ned-set-before-row-modification` and `ned-modification-dependent`. The former is used to instruct NSO which column and which value is used to deactivate a row, and the latter is used on each column that requires the row to be deactivated before modification. `ned-modification-dependent` can be used in the same table as `ned-set-before-row-modification`, or in a table that augments or sparsely augments the table with `ned-set-before-row-modification`.

By default, NSO treats a writable SMIv2 object as configuration, except if the object is of type RowStatus. Any writable object that does not represent configuration must be listed in a MIB annotation file when the MIB is compiled, with the "operational" modifier.

When NSO retrieves data from an SNMP device, e.g., when doing a `sync from-device`, it uses the GET-NEXT request to scan the table for available rows. When doing the GET-NEXT, NSO must ask for an accessible column. If the row has a column of type RowStatus, NSO uses this column. Otherwise, if the one of the INDEX objects are accessible, it uses this object. Otherwise, if table has been annotated with `ned-accessible-column`, this column is used. And, as a last resort, NSO does not indicate any column in the first GET-NEXT request, and uses the column returned from the device in subsequent requests. If the table has "holes" for this column, i.e., the column is not instantiated in all rows, NSO will not detect those rows.

NSO can automatically create and delete table rows for tables that use the RowStatus TEXTUAL-CONVENTION, defined in RFC 2580.

It is pretty common to mix configuration objects with non-configuration objects in MIBs. Specifically, it is quite common that rows are created automatically by the device, but then some columns in the row are treated as configuration data. In this case, the application programmer must tell NSO to sync from the device before attempting to modify the configuration columns, in order to let NSO learn which rows exist on the device.

Some SNMP agents require a certain order of row deletions and creations. By default, the SNMP NED send all creates before deletes. The annotation `ned-delete-before-create` can be used on a table entry in order to send row deletions before row creations, for that table.

Sometimes rows in some SNMP agents cannot be modified once created. Such rows can be marked with the annotation `ned-recreate-when-modified`. This makes the SNMP NED to first delete the row, and then immediately recreate it with the new values.

A good starting point for understanding annotations is to look at the example in `examples.ncs/snmp-ned` directory. The `BASIC-CONFIG-MIB` mib has a table where rows can be modified if the `bscActAdminState` is set to locked. In order to have NSO do this automatically when modifying entries rather than leaving it to users an annotation file can be created. See the `BASIC-CONFIG-MIB.miba` which contains the following:

```
## NCS Annotation module for BASIC-CONFIG-MIB

bscActAdminState  ned-set-before-row-modification = locked
bscActFlow        ned-modification-dependent
```

This tells NSO that before modifying the `bscActFlow` column set the `bscActAdminState` to locked and restore the previous value after committing the set operation.

All MIB annotations for a particular MIB are written to a file with the file suffix `.miba`. See `mib_annotations(5)` in *Manual Pages* for details.

Make sure that the MIB annotation file is put into the directory where all the MIB files are which is given as input to the `ncsc --ncs-compile-mib-bundle` command

Using the SNMP NED

NSO can manage SNMP devices within transactions, a transaction can span Cisco devices, NETCONF devices and SNMP devices. If a transaction fails NSO will generate the reverse operation to the SNMP device.

The basic features of the SNMP will be illustrated below by using the `examples.ncs/snmp-ned` example. First try to connect to all SNMP devices:

```
admin@ncs# devices connect

connect-result {
    device r1
    result true
    info (admin) Connected to r1 - 127.0.0.1:2501
}
connect-result {
    device r2
    result true
    info (admin) Connected to r2 - 127.0.0.1:2502
}
connect-result {
    device r3
    result true
    info (admin) Connected to r3 - 127.0.0.1:2503
}
```

When NSO executes the connect request for SNMP devices it performs a get-next request with 1.1 as var-bind. When working with the SNMP NED it is helpful to turn on the NED tracing:

```
$ ncsc -C -u admin

admin@ncs config

admin@ncs(config)# devices global-settings trace pretty trace-dir .
```

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

Commit complete.

This creates a trace-file named ned-devicename.trace. The trace for the ncs connect action looks like:

```
$ more ned-r1.trace
get-next-request reqid=2
  1.1
get-response reqid=2
  1.3.6.1.2.1.1.1.0=Tail-f ConfD agent - 1
```

When looking at SNMP trace files it is useful to have the OBJECT-DESCRIPTOR rather than the OBJECT-IDENTIFIER. To do this, pipe the trace file to the **smixlate** tool:

```
$ more ned-r1.trace | smixlate $NCS_DIR/src/ncs/snmp/mibs/SNMPv2-MIB.mib
get-next-request reqid=2
  1.1
get-response reqid=2
  sysDescr.0=Tail-f ConfD agent - 1
```

You can access the data in the SNMP systems directly (read-only and read-write objects):

```
admin@ncs# show devices device live-status

ncs live-device r1
live-status SNMPv2-MIB system sysDescr "Tail-f ConfD agent - 1"
live-status SNMPv2-MIB system sysObjectID 1.3.6.1.4.1.24961
live-status SNMPv2-MIB system sysUpTime 596197
live-status SNMPv2-MIB system sysContact ""
live-status SNMPv2-MIB system sysName ""
...
...
```

NSO can synchronize all writable objects into CDB:

```
admin@ncs# devices sync-from
sync-result {
  device r1
  result true
}

admin@ncs# show running-config devices device r1 config r:SNMPv2-MIB

devices device r1
config
  system
    sysContact  ""
    sysName     ""
    sysLocation ""
!
snmp
  snmpEnableAuthenTraps disabled;
!
```

All the standard features of NSO with transactions and roll-backs will work with SNMP devices. The sequence below shows how to enable authentication traps for all devices as one transaction. If any device fails, NSO will automatically rollback the others. At the end of the CLI sequence a manual rollback is shown:

```
admin@ncs# config
admin@ncs(config)# devices device r1-3 config r:SNMPv2-MIB snmp snmpEnableAuthenTraps enabled
```

```

admin@ncs(config)# commit
Commit complete.

admin@ncs(config)# top rollback configuration
admin@ncs(config)# commit dry-run outformat cli

cli devices {
    device r1 {
        config {
            r:SNMPv2-MIB {
                snmp {
                    - snmpEnableAuthenTraps enabled;
                    + snmpEnableAuthenTraps disabled;
                }
            }
        }
    }
    device r2 {
        config {
            r:SNMPv2-MIB {
                snmp {
                    - snmpEnableAuthenTraps enabled;
                    + snmpEnableAuthenTraps disabled;
                }
            }
        }
    }
    device r3 {
        config {
            r:SNMPv2-MIB {
                snmp {
                    - snmpEnableAuthenTraps enabled;
                    + snmpEnableAuthenTraps disabled;
                }
            }
        }
    }
}
admin@ncs(config)# commit
Commit complete.

```

Statistics

NED devices have runtime data, statistics. The first part in being able to collect non-configuration data from a NED device is to model the statistics data we wish to gather. In normal YANG files, it is common to have the runtime data nested inside the configuration data. In gathering runtime data for NED devices we have chosen to separate configuration data and runtime data. In the case of the archetypical CLI device, the `show running-config ...` and friends are used to display the running configuration of the device whereas other different `show ...` commands are used to display runtime data, for example `show interfaces`, `show routes`. Different commands for different types of routers/switches and in particular, different tabular output format for different device types.

To expose runtime data from a NED controlled device, regardless of whether it's a CLI NED or a Generic NED, we need to do two things:

- Write YANG models for the aspects of runtime data we wish to expose northbound in NSO.

- Write Java NED code that is responsible for collecting that data.

The NSO NED for the Avaya 4k device contains a data model for some real statistics for the Avaya router and also the accompanying Java NED code. Let's start to take a look at the YANG model for the stats portion, we have:

Example 185. NED stats YANG model

```
module tailf-ned-avaya-4k-stats {
    namespace 'http://tail-f.com/ned/avaya-4k-stats';
    prefix avaya4k-stats;

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

    import ietf-yang-types {
        prefix yang;
    }

    container stats {
        config false;
        container interface {
            list gigabitEthernet {
                key "num port";
                tailf:cli-key-format "$1/$2";

                leaf num {
                    type uint16;
                }

                leaf port {
                    type uint16;
                }

                leaf in-packets-per-second {
                    type uint64;
                }

                leaf out-packets-per-second {
                    type uint64;
                }

                leaf in-octets-per-second {
                    type uint64;
                }

                leaf out-octets-per-second {
                    type uint64;
                }

                leaf in-octets {
                    type uint64;
                }

                leaf out-octets {
                    type uint64;
                }
            }
        }
    }
}
```

```
    leaf in-packets {
        type uint64;
    }

    leaf out-packets {
        type uint64;
    }
}
```

It's a `config false`; list of counters per interface. We compile the NED stats module with the `--ncs-compile-module` flag or with the `--ncs-compile-bundle` flag. It's the same *non-config* module that contains both runtime data as well as commands and rpcs.

```
$ ncsc --ncs-compile-module avaya4k-stats.yang \
      --ncs-device-dir <dir>
```

The **config false**; data from a module that has been compiled with the --ncs-compile-module flag will end up mounted under /devices/device/live-status tree. Thus running the NED towards a real router we have:

Example 186. Displaying NED stats in the CLI

```
admin@ncs# show devices device r1 live-status interfaces

live-status {
    interface gigabitEthernet1/1 {
        in-packets-per-second    234;
        out-packets-per-second   177;
        in-octets-per-second     4567;
        out-octets-per-second    3561;
        in-octets                 12666;
        out-octets                16888;
        in-packets                7892;
        out-packets               2892;
    }
    ....
```

It is the responsibility of the NED code to populate the data in the live device tree. Whenever a northbound agent tries to read any data in the live device tree for a NED device, the NED code is invoked.

The NED code implements an interface called, NedConnection. This interface contains:

```
void showStatsPath(NedWorker w, int th, ConfPath path)
    throws NedException, IOException;
```

This interface method is invoked by NSO in the NED. The Java code must return what is requested, but it may also return more. The Java code always needs to signal errors by invoking `NedWorker.error()` and success by invoking `NedWorker.showStatsPathResponse()`. The latter function indicates what is returned, and also how long it shall be cached inside NSO.

The reason for this design, is that it is common for many show commands to work on for example an entire interface, or some other item in the managed device. Say that the NSO operator (or maapi code) invokes:

```
admin@host> show status devices device r1 live-status \
           interface gigabitEthernet1/1/1 out-octets
out-octets 340;
```

requesting a single leaf, the NED Java code can decide to execute any arbitrary show command towards the managed device, parse the output and populate as much data as it wants. The Java code also decides how long time NSO shall cache the data.

- When the `showStatsPath()` is invoked, the NED should indicate the state/value of the node indicated by the path (i.e. if a leaf was requested, the NED should write the value of this leaf to the provided transaction handler (th) using MAAP, or indicate its absence as described below; if a list entry or a presence container was requested then the NED should indicate presence or absence of the element, if the whole list is requested then the NED should populate the keys for this list). Often requesting such data from the actual device will give the NED more data than specifically requested, in which case the worker is free to write other values as well. The NED is not limited to populating the subtree indicated by the path, it may also write values outside this subtree. NSO will then not request those paths but read them directly from the transaction. Different timeouts can be provided for different paths.

If a leaf does not have a value, or does not exist, the NED can indicate this by returning a TTL for the path to the leaf, without setting the value in the provided transaction. This has changed from earlier versions of NSO. The same applies for optional containers and list entries. If the NED populates the keys for a certain list (both when it is requested to do so or when it decided to do so because it has received this data from the device), it should set the TTL value for the list itself to indicate the time the set of keys should be considered up to date. It may choose to provide different TTL values for some or all list entries, but it is not required to do so.

Making the NED handle default values properly

One important task when implementing a NED of any type is to make it mimic the devices handling of default values as close as possible. Network equipment can typically deal with default values in many different ways.

Some devices display default values on leafs even if they have not been explicitly set. Others use trimming, meaning that if a leaf is set to its default value it will be 'unset' and disappear from the devices configuration dump.

It is the responsibility of the NED to make the NSO aware of how the device handles default values. This is done by registering a special NED Capability entry with the NSO. Two modes are currently supported by the NSO: "trim" and "report-all".

Example 187. A device trimming default values

This is the typical behavior of a Cisco IOS device. The simple YANG code snippet below illustrates the behavior. A container with a boolean leaf. Its default value is true.

```
container aaa {
    leaf enabled {
        default true;
        type boolean;
    }
}
```

Try setting the leaf to true in NSO and commit. Then compare the configuration:

```
$ ncs_cli -C -u admin
admin@ncs# config
admin@ncs(config)# devices device a0 config aaa enabled true
admin@ncs(config)# commit
```

```

Commit complete.

admin@ncs(config)# top devices device a0 compare-config

diff
devices {
    device a0 {
        config {
            aaa {
-                enabled;
            }
        }
    }
}

```

The result shows that the configurations differ. The reason is that the device does not display the value of the leaf 'enabled'. It has been trimmed since it has its default value. The NSO is now out of sync with the device.

To solve this issue, make the NED tell the NSO that the device is trimming default values. Register an extra NED Capability entry in the Java code.

```

NedCapability capas[] = new NedCapability[2];
capas[0] = new NedCapability(
    "",
    "urn:ios",
    "tailf-ned-cisco-ios",
    "",
    "2015-01-01",
    "");
capas[1] = new NedCapability(
    "urn:ietf:params:netconf:capability:" +
    "with-defaults:1.0?basic-mode=trim", // Set mode to trim
    "urn:ietf:params:netconf:capability:" +
    "with-defaults:1.0",
    "",
    "",
    "",
    "");

```

Now, try the same operation again:

```

$ ncs_cli -C -u admin

admin@ncs# config

admin@ncs(config)# devices device a0 config aaa enabled true
admin@ncs(config)# commit

Commit complete.

admin@ncs(config)# top devices device a0 compare-config
admin@ncs(config)#

```

The NSO is now in sync with the device.

Example 188. A device displaying all default values

Some devices display default values for leafs even if they have not been explicitly set. The simple YANG code below will be used to illustrate this behavior. A list containing a key and a leaf with a default value.

```
list interface {
```

Making the NED handle default values properly

```

key id;
leaf id {
    type string;
}
leaf threshold {
    default 20;
    type uint8;
}
}

```

Try creating a new list entry in NSO and commit. Then compare the configuration:

```

$ ncs_cli -C -u admin
admin@ncs# config
admin@ncs(config)# devices device a0 config interface myinterface
admin@ncs(config)# commit
admin@ncs(config)# top devices device a0 compare-config

diff
devices {
    device a0 {
        config {
            interface myinterface {
+                threshold 20;
            }
        }
    }
}

```

The result shows that the configurations differ. The NSO is out of sync. This is because the device displays the default value of the 'threshold' leaf even if it has not been explicitly set through the NSO.

To solve this issue, make the NED tell the NSO that the device is reporting all default values. Register an extra NED Capability entry in the Java code.

```

NedCapability capas[] = new NedCapability[2];
capas[0] = new NedCapability(
    "",
    "urn:abc",
    "tailf-ned-abc",
    "",
    "2015-01-01",
    "");
capas[1] = new NedCapability(
    "urn:ietf:params:netconf:capability:" +
    "with-defaults:1.0?basic-mode=report-all", // Set mode to report-all
    "urn:ietf:params:netconf:capability:" +
    "with-defaults:1.0",
    "",
    "",
    "",
    "");

```

Now, try the same operation again:

```

$ ncs_cli -C -u admin
admin@ncs# config
admin@ncs(config)# devices device a0 config interface myinterface

```

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

admin@ncs(config)# top devices device a0 compare-config
admin@ncs(config)#

```

The NSO is now in sync with the device.

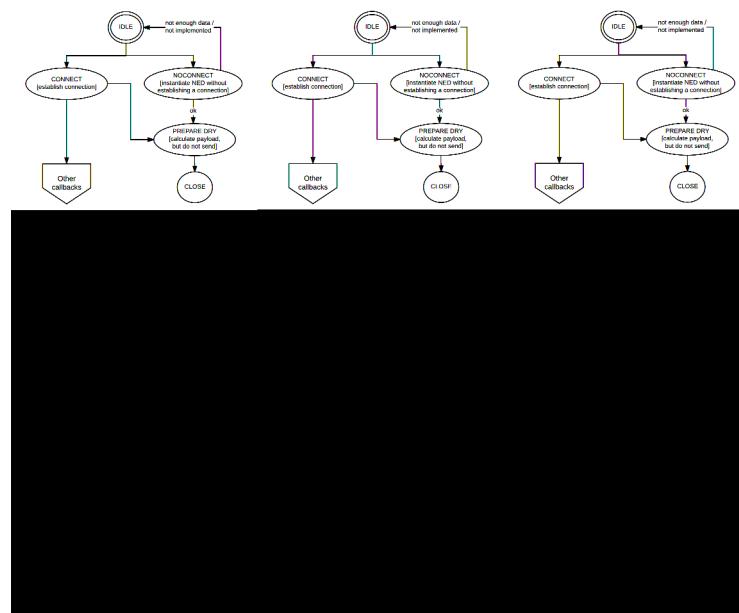
Dry-run considerations

The possibility to do a dry-run on a transaction is a feature in NSO that allows to examine the changes to be pushed out to the managed devices in the network. The output can be produced in different formats, namely cli, xml and native. In order to produce dry-run in the native output format NSO needs to know the exact syntax used by the device, and the task of converting the commands or operations produced by the NSO into the device-specific output belongs the corresponding NED. This is the purpose of the `prepareDry()` callback in the NED interface.

In order to be able to invoke a callback an instance of the NED object needs to be created first. There are two ways to instantiate a NED:

- `newConnection()` callback that tells the NED to establish connection to the device which can later be used to perform any action such as show configuration, apply changes or view operational data as well as produce dry-run output.
- Optional `initNoConnect()` callback that tells the NED to create an instance that would not need to communicate with the device, and hence must not establish a connection or otherwise communicate with the device. This instance will only be used to calculate dry-run output. It is possible for a NED to reject the `initNoConnect()` request if it is not able to calculate the dry-run output without establishing a connection to the device, for example if a NED is capable of managing devices with different flavors of syntax and it is not known at the moment which syntax is used by this particular device.

The following state diagram displays NED states specific to dry-run scenario.



NED dry-run states

NED identification

Each managed device in NSO has a device type, which informs NSO how to communicate with the device. The device type is one of *netconf*, *snmp*, *cli*, or *generic*. In addition, a special *ned-id* identifier is needed.

NSO uses a technique called *YANG Schema Mount*, where all the data models from a device are mounted into the `/devices` tree in NSO. Each set of mounted data models is completely separated from the others (they are confined to a "mount jail"). This makes it possible to load different versions of the same YANG module for different devices. The functionality is called Common Data Models (CDM).

In most cases, there are many devices running the same software version in the network managed by NSO, thus using the exact same set of YANG modules. With CDM, all YANG modules for a certain device (or family of devices) are contained in a *NED package* (or just *NED* for short). If the YANG modules on the device are updated in a backwards compatible way, the NED is also updated.

However, if the YANG modules on the device are updated in an incompatible way in a new version of the device's software, it might be necessary to create a new NED package for the new set of modules. Without CDM, this would not be possible, since there would be two different packages that contained different versions of the same YANG module.

When a NED is being built, its YANG modules are compiled to be mounted into the NSO YANG model. This is done by *device compilation* of the device's YANG modules, and is performed via the `ncsc` tool provided by NSO.

The *ned-id* identifier is a YANG identity, which must be derived from one of the pre-defined identities in `$NCS_DIR/src/ned/yang/tailf-ncs-ned.yang`.

A YANG model for devices handled by NED code needs to extend the base identity and provide a new identity that can be configured.

Example 189. Defining a user identity

```
import tailf-ncs-ned {
    prefix ned;
}

identity cisco-ios {
    base ned:cli-ned-id;
}
```

The Java NED code registers the identity it handles with NSO.

Similar to how we import device models for NETCONF based devices, we use the `ncsc --ncs-compile-bundle` command to import YANG models for NED handled devices.

Once we have imported such a YANG model into NSO, we can configure managed device in NSO to be handled by the appropriate NED handler (which is user Java code, more on that later)

Example 190. Setting the device type

```
admin@ncs# show running config devices device r1

address      127.0.0.1
port         2025
authgroup   default
device-type cli ned-id cisco-ios
state admin-state unlocked
...
```

When NSO needs to communicate southbound towards a managed device which is not of type NETCONF, it will look for a NED that has registered with the name of the identity, in the case above, the string "ios".

Thus before NSO attempts to connect to a NED device, before it tries to sync, or manipulate the configuration of the device, a user based Java NED code must have registered with the NSO service manager indicating which Java class is responsible for the NED with the string of the identity, in this case the string "ios". This happens automatically when the NSO java VM gets a `instantiate-component` request for a NSO package component of type ned.

The component java class `myNed` needs to implement either of the interfaces `NedGeneric` or `NedCli`. Both interfaces require the NED class to implement the following:

Example 191. NED identification callbacks

```
// should return "cli" or "generic"
String type();

// Which YANG modules are covered by the class
String [] modules();

// Which identity is implemented by the class
String identity();
```

The above three callbacks are used by the NSO Java VM to connect the NED Java class with NSO. They are called at when the NSO Java VM receives the `instantiate-component` request.

The underlying NedMux will start a number of threads, and invoke the registered class with other data callbacks as transactions execute.

Migrating to the juniper-junos_nc-gen NED

NSO has supported Junos devices from early on. The legacy Junos NED is NETCONF-based, but as Junos devices did not provide YANG modules in the past, complex NSO machinery translated Juniper's XML Schema Description (XSD) files into a single YANG module. This was an attempt to aggregate several Juniper device modules/versions.

Juniper nowadays provides YANG modules for Junos devices. Junos YANG modules can be downloaded from the device and used directly in NSO with the new `juniper-junos_nc-gen` NED.

By downloading the YANG modules using `juniper-junos_nc-gen` NED tools and rebuilding the NED, the NED can provide full coverage immediately when the device is updated instead of waiting for a new legacy NED release.

This guide describes how to replace the legacy `juniper-junos` NED and migrate NSO applications to the `juniper-junos_nc-gen` NED using the NSO MPLS VPN example from the NSO examples collection as a reference.

Prepare the example:

- 1 Add the `juniper-junos` and `juniper-junos_nc-gen` NED packages to the example.
- 2 Configure the connection to the Junos device.
- 3 Add the MPLS VPN service configuration to the simulated network, including the Junos device using the legacy `juniper-junos` NED.

Adapting the service to the `juniper-junos_nc-gen` NED:

- 1 Un-deploy MPLS VPN service instances with **no-networking**.
- 2 Delete Junos device config with **no-networking**.

- 3 Set the Junos device to NETCONF/YANG compliant mode.
- 4 Switch the ned-id for the Junos device to the juniper-junos_nc-gen NED package.
- 5 Download the compliant YANG models, build, and reload the juniper-junos_nc-gen NED package.
- 6 Sync from the Junos device to get the compliant Junos device config.
- 7 Update the MPLS VPN service to handle the difference between the non-compliant and compliant configurations belonging to the service.
- 8 Re-deploy the MPLS VPN service instances with **no-networking** to make the MPLS VPN service instances own the device configuration again.

**Note**

If applying the steps for this example on a production system, you should first take a backup using the ncs-backup tool before proceeding.

Prepare the Example

This guide uses the MPLS VPN example in Python from the NSO example set under \$NCS_DIR/examples.ncs/getting-started/developing-with-ncs/17-mpls-vpn-python to demonstrate porting an existing application to use the juniper-junos_nc-gen NED. The simulated Junos device is replaced with a Junos vMX 21.1R1.11 container, but other NETCONF/YANG-compliant Junos versions also work.

Add the juniper-junos and juniper-junos_nc-gen NED Packages

The first step is to add the latest juniper-junos and juniper-junos_nc-gen NED packages to the example's package directory. The NED tar-balls must be available and downloaded from your <https://software.cisco.com/download/home> account to the 17-mpls-vpn-python example directory. Replace the NSO_VERSION and NED_VERSION variables with the versions you use:

```
$ cd $NCS_DIR/examples.ncs/getting-started/developing-with-ncs/17-mpls-vpn-python
$ cp ./ncs-NSO_VERSION-juniper-junos-NED_VERSION.tar.gz packages/
$ cd packages
$ tar xfz ../ncs-NSO_VERSION-juniper-junos_nc-NED_VERSION.tar.gz
$ cd -
```

Build and start the example:

```
$ make all start
```

Configure the Connection to the Junos Device

Replace the netsim device connection configuration in NSO with the configuration for connecting to the Junos device. Adjust the USER_NAME, PASSWORD, and HOST_NAME/IP_ADDR variables and the timeouts as required for the Junos device you are using with this example:

```
$ ncs_cli -u admin -C
admin@ncs# config
admin@ncs(config)# devices authgroups group juniper umap admin remote-name USER_NAME \
remote-password PASSWORD
admin@ncs(config)# devices device pe2 authgroup juniper address HOST_NAME/IP_ADDR port 830
admin@ncs(config)# devices device pe2 connect-timeout 240
admin@ncs(config)# devices device pe2 read-timeout 240
admin@ncs(config)# devices device pe2 write-timeout 240
admin@ncs(config)# commit
admin@ncs(config)# end
admin@ncs# exit
```

Open a CLI terminal or use NETCONF on the Junos device to verify that the rfc-compliant and yang-compliant modes are not yet enabled. Examples:

```
$ ssh USER_NAME@HOST_NAME/IP_ADDR
junos> configure
junos# show system services netconf
ssh;
```

Or:

```
$ netconf-console -s plain -u USER_NAME -p PASSWORD --host=HOST_NAME/IP_ADDR \
--port=830 --get-config
--subtree-filter=-<<<'<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm">
    <system>
        <services>
            <netconf/>
        </services>
    </system>
</configuration>'>
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/21.1R0/junos"
           xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
<data>
    <configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm">
        <system>
            <services>
                <netconf>
                    <ssh>
                        </ssh>
                </netconf>
            </services>
        </system>
    </configuration>
</data>
</rpc-reply>
```

The rfc-compliant and yang-compliant nodes must not be enabled yet for the legacy Junos NED to work. If enabled, delete in the Junos CLI or using NETCONF. A netconf-console example:

```
$ netconf-console -s plain -u USER_NAME -p PASSWORD --host=HOST_NAME/IP_ADDR --port=830
--db=candidate
--edit-config=- <<<'<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm"
           xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
    <system>
        <services>
            <netconf>
                <rfc-compliant nc:operation="remove"/>
                <yang-compliant nc:operation="remove"/>
            </netconf>
        </services>
    </system>
</configuration>'>
$ netconf-console -s plain -u USER_NAME -p PASSWORD --host=HOST_NAME/IP_ADDR \
--port=830 --commit
```

Back to the NSO CLI to upgrade the legacy juniper-junos NED to the latest version:

```
$ ncs_cli -u admin -C
admin@ncs# config
admin@ncs(config)# devices device pe2 ssh fetch-host-keys
admin@ncs(config)# devices device pe2 migrate new-ned-id juniper-junos-nc-NED_VERSION
admin@ncs(config)# devices sync-from
```

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

Add the MPLS VPN Service Configuration to the Simulated Network

Turn off autowizard and complete-on-space to make it possible to paste configs:

```
admin@ncs# autowizard false
admin@ncs# complete-on-space false
```

The example service config for two MPLS VPNs where the endpoints have been selected to pass through the PE node PE2, which is a Junos device:

```
vpn 13vpn ike
as-number 65101
endpoint branch-office1
    ce-device    ce1
    ce-interface GigabitEthernet0/11
    ip-network   10.7.7.0/24
    bandwidth    6000000
!
endpoint branch-office2
    ce-device    ce4
    ce-interface GigabitEthernet0/18
    ip-network   10.8.8.0/24
    bandwidth    300000
!
endpoint main-office
    ce-device    ce0
    ce-interface GigabitEthernet0/11
    ip-network   10.10.1.0/24
    bandwidth    12000000
!
qos qos-policy GOLD
!
vpn 13vpn spotify
as-number 65202
endpoint branch-office1
    ce-device    ce5
    ce-interface GigabitEthernet0/1
    ip-network   10.2.3.0/24
    bandwidth    10000000
!
endpoint branch-office2
    ce-device    ce3
    ce-interface GigabitEthernet0/4
    ip-network   10.4.5.0/24
    bandwidth    20000000
!
endpoint main-office
    ce-device    ce2
    ce-interface GigabitEthernet0/8
    ip-network   10.0.1.0/24
    bandwidth    40000000
!
qos qos-policy GOLD
!
```

To verify that the traffic passes through PE2:

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

Toward the end of this lengthy output, observe that some config changes are going to the PE2 device using the <http://xml.juniper.net/xnm/1.1/xnm> legacy namespace:

```

device {
    name pe2
    data <rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
        <edit-config xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
            <target>
                <candidate/>
            </target>
            <test-option>test-then-set</test-option>
            <error-option>rollback-on-error</error-option>
            <with-inactive xmlns="http://tail-f.com/ns/netconf/inactive/1.0"/>
            <config>
                <configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm">
                    <interfaces>
                        <interface>
                            <name>xe-0/0/2</name>
                            <unit>
                                <name>102</name>
                                <description>Link to CE / ce5 - GigabitEthernet0/1</description>
                                <family>
                                    <inet>
                                        <address>
                                            <name>192.168.1.22/30</name>
                                        </address>
                                    </inet>
                                </family>
                                <vlan-id>102</vlan-id>
                            </unit>
                        </interface>
                    </interfaces>
                ...
            </config>
        </edit-config>
    </rpc>
}

```

Looks good. Commit to the network:

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

Adapting the Service to the juniper-junos_nc-gen NED

Now that the service's configuration is in place using the legacy juniper-junos NED to configure the PE2 Junos device, proceed and switch to using the juniper-junos_nc-gen NED with PE2 instead. The service template and Python code will need a few adaptations.

Un-deploy MPLS VPN Services Instances with no-networking

To keep the NSO service meta-data information intact when bringing up the service with the new juniper-junos_nc-gen NED, first **un-deploy** the service instances in NSO, only keeping the configuration on the devices:

```
admin@ncs(config)# vpn 13vpn * un-deploy no-networking
```

Delete Junos Device Config with no-networking

First, save the legacy Junos non-compliant mode device configuration to later diff against the compliant mode config:

```
admin@ncs(config)# show full-configuration devices device pe2 config \
    configuration | display xml | save legacy.xml
```

Delete the PE2 configuration in NSO to prepare for retrieving it from the device in a NETCONF/YANG compliant format using the new NED:

```
admin@ncs(config)# no devices device pe2 config
```

```
admin@ncs(config)# commit no-networking
admin@ncs(config)# end
admin@ncs# exit
```

Set the Junos Device to NETCONF/YANG Compliant Mode

Using the Junos CLI:

```
$ ssh USER_NAME@HOST_NAME/IP_ADDR
junos> configure
junos# set system services netconf rfc-compliant
junos# set system services netconf yang-compliant
junos# show system services netconf
ssh;
rfc-compliant;
yang-compliant;
junos# commit
```

Or using the NSO **netconf-console** tool:

```
$ netconf-console -s plain -u USER_NAME -p PASSWORD --host=HOST_NAME/IP_ADDR --port=830 \
--db=candidate
--edit-config=- <<<'<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm">
<system>
    <services>
        <netconf>
            <rfc-compliant/>
            <yang-compliant/>
        </netconf>
    </services>
</system>
</configuration>'
```



```
$ netconf-console -s plain -u USER_NAME -p PASSWORD --host=HOST_NAME/IP_ADDR --port=830 \
--commit
```

Switch the NED ID for the Junos Device to the juniper-junos_nc-gen NED Package

```
$ ncs_cli -u admin -C
admin@ncs# config
admin@ncs(config)# devices device pe2 device-type generic ned-id juniper-junos_nc-gen-1.0
admin@ncs(config)# commit
admin@ncs(config)# end
```

Download the Compliant YANG models, Build, and Load the juniper-junos_nc-gen NED Package

The juniper-junos_nc-gen NED is delivered without YANG modules, enabling populating it with device-specific YANG modules. The YANG modules are retrieved directly from the Junos device:

```
$ ncs_cli -u admin -C
admin@ncs# devices device pe2 connect
admin@ncs# devices device pe2 rpc rpc-get-modules get-modules
admin@ncs# exit
```

See the juniper-junos_nc-gen README for more options and details.

Build the YANG modules retrieved from the Junos device with the juniper-junos_nc-gen NED:

```
$ make -C packages/juniper-junos_nc-gen-1.0/src
```

Reload the packages to load the juniper-junos_nc-gen NED with the added YANG modules:

```
$ ncs_cli -u admin -C
admin@ncs# packages reload
```

Sync From the Junos Device to get the Device Configuration in NETCONF/YANG Compliant Format

```
admin@ncs# devices device pe2 sync-from
```

Update the MPLS VPN Service

The service must be updated to handle the difference between the Junos device's non-compliant and compliant configuration. The NSO service uses Python code to configure the Junos device using a service template. One way to find the required updates to the template and code is to check the difference between the non-compliant and compliant configurations for the parts covered by the template.

show running-config	original l3vpn-pe.xml
<pre><configuration xmlns="..."> <interfaces xmlns="..."> <interfaces> <interface> <name>xe-0/0/2</name> <no-traps/> <per-unit-scheduler/> <vlan-tagging/> <unit> <name>101</name></pre>	<pre><configuration xmlns="..."> <interfaces xmlns="..."> <interfaces> <interface> <name>{\$PE_INT_NAME}</name> <no-traps/> <per-unit-scheduler/> <vlan-tagging/> <unit> <name>{\$VLAN_ID}</name></pre>

Side by side, running config on the left, template on the right.

Checking the packages/13vpn/templates/l3vpn-pe.xml service template Junos device part under the legacy <http://xml.juniper.net/xnm/1.1/xnm> namespace, you can observe that it configures interfaces, routing-instances, policy-options, and class-of-service.

You can save the NETCONF/YANG compliant Junos device configuration and diff it against the non-compliant configuration from the previously stored legacy.xml file:

```
admin@ncs# show running-config devices device pe2 config configuration \
| display xml | save new.xml
```

Examining the difference between the configuration in the legacy.xml and new.xml files for the parts covered by the service template:

- 1 There is no longer a single namespace covering all configurations. The configuration is now divided into multiple YANG modules with a namespace for each.
- 2 The /configuration/policy-options/policy-statement/then/community node choice identity is no longer provided with a leaf named key1. Instead, the leaf name is choice-ident, and a choice-value leaf is set.
- 3 The /configuration/class-of-service/interfaces/interface/unit/shaping-rate/rate leaf format has changed from using an int32 value to a string with either no suffix, or a

"k", "m" or "g" suffix. This differs from the other devices controlled by the template, so a new template BW_SUFFIX variable set from the Python code is needed.

To enable the template to handle a Junos device in NETCONF/YANG compliant mode, add the following to the packages/13vpn/templates/13vpn-pe.xml service template:

```

        </interfaces>
        </class-of-service>
    </configuration>
+
+    <configuration xmlns="http://yang.juniper.net/junos/conf/root" tags="merge">
+        <interfaces xmlns="http://yang.juniper.net/junos/conf/interfaces">
+            <interface>
+                <name>{$PE_INT_NAME}</name>
+                <no-traps/>
+                <vlan-tagging/>
+                <per-unit-scheduler/>
+                <unit>
+                    <name>{$VLAN_ID}</name>
+                    <description>Link to CE / {$CE} - {$CE_INT_NAME}</description>
+                    <vlan-id>{$VLAN_ID}</vlan-id>
+                    <family>
+                        <inet>
+                            <address>
+                                <name>{$LINK_PE_ADR}/{LINK_PREFIX}</name>
+                            </address>
+                        </inet>
+                    </family>
+                </unit>
+            </interface>
+        </interfaces>
+        <routing-instances xmlns="http://yang.juniper.net/junos/conf/routing-instances">
+            <instance>
+                <name>{/name}</name>
+                <instance-type>vrf</instance-type>
+                <interface>
+                    <name>{$PE_INT_NAME}.{$VLAN_ID}</name>
+                </interface>
+                <route-distinguisher>
+                    <rd-type>{/as-number}:1</rd-type>
+                </route-distinguisher>
+                <vrf-import>{/name}-IMP</vrf-import>
+                <vrf-export>{/name}-EXP</vrf-export>
+                <vrf-table-label>
+                </vrf-table-label>
+                <protocols>
+                    <bgp>
+                        <group>
+                            <name>{/name}</name>
+                            <local-address>{$LINK_PE_ADR}</local-address>
+                            <peer-as>{/as-number}</peer-as>
+                            <local-as>
+                                <as-number>100</as-number>
+                            </local-as>
+                            <neighbor>
+                                <name>{$LINK_CE_ADR}</name>
+                            </neighbor>
+                        </group>
+                    </bgp>
+                </protocols>
+            </instance>
+        </routing-instances>

```

```

+
<policy-options xmlns="http://yang.juniper.net/junos/conf/policy-options">
+    <policy-statement>
+        <name>{/name}-EXP</name>
+        <from>
+            <protocol>bgp</protocol>
+        </from>
+        <then>
+            <community>
+                <choice-ident>add</choice-ident>
+                <choice-value/>
+                <community-name>{/name}-comm-exp</community-name>
+            </community>
+            <accept/>
+        </then>
+    </policy-statement>
+    <policy-statement>
+        <name>{/name}-IMP</name>
+        <from>
+            <protocol>bgp</protocol>
+            <community>{/name}-comm-imp</community>
+        </from>
+        <then>
+            <accept/>
+        </then>
+    </policy-statement>
+    <community>
+        <name>{/name}-comm-imp</name>
+        <members>target:{/as-number}:1</members>
+    </community>
+    <community>
+        <name>{/name}-comm-exp</name>
+        <members>target:{/as-number}:1</members>
+    </community>
+</policy-options>
<class-of-service xmlns="http://yang.juniper.net/junos/conf/class-of-service">
+    <interfaces>
+        <interface>
+            <name>{$PE_INT_NAME}</name>
+            <unit>
+                <name>{$VLAN_ID}</name>
+                <shaping-rate>
+                    <rate>{$BW_SUFFIX}</rate>
+                </shaping-rate>
+                </unit>
+            </interface>
+        </interfaces>
+    </class-of-service>
+</configuration>
</config>
</device>
</devices>
```

The Python file changes to handle the new BW_SUFFIX variable to generate a string with a suffix instead of an int32:

```

# of the service. These functions can be useful e.g. for
# allocations that should be stored and existing also when the
# service instance is removed.
+
+    @staticmethod
+    def int32_to_numeric_suffix_str(val):
+        for suffix in ["", "k", "m", "g", ""]:
```

```

+
+         suffix_val = int(val / 1000)
+         if suffix_val * 1000 != val:
+             return str(val) + suffix
+         val = suffix_val
+
@ncs.application.Service.create
def cb_create(self, tctx, root, service, proplist):
    # The create() callback is invoked inside NCS FASTMAP and must

```

Code that uses the function and set the string to the service template:

```

tv.add('LOCAL_CE_NET', getIpAddress(endpoint.ip_network))
tv.add('CE_MASK', getNetMask(endpoint.ip_network))
+
tv.add('BW_SUFFIX', self.int32_to_numeric_suffix_str(endpoint.bandwidth))
tv.add('BW', endpoint.bandwidth)
tmpl = ncs.template.Template(service)
tmpl.apply('13vpn-pe', tv)

```

After making the changes to the service template and Python code, reload the updated package(s):

```
$ ncs_cli -u admin -C
admin@ncs# packages reload
```

Re-deploy the MPLS VPN Service Instances

The service instances need to be re-deployed to own the device configuration again:

```
admin@ncs# vpn 13vpn * re-deploy no-networking
```

The service is now in sync with the device configuration stored in NSO CDB:

```
admin@ncs# vpn 13vpn * check-sync
vpn 13vpn ike-a check-sync
in-sync true
vpn 13vpn spotify check-sync
in-sync true
```

When re-deploying the service instances, any issues with the added service template section for the compliant Junos device configuration, such as the added namespaces and nodes, are discovered.

As there is no validation for the rate leaf string with a suffix in the Junos device model, no errors are discovered if it is provided in the wrong format until updating the Junos device. Comparing the device configuration in NSO with the configuration on the device shows such inconsistencies without having to test the configuration with the device:

```
admin@ncs# devices device pe2 compare-config
```

If there are issues, correct them and redo the **re-deploy no-networking** for the service instances.

When all issues have been resolved, the service configuration is in sync with the device configuration, and the NSO CDB device configuration matches to the configuration on the Junos device:

```
$ ncs_cli -u admin -C
admin@ncs# vpn 13vpn * re-deploy
```

The NSO service instances are now in sync with the configuration on the Junos device using the juniper-junos_nc-gen NED.



CHAPTER 21

NED Upgrades and Migration

- [Introduction, page 457](#)
- [The Migrate Action, page 457](#)

Introduction

Many services in NSO rely on NEDs to perform network provisioning. These services map service-specific configuration to the device data models, provided by the NEDs. As the NED packages can be upgraded independently, they can introduce changes in the device YANG models that cause issues for the services using them.

NSO provides tools to migrate between backwards incompatible NED versions. The tools are designed to give you a structured analysis of which paths will change between two NED versions and visibility into the scope of the potential impact that a change in the NED will drive in the service code.

The tools allows for a usage-based analysis of which parts of the NED data model (and instance tree) a particular service has written to. This will give you an (at least opportunistic) sense of which paths must change in the service code.

These features aim to lower the barrier of upgrading NEDs and significantly reduce the amount of uncertainty and side effects that NED upgrades were historically associated with.

The Migrate Action

By using the `/ncs:devices/device/migrate` action you can change the NED major/minor version of a device. The action migrates all configuration and service meta-data. The action can also be executed in parallel on a device group or on all devices matching a NED identity. The procedure for migrating devices is further described in the section called “NED Migration” in *Administration Guide*.

Additionally, the example `examples.ncs/getting-started/developing-with-ncs/26-ned-migration` in the NSO examples collection illustrates how to migrate devices between different NED versions using the `migrate` action.

What makes it particularly useful to a service developer is that the action reports what paths have been modified and the service instances affected by those changes. This information can then be used to prepare the service code to handle the new NED version. If the `verbose` option is used, all service instances are reported instead of just the service points. If the `dry-run` option is used, the action simply reports what it would do. This gives you the chance to do analysis before any actual change is performed.



CHAPTER 22

Service Handling of Ambiguous Device Models

When new NED versions with diverging XML namespaces are introduced, adaptations might be needed in the services for these new NEDs. But not necessarily; it depends on where in the specific NED models that the ambiguities reside. Existing services might not refer to these parts of the model and in that case they do not need any adaptations.

Finding out if and where services need adaptations can be non-trivial. An important exception is template services which check and point out ambiguities at load time (NSO startup). In Java or Python code this is harder and basically falls back to code reviews and testing.

The changes in service code to handle ambiguities are straightforward but different for templates and code.

- [Template Services, page 459](#)
- [Java Services, page 460](#)
- [Python Services, page 461](#)

Template Services

In templates there are new processing instructions *if-ned-id* and *elif-ned-id*. When the template specifies a node in an XML namespace where an ambiguity exists, the *if-ned-id* process instruction is used to resolve that ambiguity.

The processing instruction *else* can be used in conjunction with *if-ned-id* and *elif-ned-id* to capture all other ned-ids.

For the nodes in the XML namespace where no ambiguities occur this process instruction is not necessary.

```
<config-template xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device foreach="{apache-device}">
      <name>{current()}</name>
      <config>
        <?if-ned-id apache-nc-1.0:apache-nc-1.0?>
          <vhosts xmlns="urn:apache">
            <vhost>
              <hostname>{/vhost}</hostname>
              <doc-root>/srv/www/{/vhost}</doc-root>
            </vhost>
          </vhosts>
        <?elif-ned-id apache-nc-1.1:apache-nc-1.1?>
          <public xmlns="urn:apache">
            <vhosts>
```

```

<vhost>
    <hostname>{/vhost}</hostname>
    <aliases>{/vhost}.public</aliases>
    <doc-root>/srv/www/{/vhost}</doc-root>
    </vhost>
</vhosts>
</public>
<?end?>
</config>
</device>
</devices>
</config-template>

```

Java Services

In Java the service code must handle the ambiguities by code where the devices ned-id is tested before setting the nodes and values for the diverging paths.

The ServiceContext class has a new convenience method, `getNEDIdByDeviceName` which helps retrieving the ned-id from the device name string.

```

@ServiceCallback(servicePoint="websiteservice",
                 callType=ServiceCBType.CREATE)
public Properties create(ServiceContext context,
                        NavuNode service,
                        NavuNode root,
                        Properties opaque)
throws DpCallbackException {

    ...

    NavuLeaf elemName = elem.leaf(Ncs._name_);
    NavuContainer md = root.container(Ncs._devices_).
        list(Ncs._device_).elem(elemName.toKey());

    String ipv4Str = baseIp + ((subnet<<3) + server);
    String ipv6Str = "::ff:ff:" + ipv4Str;
    String ipStr = ipv4Str;
    String nedIdStr =
        context.getNEDIdByDeviceName(elemName.valueAsString());
    if ("webserver-nc-1.0:webserver-nc-1.0".equals(nedIdStr)) {
        ipStr = ipv4Str;
    } else if ("webserver2-nc-1.0:webserver2-nc-1.0"
               .equals(nedIdStr)) {
        ipStr = ipv6Str;
    }

    md.container(Ncs._config_).
        container(webserver.prefix, webserver._wsConfig_).
        list(webserver._listener_).
        sharedCreate(new String[] {ipStr, "+8008});

    ms.list(lb._backend_).sharedCreate(
        new String[]{baseIp + ((subnet<<3) + server++),
                    "+8008});

    ...

    return opaque;
} catch (Exception e) {
    throw new DpCallbackException("Service create failed", e);
}

```

}

Python Services

In the Python API there is also a need to handle ambiguities by checking the ned-id before setting the diverging paths. Use get_ned_id() from ncs.application to resolve ned-ids

```
import ncs

def _get_device(service, name):
    dev_path = '/ncs:devices/ncs:device{%s}' % (name, )
    return ncs.maagic.cd(service, dev_path)

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

        for name in service.apache_device:
            self.create_apache_device(service, name)

        template = ncs.template.Template(service)
        self.log.info(
            'applying web-server-template for device {}'.format(name))
        template.apply('web-server-template')
        self.log.info(
            'applying load-balancer-template for device {}'.format(name))
        template.apply('load-balancer-template')

    def create_apache_device(self, service, name):
        dev = _get_device(service, name)
        if 'apache-nc-1.0:apache-nc-1.0' == ncs.application.get_ned_id(dev):
            self.create_apache1_device(dev)
        elif 'apache-nc-1.1:apache-nc-1.1' == ncs.application.get_ned_id(dev):
            self.create_apache2_device(dev)
        else:
            raise Exception('unknown ned-id {}'.format(get_ned_id(dev)))

    def create_apache1_device(self, dev):
        self.log.info(
            'creating config for apache1 device {}'.format(dev.name))
        dev.config.ap__listen_ports.listen_port.create(("*", 8080))
        dev.config.ap__clash = dev.name

    def create_apache2_device(self, dev):
        self.log.info(
            'creating config for apache2 device {}'.format(dev.name))
        dev.config.ap__system.listen_ports.listen_port.create(("*", 8080))
        dev.config.ap__clash = dev.name
```




CHAPTER 23

Scaling and Performance Optimization

With an increasing number of services and managed devices in NSO, performance becomes a more important aspect of the system. At the same time, other aspects, such as the way you organize code, also start playing an important role when using NSO on a bigger scale. The following chapter examines these concerns and presents the available options for scaling your NSO automation solution.

- [Understanding Your Use Case, page 463](#)
- [Where to Start?, page 465](#)
- [Divide the Work Correctly, page 465](#)
- [Optimizing Device Communication, page 465](#)
- [Improving Subscribers, page 466](#)
- [Minimizing Concurrency Conflicts, page 466](#)
- [Fine-tuning the Concurrency Parameters, page 467](#)
- [Enabling Even More Parallelism, page 467](#)
- [Limit sync-from, page 468](#)
- [Designing for Maximal Transaction Throughput, page 468](#)
- [Scaling RAM and Disk, page 488](#)
- [Checklists, page 493](#)
- [Hardware Sizing, page 494](#)

Understanding Your Use Case

NSO allows you to tackle different automation challenges and every solution has its own specifics. Therefore, the best approach to scaling depends on the way the solution is implemented. What works in one case may be useless, or effectively degrade performance, for another. You must first analyze and understand how your particular use case behaves, which will then allow you to take the right approach to scaling.

When trying to improve the performance, a very good, possibly even the best starting point is to inspect the tracing data. Tracing is further described in [Chapter 30, Progress Trace](#). Yet a simple `commit | details` command already provides a lot of useful data.

Example 192. Example progress trace output for a service

```
admin@ncs(config-mysvc-test)# commit | details
```

```

2022-09-16T09:17:48.977 applying transaction...
entering validate phase for running usid=54 tid=225 trace-id=3a4a3b7f-a09f-4f9d-b05e-1656310ea5b
2022-09-16T09:17:48.977 creating rollback checkpoint... ok (0.000 s)
2022-09-16T09:17:48.978 creating rollback file... ok (0.004 s)
2022-09-16T09:17:48.983 creating pre-transform checkpoint... ok (0.000 s)
2022-09-16T09:17:48.983 run pre-transform validation... ok (0.000 s)
2022-09-16T09:17:48.983 creating transform checkpoint... ok (0.000 s)
2022-09-16T09:17:48.983 run transforms and transaction hooks...
2022-09-16T09:17:48.985 taking service write lock... ok (0.000 s)
2022-09-16T09:17:48.985 holding service write lock...
2022-09-16T09:17:48.986 service /mysvc[name='test']: run service... ok (0.012 s)
2022-09-16T09:17:48.999 run transforms and transaction hooks: ok (0.016 s)
2022-09-16T09:17:48.999 creating validation checkpoint... ok (0.000 s)
2022-09-16T09:17:49.000 mark inactive... ok (0.000 s)
2022-09-16T09:17:49.001 pre validate... ok (0.000 s)
2022-09-16T09:17:49.001 run validation over the changeset... ok (0.000 s)
2022-09-16T09:17:49.002 run dependency-triggered validation... ok (0.000 s)
2022-09-16T09:17:49.003 check configuration policies... ok (0.000 s)
2022-09-16T09:17:49.003 check for read-write conflicts... ok (0.000 s)
2022-09-16T09:17:49.004 taking transaction lock... ok (0.000 s)
2022-09-16T09:17:49.004 holding transaction lock...
2022-09-16T09:17:49.004 check for read-write conflicts... ok (0.000 s)
2022-09-16T09:17:49.004 applying service meta-data... ok (0.000 s)
leaving validate phase for running usid=54 tid=225 trace-id=3a4a3b7f-a09f-4f9d-b05e-1656310ea5b6
entering write-start phase for running usid=54 tid=225 trace-id=3a4a3b7f-a09f-4f9d-b05e-1656310ea5b6
2022-09-16T09:17:49.005 cdb: write-start
2022-09-16T09:17:49.006 ncs-internal-service-mux: write-start
2022-09-16T09:17:49.006 ncs-internal-device-mgr: write-start
2022-09-16T09:17:49.007 cdb: match subscribers... ok (0.000 s)
2022-09-16T09:17:49.007 cdb: create pre commit running... ok (0.000 s)
2022-09-16T09:17:49.007 cdb: write changeset... ok (0.000 s)
2022-09-16T09:17:49.008 check data kickers... ok (0.000 s)
leaving write-start phase for running usid=54 tid=225 trace-id=3a4a3b7f-a09f-4f9d-b05e-1656310ea5b6
entering prepare phase for running usid=54 tid=225 trace-id=3a4a3b7f-a09f-4f9d-b05e-1656310ea5b6
2022-09-16T09:17:49.009 cdb: prepare
2022-09-16T09:17:49.009 ncs-internal-device-mgr: prepare
2022-09-16T09:17:49.022 device ex1: push configuration...
leaving prepare phase for running usid=54 tid=225 trace-id=3a4a3b7f-a09f-4f9d-b05e-1656310ea5b6
entering commit phase for running usid=54 tid=225 trace-id=3a4a3b7f-a09f-4f9d-b05e-1656310ea5b6
2022-09-16T09:17:49.130 cdb: commit
2022-09-16T09:17:49.130 cdb: switch to new running... ok (0.000 s)
2022-09-16T09:17:49.132 ncs-internal-device-mgr: commit
2022-09-16T09:17:49.149 device ex1: push configuration: ok (0.126 s)
2022-09-16T09:17:49.151 holding service write lock: ok (0.166 s)
2022-09-16T09:17:49.151 holding transaction lock: ok (0.147 s)
leaving commit phase for running usid=54 tid=225 trace-id=3a4a3b7f-a09f-4f9d-b05e-1656310ea5b6
2022-09-16T09:17:49.151 applying transaction: ok (0.174 s)
Commit complete.
admin@ncs(config-mysvc-test)#

```

Pay attention to the time NSO spends doing specific tasks. For a simple service, these are mainly:

- Validate service data (pre-transform validation)
- Run service mapping logic
- Validate produced configuration (changeset)
- Push changes to affected devices
- Commit the new configuration

Tracing data can often quickly reveal a bottleneck, a hidden delay, or some other unexpected inefficiency in your code. The best strategy is to first address any such concerns if they show up, since only well-

performing code is a good candidate for further optimization. Otherwise, you might find yourself optimizing the wrong parameters and hitting a dead end. Visualizing the progress trace is often helpful in identifying bottlenecks. See [the section called “Measuring Transaction Throughput”](#).

Analyzing the service in isolation can yield useful insight. But it may also lead you in the wrong direction, because some issues only manifest under load and the data from a live system can surprise you. That is why NSO supports different ways of exposing tracing information, including operational data and notification events. Remember to always verify that your observations and assumptions hold for a live, production system, too.

Where to Start?

The times for different parts of the transaction, as reported by the tracing data, are very useful in determining where to focus your efforts.

For example, if your service data model uses a very broad `must` or similar XPath statement, then NSO may potentially need to evaluate thousands of data entries. Such evaluation requires a considerable amount of additional processing and is, in turn, reflected in increased time spent in validation. The solution in this case is to limit the scope of the data referenced in the YANG constraint, which you can often achieve with a more specific XPath expression.

Similarly, if a significant amount of time is spent constructing a service mapping, perhaps there is some redundant work occurring that you could optimize? Sometimes, however, provisioning requires calls to other systems or some computationally expensive operation, which you cannot easily manage without. Then you might want to consider splitting the provisioning process into smaller pieces, using nano services, for example. See [the section called “Simplify the Per-Device Concurrent Transaction Creation Using a Nano Service”](#) for an example use-case and references to the nano service documentation.

In general, your own code for a single transaction with no additional load on NSO should execute quickly (sub-second, as a rule of thumb). The faster each service or action code is, the better the overall system performance. Using a service design pattern to both improve performance, scale, and avoid conflicts is described by [the section called “Design to Minimize Conflicts”](#).

Divide the Work Correctly

Things such as reading external data or large computations should not be done inside the create code. Consider using an action to encapsulate these functions. An action does not run under the lock unless it triggers a transaction and can perform side effects as desired.

There are several ways to utilize an action:

- An action is allowed to perform side-effects.
- An action can read operational data from devices or external systems.
- An action can write values to operational data in CDB, for later use from the service.
- An action can write configuration to CDB, potentially triggering a service.

Actions can be used together with nano services, see [the section called “Simplify the Per-Device Concurrent Transaction Creation Using a Nano Service”](#).

Optimizing Device Communication

With the default configuration, one of the first things you might notice standing out in the tracing data is that pushing device configuration takes a significant amount of time compared to other parts of service provisioning. Why is that?

All changes in NSO happen inside a transaction. Network devices participate in the transaction, which gives you the all-or-nothing behavior, to ensure correctness and consistency across the network. But network communication is not instantaneous and a transaction in NSO holds a lock while waiting for devices to process the change. This way, changes to network devices are serialized, even when there are multiple simultaneous transactions. However, a lock blocks other transactions from proceeding, ultimately limiting the overall NSO transaction rate.

So, in many cases the NSO system is not really resource constrained but merely experiencing lock contention. Therefore, making locks as short as possible is the best way to improve performance. In the example trace from the section called “[Understanding Your Use Case](#)”, most of the time is spent in the *prepare phase*, where configuration changes are propagated to the network devices. Change propagation requires a management session with each participating device, as well as updating and validating the new configuration on the device side. Understandably, all of these tasks take time.

NSO allows you to influence this behavior. Please take a look at the section called “Commit Queue” in *User Guide* documentation on how to avoid long device locks with commit queues and the trade-offs they bring. Usually, enabling the commit queue feature is the first and the most effective step to significantly improving transaction times.

Improving Subscribers

The CDB subscriber mechanism is used to notify application code about CDB changes and runs at the end of the transaction commit, inside a global lock. Due to this fact, the number and configuration of subscribers affects performance and should be investigated early in your performance optimization efforts.

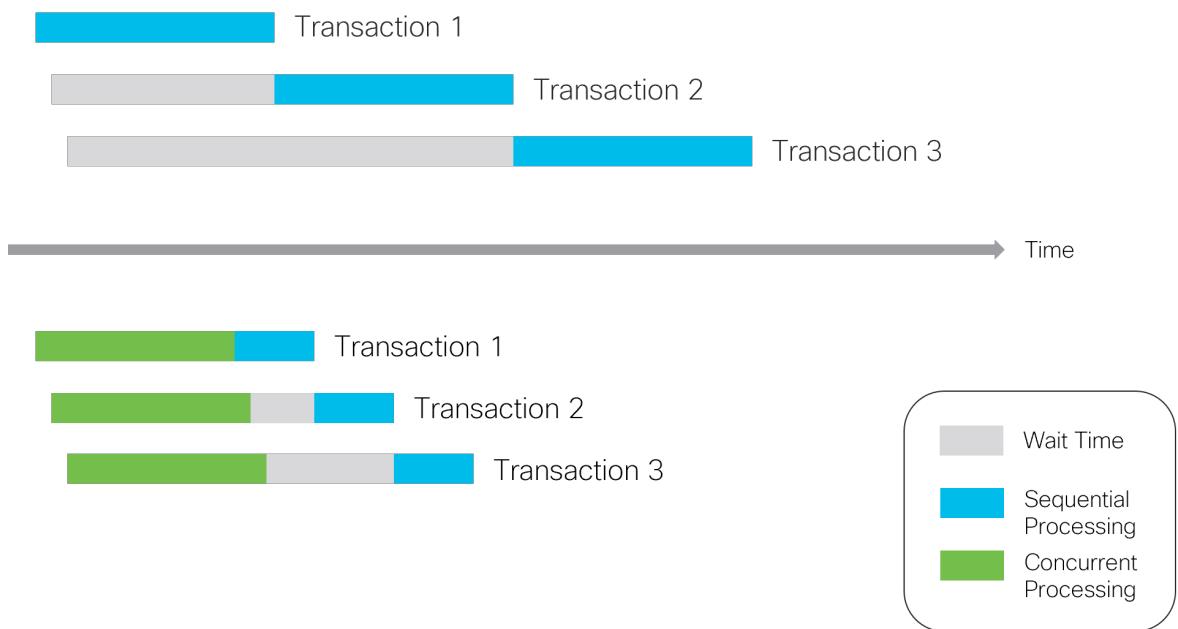
A badly implemented subscriber prolongs the time the transaction holds the lock, preventing other transactions from completing, in addition to the original transaction taking more time to commit. There are mainly two reasons for suboptimal operation: either the subscriber is too broad and must process too many (irrelevant) changes, or it performs more work inside the lock as necessary. As a recommended practice, subscriber should only note the changes and schedule the processing to be done later, in order to return and release the lock as quickly as possible.

Moreover, subscribers incur processing overhead regardless of their implementation because NSO needs to communicate with the custom subscriber code, typically written in Java or Python.

That is why modern, performant code in NSO should use the kicker mechanism instead of implementing custom subscribers. While it is still possible to create a badly performing kicker, you are less likely to do so inadvertently. In most situations kickers are also easier to implement and troubleshoot. You can read more on kickers in [Chapter 28, Kicker](#).

Minimizing Concurrency Conflicts

The time it takes to complete a transaction is certainly an important performance metric. However, after a certain point, it gets increasingly hard or even impossible to get meaningful improvement from optimizing each individual transaction. As it turns out, on a busy system, there are usually multiple outstanding requests. So, instead of trying to process each as fast as possible one after another, the system might process them in parallel.

Figure 193. Running Transactions Sequentially and in Parallel

In practice and as the figure shows, some parts must still be processed sequentially to ensure transactional properties. But there is a significant gain in the overall time it takes to process all transactions in a busy system, even though each might take a little longer individually due to the concurrency overhead.

Throughput then becomes a more relevant metric. It is the number of requests or transactions that the system can process in a given time unit. While throughput is still related to individual transaction times, other factors also come into play. An important one is the way in which NSO implements concurrency and the interaction between the transaction system and your, user, code. Designing for transaction throughput is covered in detail later in this chapter, and the NSO concurrency model is detailed in [Chapter 24, NSO Concurrency Model](#).

The chapter provides guidance on identifying transaction conflicts and what affects their occurrence, so you can make your code more resistant to producing them. Conflicts arise more frequently on busier systems and negatively affect throughput, which makes them a good candidate for optimization.

Fine-tuning the Concurrency Parameters

Depending on the specifics of the server running NSO, additional performance improvement might be possible by fine-tuning the `transaction-limits` set of configuration parameters in `ncs.conf`. Please see the `ncs.conf(1)` manpage for details.

Enabling Even More Parallelism

If you are experiencing high resource utilization, such as memory and CPU usage, while individual transactions are optimized to execute fast and the rate of conflicts is low, it's possible you are starting to see the level of demand that pushes the limits of this system.

First, you should try adding more resources, in a “scale up” manner, if possible. At the same time, you might also have some services that are using an older, less performant user code execution model. For example, the way Python code is executed is controlled by the `callpoint-model` option, described in the section called “The application component”, which you should ensure is set to the most performant setting.

Regardless, a single system cannot scale indefinitely. After you have exhausted all other options, you will need to “scale out,” that is, split the workload across multiple NSO instances. You can achieve this by using the Layered Service Architecture (LSA) approach. But the approach has its trade-offs, so make sure it provides the right benefits in your case. The LSA is further documented in Chapter 1, *LSA Overview in Layered Service Architecture*.

Limit sync-from

In a brownfield environment, where the configuration is not 100% automated and controlled by NSO alone but also written to by other systems or operators, NSO is bound to end up out-of-sync with the device.

How to handle synchronization is a big topic, and it is vital to understand what it means to you when things are out-of-sync. This will help guide your strategy.

If NSO is frequently brought out-of-sync, it can be tempting to invoke **sync-from** from the create callback. While it does achieve a higher degree of reliability in the sense that service modifications won't return an out-of-sync error, the impact on performance is usually catastrophic. The typical **sync-from** operation takes orders of magnitudes longer than the typical service modification, and transactional throughput will suffer greatly.

But other alternatives are often better:

- You can synchronize the configuration from the device when it reports a change rather than when the service is modified by listening for configuration change events from the device, e.g., via RESTCONF or NETCONF notifications, SNMP traps, or syslog, and invoking **sync-from** or **partial-sync-from** when another party (not NSO) has modified the device. See also the section called “[Partial Sync](#)”.
- Using the **devices sync-from** command does not hold the transaction lock and run across devices concurrently, which reduces the total amount of time spent time synchronizing. This is particularly useful for periodic synchronization to lower the risk of being out-of-sync when committing configuration changes.
- Using the **no-overwrite** commit flag, you can be more lax about being in sync and focus on not overwriting modified configuration.
- If the configuration is 100% automated and controlled by NSO alone, using **out-of-sync-behaviour accept**, you can completely ignore if the device is in sync or not.
- Letting your modification fail with out-of-sync error and handling that error at the calling side.

Designing for Maximal Transaction Throughput

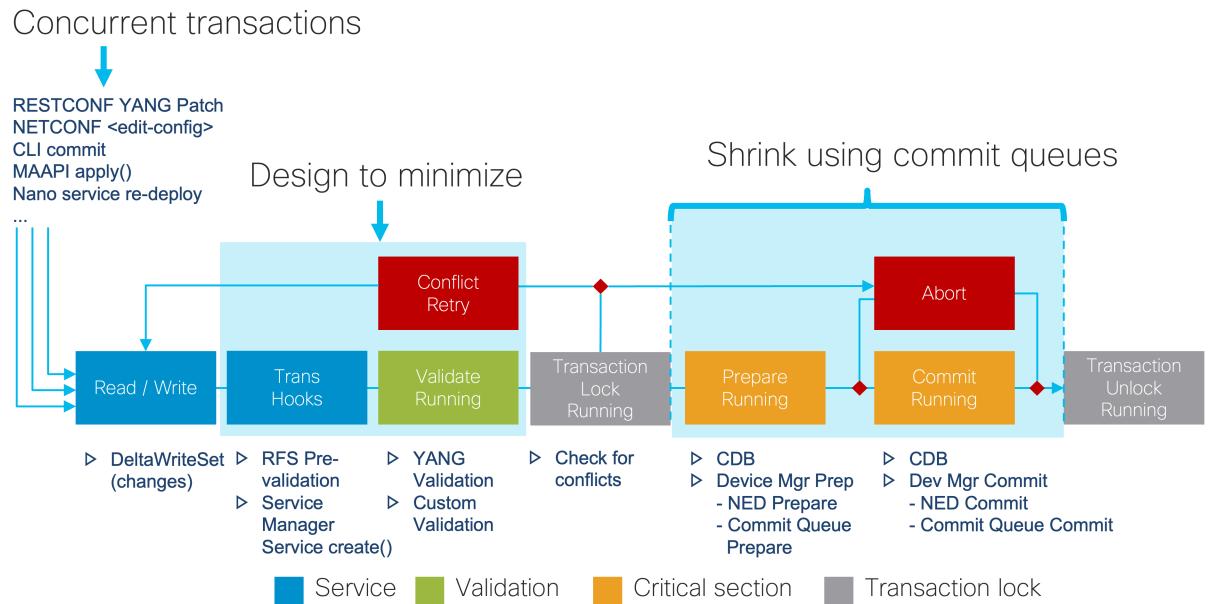
Maximal transaction throughput refers to the maximum number of transactions a system can handle within a given period. Factors that can influence maximal transaction throughput include:

- Hardware capabilities (e.g., processing power, memory).
- Software efficiency.
- Network bandwidth.
- The complexity of the transactions themselves.

Besides making sure the system hardware capabilities and network bandwidth are not a bottleneck, there are four areas where the NSO user can significantly affect the transaction throughput performance for an NSO node:

- Run multiple transactions concurrently. For example, multiple concurrent RESTCONF or NETCONF edits, CLI commits, MAAPI `apply()`, nano service re-deploy, etc.

- Design to avoid conflicts and minimize the service create() and validation implementation. For example, in service templates and code mapping to devices or other service instances, YANG must statements with XPath expressions or validation code.
- Using commit queues to exclude the time to push configuration changes to devices from inside the transaction lock.
- Simplify using nano and stacked services. If the processor where NSO with a stacked service runs becomes a severe bottleneck, the added complexity of migrating the stacked service to an LSA setup can be motivated. LSA helps expose only a single service instance when scaling up the number of devices by increasing the number of available CPU cores beyond a single processor.

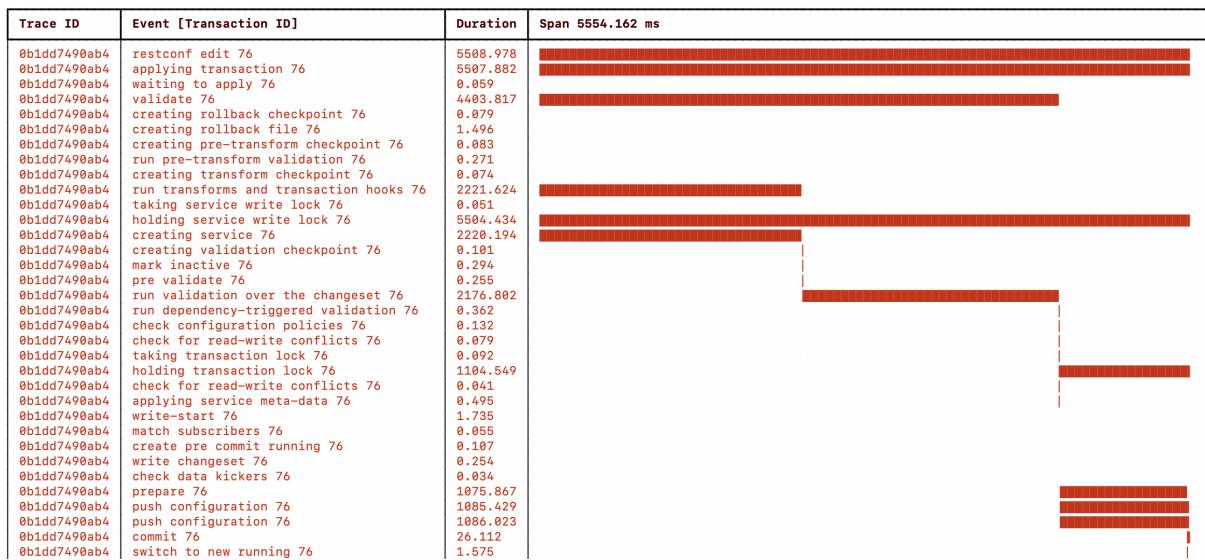


Measuring Transaction Throughput

Measuring transaction performance includes measuring the total wall-clock time for the service deployment transaction(s) and using the detailed NSO progress trace of the transactions to find bottlenecks. The developer log helps debug the NSO internals, and the XPath trace log helps find misbehaving XPath expressions used in, for example, YANG must statements.

The picture below shows a visualization of the NSO progress trace when running a single transaction for two service instances configuring a device each:

Running the `perf-trans` Example Using a Single Transaction



The total RESTCONF edit took ~5 seconds, and the service mapping (“creating service” event) and validation (“run validation ...” event) were done sequentially for the service instances and took 2 seconds each. The configuration push to the devices was done concurrently in 1 second.

For progress trace documentation, see [Chapter 30, Progress Trace](#).

Running the `perf-trans` Example Using a Single Transaction

The `perf-trans` example from the NSO example set explores the opportunities to improve the wall-clock time performance and utilization, as well as opportunities to avoid common pitfalls.

The example uses simulated CPU loads for service creation and validation work. Device work is simulated with `sleep()` as it will not run on the same processor in a production system.

The example shows how NSO can benefit from running many transactions concurrently if the service and validation code allow concurrency. It uses the NSO progress trace feature to get detailed timing information for the transactions in the system.

The provided code sets up an NSO instance that exports tracing data to a `.csv` file, provisions one or more service instances, which each map to a device, and shows different (average) transaction times and a graph to visualize the sequences plus concurrency.

Play with the `perf-trans` example by tweaking the `measure.py` script parameters:

```
-nt NTRANS, --ntrans NTRANS
The number of transactions updating the same service in parallel. For this
example, we use NTRANS parallel RESTCONF plain patch.
Default: 1.

-nw NWORK, --nwork NWORK
Work per transaction in the service creation and validation phases. One
second of CPU time per work item.
Default: 3 seconds of CPU time.

-nd 0..10, --ndtrans 0..10
Number of devices the service will configure per service transaction.
Default: 1
```

```

-dd DDELAY, --ddelay DDELAY
    Transaction delay (simulated by sleeping) on the netsim devices (seconds).
    Default: 0s

-cq {async, sync, bypass, none}, --cqparam {async, sync, bypass, none}
    Commit queue behavior. Select "none" to use the global or device setting.
    Default: none

```

See the README in the perf-trans example for details.

To run the `perf-trans` example from the NSO example set and recreate the variant shown in the progress trace above:

```

cd $NCS_DIR/examples.ncs/development-guide/concurrency-model/perf-trans
make NDEVS=2 python
python3 measure.py --ntrans 1 --nwork 2 --ndtrans 2 --cqparam bypass --ddelay 1
python3 ../common/simple_progress_trace_viewer.py $(ls logs/*.csv)

```

The following is a sequence diagram and the progress trace of the example, describing the transaction t1. The transaction deploys service configuration to the devices using a single RESTCONF `patch` request to NSO and then NSO configures the netsim devices using NETCONF:

```

RESTCONF      service      validate      push config
patch         create       config        ndtrans=2           netsim
ntrans=1     nwork=2     nwork=2      cqparam=bypass      device      ddelay=1
               t1 -----> 2s -----> 2s -----> ex0 -----> 1s
                                         \-----> ex1 -----> 1s
               wall-clock 2s          2s                      1s = 5s

```

The only part running concurrently in the example above was configuring the devices. It is the most straightforward option if transaction throughput performance is not a concern or the service creation and validation work are insignificant. A single transaction service deployment will not need to use commit queues as it is the only transaction holding the transaction lock configuring the devices inside the critical section. See the “holding transaction lock” event in the progress trace above.

Stop NSO and the netsim devices:

```
make stop
```

Concurrent Transactions

Everything from smartphones and tablets to laptops, desktops, and servers now contain multi-core processors. For maximal throughput, the powerful multi-core systems need to be fully utilized. This way, the wall clock time is minimized when deploying service configuration changes to the network, which is usually equated with performance. Therefore, enabling NSO to spread as much work as possible across all available cores becomes important. The goal is to have service deployments maximize their utilization of the total available CPU time to deploy services faster to the users who ordered them.

Close to full utilization of every CPU core when running under maximal load, for example, ten transactions to ten devices, is ideal, as some process viewer tools such as `htop` visualize with meters:

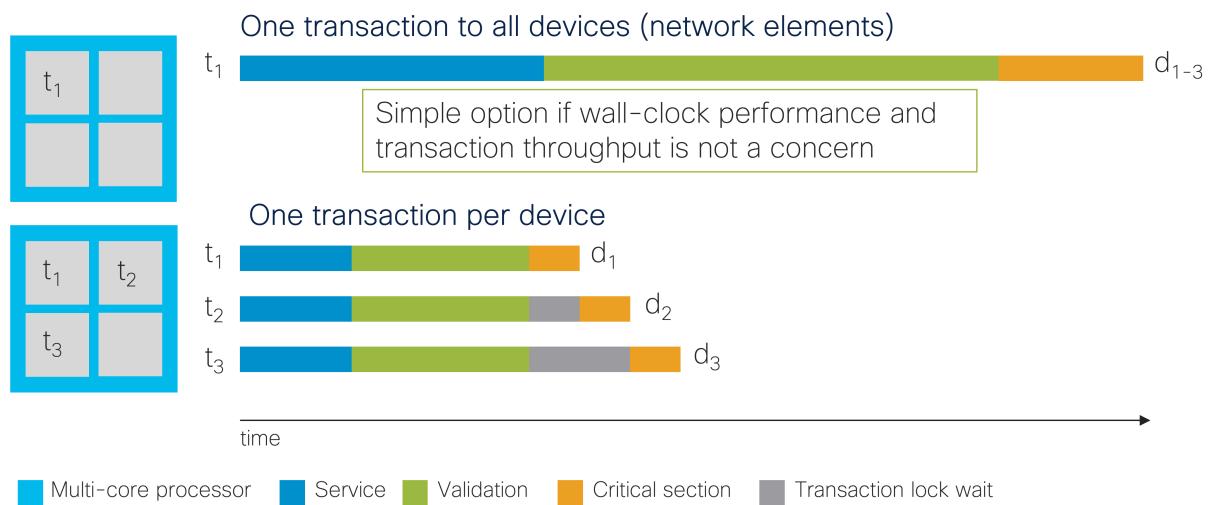
```

0[|||||||||||||||||]100.0%
1[|||||||||||||||||]100.0%
2[|||||||||||||||]99.3%
3[|||||||||||||||]99.3%
4[|||||||||||||||]99.3%
5[|||||||||||||||]99.3%
6[|||||||||||||||]98.7%
7[|||||||||||||||]98.7%

```

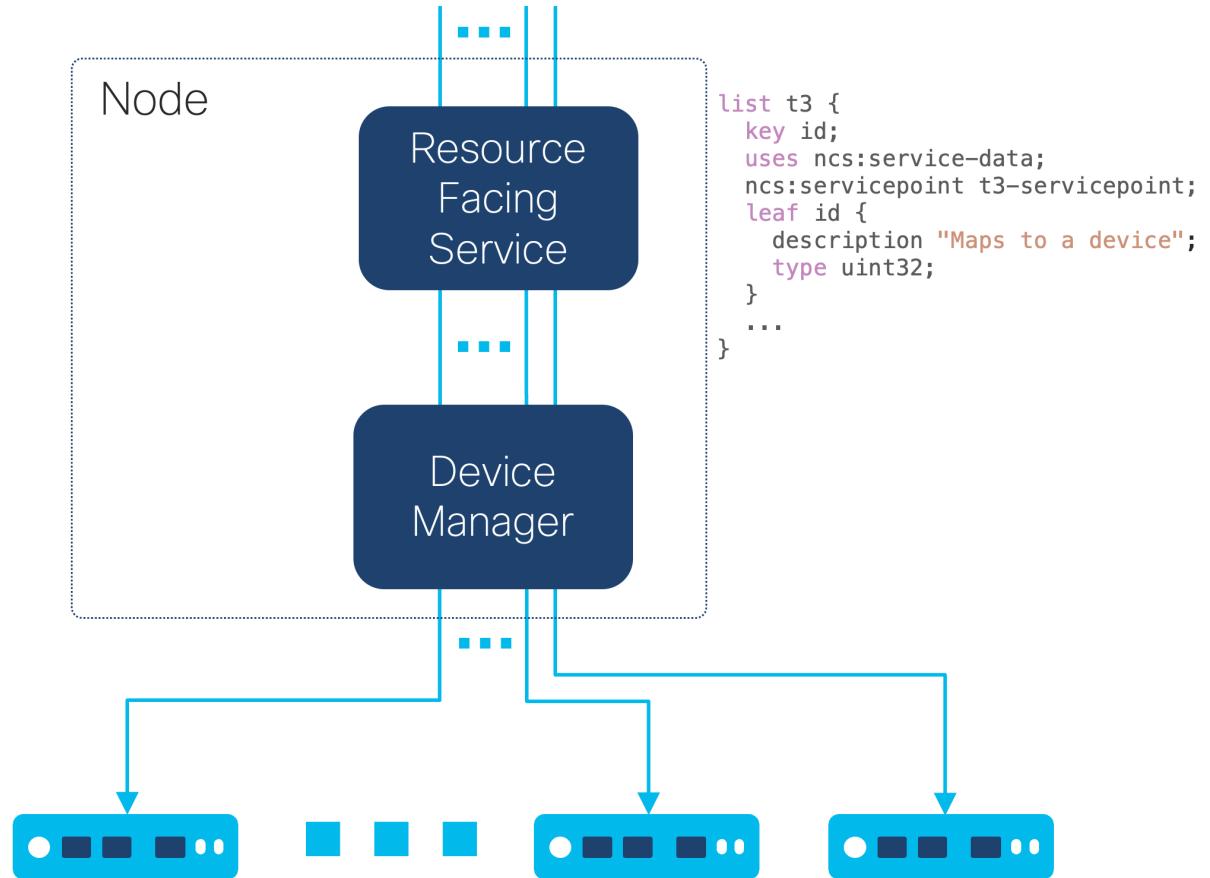
```
8[|||||||||||||||||||||||||||||||||] 98.7%]  
9[|||||||||||||||||||||||||||||] 98.7%]  
...
```

One transaction per RFS instance and device will allow each NSO transaction to run on a separate core concurrently. Multiple concurrent RESTCONF or NETCONF edits, CLI commits, MAAPI `apply()`, nano service re-deploy, etc. Keep the number of running concurrent transactions equal to or below the number of cores available in the multi-core processor to avoid performance degradation due to increased contention on system internals and resources. NSO helps by limiting the number of transactions applying changes in parallel to, by default, the number of logical processors (e.g., CPU cores). See `ncs.conf(5)` in *Manual Pages* under `/ncs-config/transaction-limits/max-transactions` for details.



Design to Minimize Conflicts

Conflicts between transactions and how to avoid them are described in [the section called “Minimizing Concurrency Conflicts”](#) and in detail by [Chapter 24, NSO Concurrency Model](#). While NSO can handle transaction conflicts gracefully with retries, retries affect transaction throughput performance. A simple but effective design pattern to avoid conflicts is to update one device with one Resource Facing Service (RFS) instance where service instances do not read each other’s configuration changes.



Design to Minimize Service and Validation Processing Time

An overly complex service or validation implementation using templates, code, and XPath expressions increases the processing required and, even if transactions are processed concurrently, will affect the wall-clock time spent processing and, thus, transaction throughput.

When data processing performance is of interest, the best practice rule of thumb is to ensure that `must` and `when` statement XPath expressions in YANG models and service templates are only used as necessary and kept as simple as possible.

If a service creates a significant amount of configuration data for devices, it is often significantly faster using a single MAAPI `shared_set_values()` call instead of using multiple `create()` and `set()` calls or a service template.

Running the `perf-setvals` Example Using a Single Call to MAAPI `shared_set_values()`

The `perf-setvals` example writes configuration to an access control list and a route list of a Cisco Adaptive Security Appliance (ASA) device. It uses either MAAPI Python `create()` and `set()` calls, Python `shared_set_values()`, or Java `sharedSetValues()` to write the configuration in XML format.

To run the `perf-setvals` example using MAAPI Python `create()` and `set()` calls to create 3000 rules and 3000 routes on one device:

Design to Minimize Service and Validation Processing Time

```
cd $NCS_DIR/examples.ncs/development-guide/concurrency-model/perf-setvals
./measure.sh -r 3000 -t py_create -n true
```

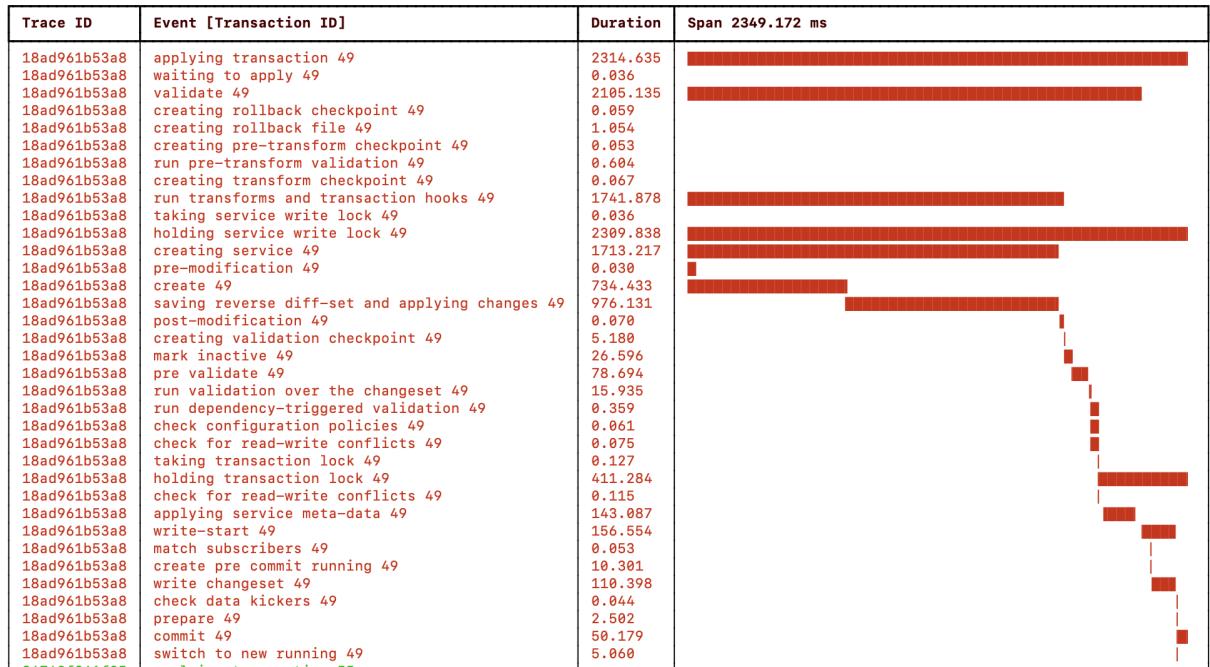
The commit uses the no-networking parameter to skip pushing the configuration to the simulated and un-proportionally slow Cisco ASA netsim device. The resulting NSO progress trace:

Trace ID	Event [Transaction ID]	Duration	Span 5585.874 ms
f6c009c46d07	applying transaction 48	5550.757	
f6c009c46d07	waiting to apply 48	0.033	
f6c009c46d07	validate 48	5297.317	
f6c009c46d07	creating rollback checkpoint 48	0.084	
f6c009c46d07	creating rollback file 48	0.982	
f6c009c46d07	creating pre-transform checkpoint 48	0.053	
f6c009c46d07	run pre-transform validation 48	0.599	
f6c009c46d07	creating transform checkpoint 48	0.074	
f6c009c46d07	run transforms and transaction hooks 48	4782.374	
f6c009c46d07	taking service write lock 48	0.055	
f6c009c46d07	holding service write lock 48	5545.673	
f6c009c46d07	creating service 48	4743.962	
f6c009c46d07	pre-modification 48	0.048	
f6c009c46d07	create 48	3511.753	
f6c009c46d07	saving reverse diff-set and applying changes 48	1229.111	
f6c009c46d07	post-modification 48	0.065	
f6c009c46d07	creating validation checkpoint 48	6.023	
f6c009c46d07	mark inactive 48	30.139	
f6c009c46d07	pre validate 48	88.687	
f6c009c46d07	run validation over the changeset 48	19.736	
f6c009c46d07	run dependency-triggered validation 48	0.257	
f6c009c46d07	check configuration policies 48	0.197	
f6c009c46d07	check for read-write conflicts 48	0.099	
f6c009c46d07	taking transaction lock 48	0.174	
f6c009c46d07	holding transaction lock 48	579.916	
f6c009c46d07	check for read-write conflicts 48	0.091	
f6c009c46d07	applying service meta-data 48	248.743	
f6c009c46d07	write-start 48	191.270	
f6c009c46d07	match subscribers 48	0.057	
f6c009c46d07	create pre commit running 48	10.607	
f6c009c46d07	write changeset 48	142.774	
f6c009c46d07	check data kickers 48	0.060	
f6c009c46d07	prepare 48	3.899	
f6c009c46d07	commit 48	57.995	
f6c009c46d07	switch to new running 48	6.238	

Next, run the perf-setvals example using a single MAAPI Python shared_set_values() call to create 3000 rules and 3000 routes on one device:

```
./measure.sh -r 3000 -t py_setvals_xml -n true
```

The resulting NSO progress trace:



Using the MAAPI `shared_set_values()` function, the service `create` callback is, for this example, ~5x faster than using the MAAPI `create()` and `set()` functions. The total wall-clock time for the transaction is more than 2x faster, and the difference will increase for larger transactions.

Stop NSO and the netsim devices:

```
make stop
```

Use a Data-Kicker Instead of a CDB Subscriber

A kicker triggering on a CDB change, a data-kicker, should be used instead of a CDB subscriber when the action taken does not have to run inside the transaction lock, i.e., the critical section of the transaction. A CDB subscriber will be invoked inside the critical section and, thus, will have negative impact on the transaction throughput. See the section called “[Improving Subscribers](#)” for more details.

Shorten the Time Used for Writing Configuration to Devices

Writing to devices and other network elements that are slow to configure will stall transaction throughput if you do not enable commit queues, as transactions waiting for the transaction lock to be released cannot start configuring devices before the transaction ahead of them is done writing. For example, if one device is configured using CLI transported with [IP over Avian Carriers](#), the transactions, including such a device, will significantly stall transactions behind it going to devices supporting [RESTCONF](#) or [NETCONF](#) over a fast optical transport. Where transaction throughput performance is a concern, choosing devices that can be configured efficiently to implement their part of the service configuration is wise.

Running the perf-trans Example Using One Transaction per Device

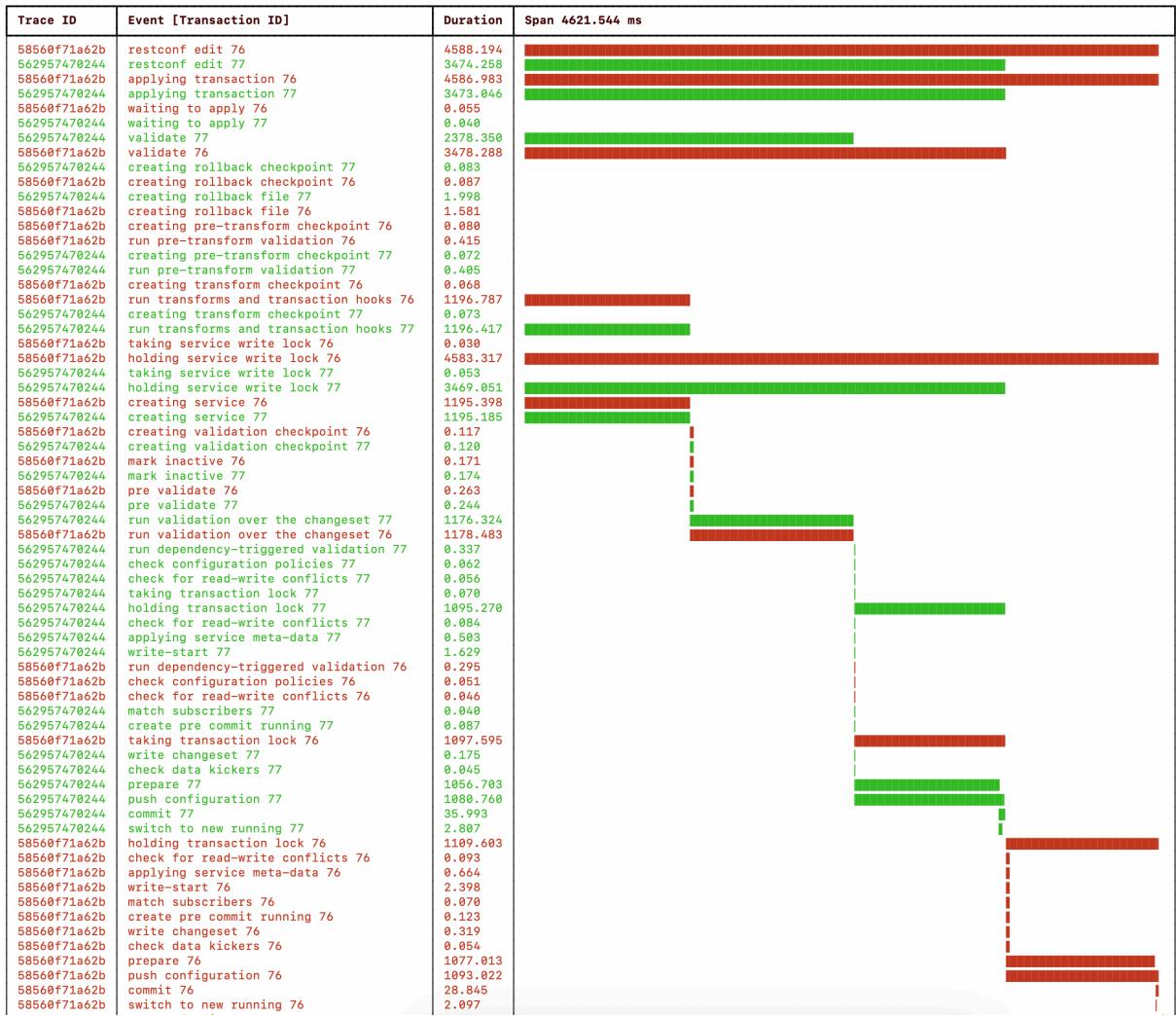
Dividing the service creation and validation work into two separate transactions, one per device, allows the work to be spread across two CPU cores in a multi-core processor. To run the `perf-trans` example with the work divided into one transaction per device:

```
cd $NCS_DIR/examples.ncs/development-guide/concurrency-model/perf-trans
make stop clean NDEVS=2 python
```

Running the perf-trans Example Using One Transaction per Device

```
python3 measure.py --ntrans 2 --nwork 1 --ndtrans 1 --cqparam bypass --ddelay 1
python3 ./common/simple_progress_trace_viewer.py $(ls logs/*.csv)
```

The resulting NSO progress trace:



A sequence diagram with transactions t1 and t2 deploying service configuration to two devices using RESTCONF patch requests to NSO with NSO configuring the netsim devices using NETCONF:

```
RESTCONF    service    validate    push config
patch        create     config      ndtrans=1      netsim           netsim
ntrans=2     nwork=1   nwork=1   cqparam=bypass  device  ddelay=1  device  ddelay=1
              t1 -----> ls -----> ls -----> ex0 ---> ls
              t2 -----> ls -----> ls -----> ex1 ---> ls
              wall-clock ls           ls                           1s           1s = 4s
```

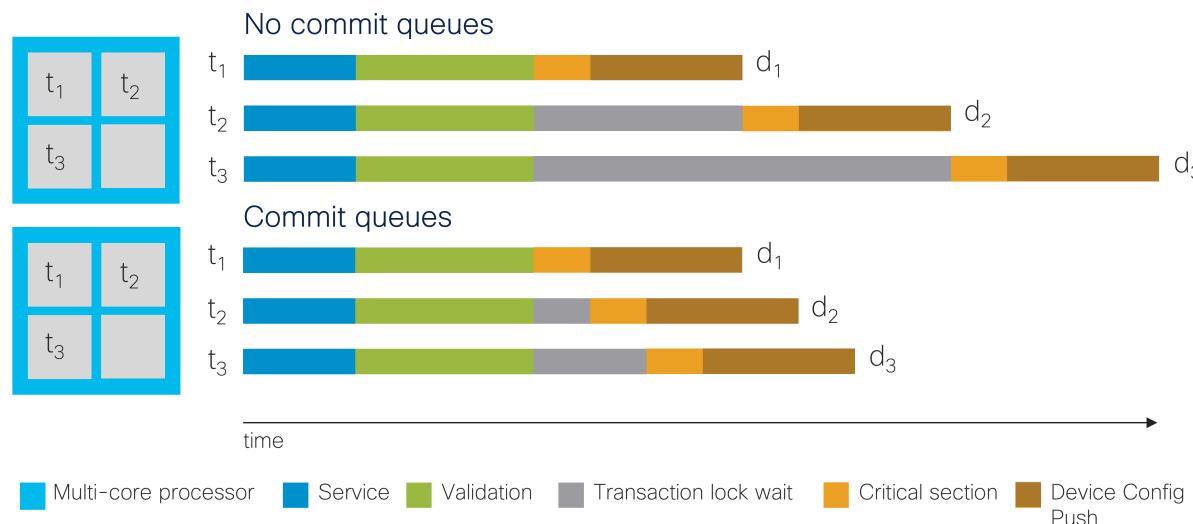
Note how the service creation and validation work now is divided into 1s per transaction and runs concurrently on one CPU core each. However, the two transactions cannot push the configuration concurrently to a device each as the config push is done inside the critical section, making one of the transactions wait for the other to release the transaction lock. See the two “holding the transaction lock” events in the above progress trace visualization.

To enable transactions to push configuration to devices concurrently, we must enable *commit queues*.

Using Commit Queues

The concept of a network-wide transaction requires NSO to wait for the managed devices to process the configuration change before exiting the critical section, i.e., before NSO can release the transaction lock. In the meantime, other transactions have to wait their turn to write to CDB and the devices. The commit queue feature avoids waiting for configuration to be written to the device and increases the throughput. For most use cases, commit queues improve transaction throughput significantly.

Writing to a commit queue instead of the device moves the device configuration push outside of the critical region, and the transaction lock can instead be released when the change has been written to the commit queue.



For commit queue documentation, see the section called “Commit Queue” in *User Guide*.

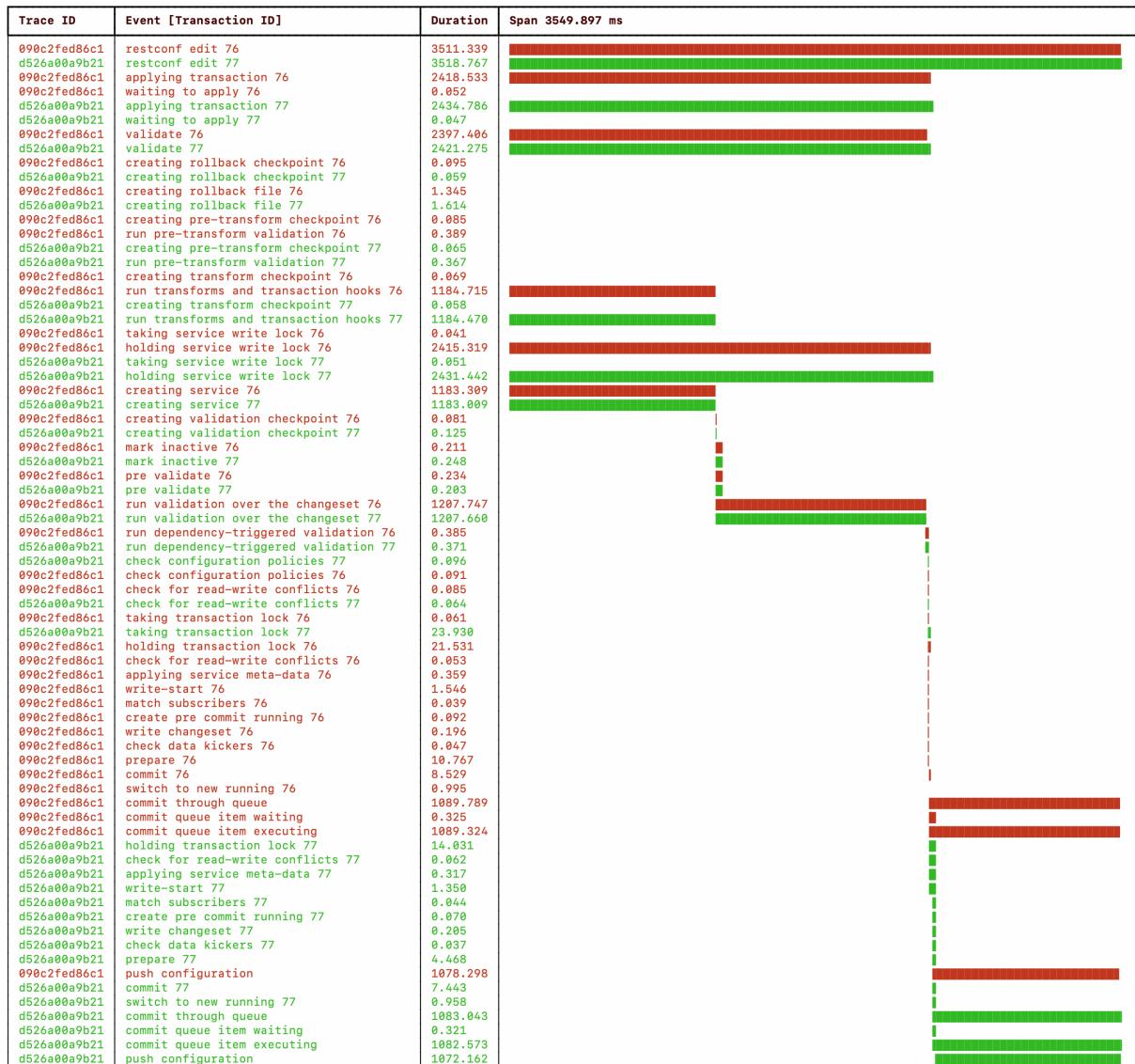
Enabling Commit Queues for the perf-trans Example

Enabling commit queues allows the two transactions to spread the create, validation, and configuration push to devices work across CPU cores in a multi-core processor. Only the CDB write and commit queue write now remain inside the critical section, and the transaction lock is released as soon as the device configuration changes have been written to the commit queues instead of waiting for the config push to the devices to complete. To run the `perf-trans` example with the work divided into one transaction per device and commit queues enabled:

```
make stop clean NDEVS=2 python
python3 measure.py --ntrans 2 --nwork 1 --ndtrans 1 --cqp param sync --ddelay 1
python3 ../common/simple_progress_trace_viewer.py $(ls logs/*.csv)
```

The resulting NSO progress trace:

Enabling Commit Queues for the perf-trans Example



A sequence diagram with transactions t1 and t2 deploying service configuration to two devices using RESTCONF patch requests to NSO with NSO configuring the netsim devices using NETCONF:

```

RESTCONF    service    validate    push config
patch        create     config      ndtrans=1           netsim
ntrans=2     nwork=1   nwork=1   cqparam=sync       device   ddelay=1
          t1 -----> ls -----> ls -----[---]----> ex0 -----> ls
          t2 -----> ls -----> ls -----[---]----> ex1 -----> ls
          wall-clock ls           ls                               1s = 3s
  
```

Note how the two transactions now push the configuration concurrently to a device each as the config push is done outside of the critical section. See the two “push configuration” events in the above progress trace visualization.

Stop NSO and the netsim devices:

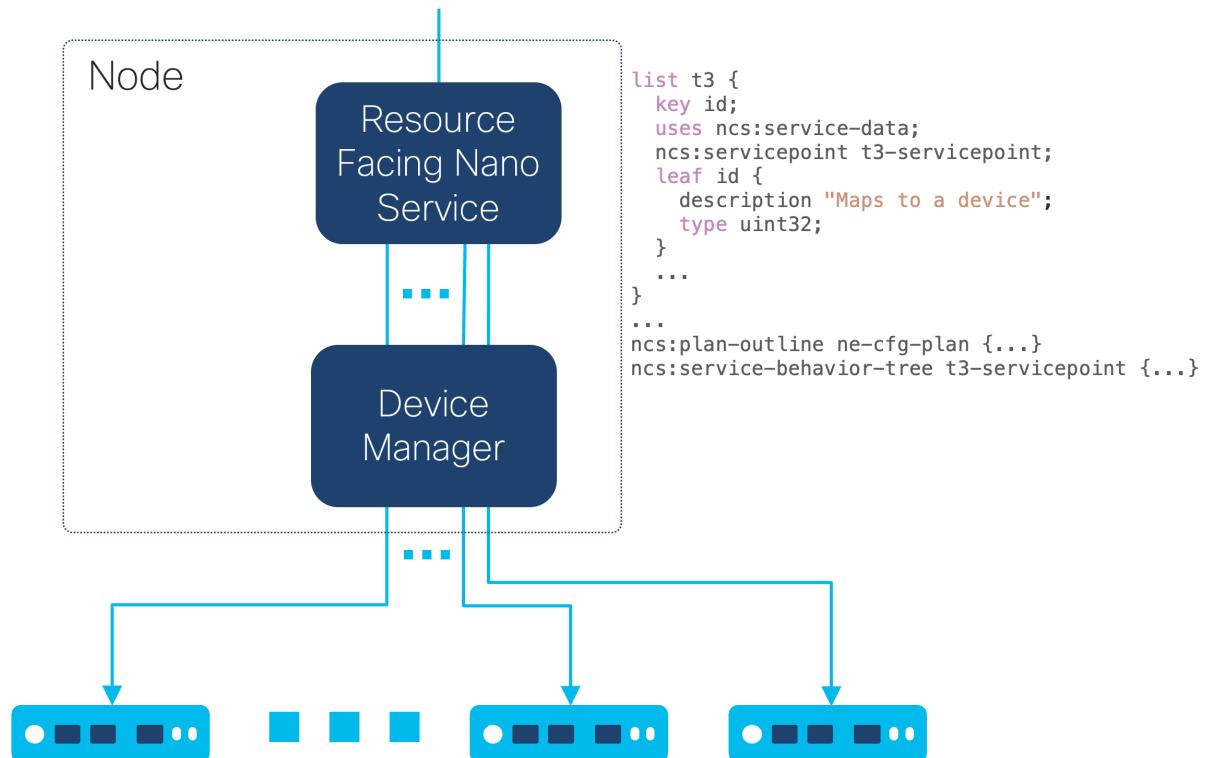
```
make stop
```

Running the `perf-setvals` example with two devices and commit queues enabled will produce a similar result.

Simplify the Per-Device Concurrent Transaction Creation Using a Nano Service

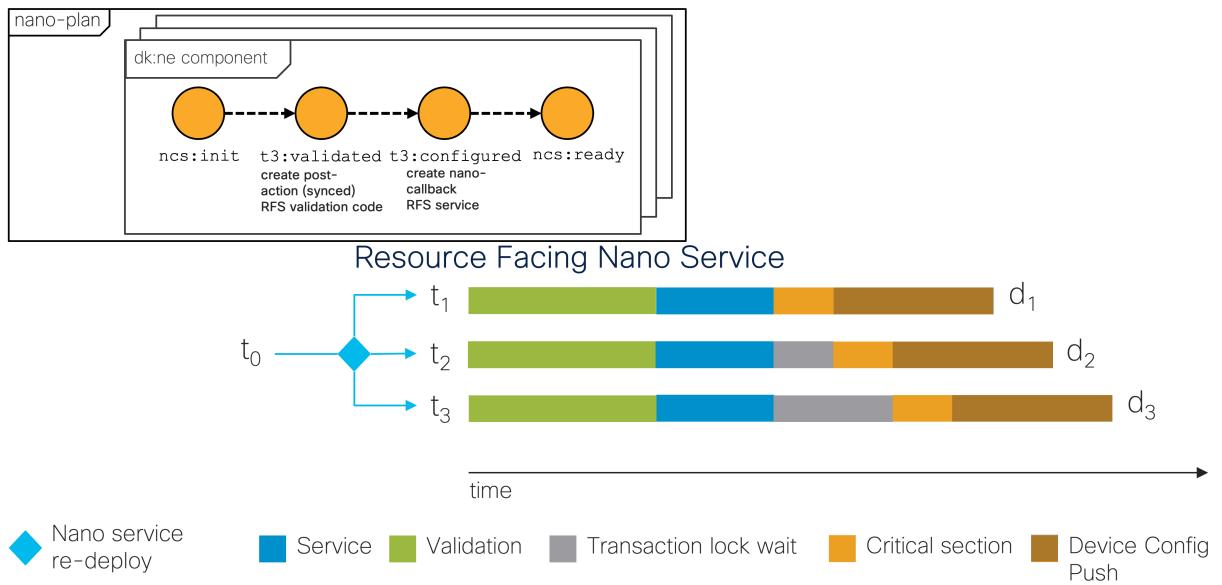
The `perf-trans` example service uses one transaction per service instance where each service instance configures one device. This enables transactions to run concurrently on separate CPU cores in a multi-core processor. The example sends RESTCONF patch requests concurrently to start transactions that run concurrently with the NSO transaction manager. However, dividing the work into multiple processes may not be practical for some applications using the NSO northbound interfaces, e.g., CLI or RESTCONF. Also, it makes a future migration to LSA more complex.

To simplify the NSO manager application, a resource-facing nano service (RFS) can start a process per service instance. The NSO manager application or user can then use a single transaction, e.g., CLI or RESTCONF, to configure multiple service instances where the NSO nano service divides the service instances into transactions running concurrently in separate processes.



The nano service can be straightforward, for example, using a single `t3:configured` state to invoke a service template or a `create()` callback. If validation code is required, it can run in a nano service post-action, `t3:validated` state, instead of a validation point callback to keep the validation code in the process created by the nano service.

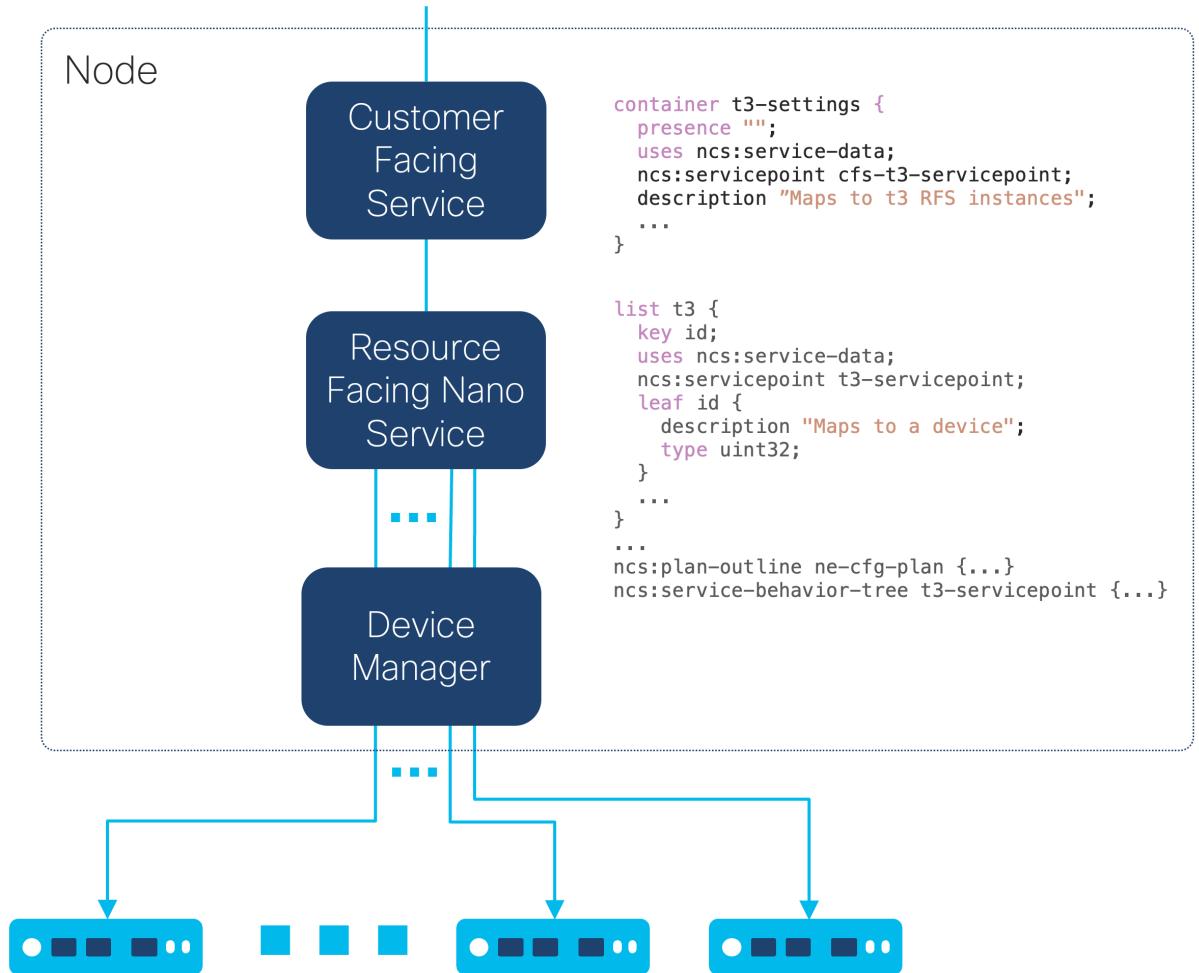
Simplify Using a CFS



See [Chapter 31, Nano Services for Staged Provisioning](#) and Chapter 5, *Developing and Deploying a Nano Service* in *Getting Started* for nano service documentation.

Simplify Using a CFS

A Customer Facing Service (CFS) that is stacked with the RFS and maps to one RFS instance per device can simplify the service that is exposed to the NSO northbound interfaces so that a single NSO northbound interface transaction spawns multiple transactions, for example, one transaction per RFS instance when using the converge-on-re-deploy YANG extension with the nano service behavior tree.



Running the CFS and Nano Service Enabled perf-stack Example

The `perf-stack` example showcases how a CFS on top of a simple resource-facing nano service can be implemented with the `perf-trans` example by modifying the existing t3 RFS and adding a CFS. Instead of multiple RESTCONF transactions, the example uses a single CLI CFS service commit that updates the desired number of service instances. The commit configures multiple service instances in a single transaction where the nano service runs each service instance in a separate process to allow multiple cores to be used concurrently.

Running the CFS and Nano Service Enabled perf-stack Example

NSO node



Run as below to start two transactions with a 1-second CPU time workload per transaction in both the service and validation callbacks, each transaction pushing the device configuration to one device, each using a synchronous commit queue, where each device simulates taking 1 second to make the configuration changes to the device:

```
cd $NCS_DIR/examples.ncs/development-guide/concurrency-model/perf-stack
./showcase.sh -d 2 -t 2 -w 1 -r 1 -q 'True' -y 1
```

Trace ID	Event [transaction ID]	Duration	Span 3521.195 ms
a86a0e077fd7	running action 97	1.168.554	
6e7ff6d7decb	write-start 87	1.930	
6e7ff6d7decb	match subscribers 87	0.046	
6e7ff6d7decb	create pre commit running 87	0.132	
6e7ff6d7decb	write changeset 87	0.129	
6e7ff6d7decb	check data kickers 87	0.134	
6e7ff6d7decb	prepare 87	0.869	
6e7ff6d7decb	commit 87	3.587	
6e7ff6d7decb	switch to new running 87	0.676	
6e7ff6d7decb	invoking data kickers 87	0.389	
f752c1df9455	running action 99	1.161.190	
19fa28cd81c7	re-deploy 108	2267.560	
a86a0e077fd7	match subscribers 97	0.031	
a86a0e077fd7	create pre commit running 97	0.124	
19fa28cd81c7	applying transaction 108	18.730	
19fa28cd81c7	waiting to apply 108	0.067	
a86a0e077fd7	write changeset 97	0.065	
19fa28cd81c7	validate 108	4.394	
19fa28cd81c7	creating rollback checkpoint 108	0.088	
6e7ff6d7decb	applying transaction 109	1169.524	
6e7ff6d7decb	modifying service 109	1158.344	
19fa28cd81c7	check configuration policies 108	0.043	
19fa28cd81c7	check for read-write conflicts 108	0.127	
19fa28cd81c7	taking transaction lock 108	0.057	
19fa28cd81c7	holding transaction lock 108	6.428	
19fa28cd81c7	check for read-write conflicts 108	0.098	
19fa28cd81c7	write-start 108	1.596	
d86515837f0f	running action 112	0.160	
d86515837f0f	re-deploy 112	2269.721	
19fa28cd81c7	match subscribers 108	0.066	
19fa28cd81c7	create pre commit running 108	0.131	
19fa28cd81c7	write changeset 108	0.078	
19fa28cd81c7	check data kickers 108	0.284	
d86515837f0f	applying transaction 112	8.897	
19fa28cd81c7	prepare 108	0.210	
d86515837f0f	waiting to apply 112	0.063	
d86515837f0f	validate 112	4.720	
d86515837f0f	creating rollback checkpoint 112	0.083	
6e7ff6d7decb	applying transaction 113	1177.500	
6e7ff6d7decb	modifying service 113	1148.828	
6e7ff6d7decb	commit through queue	1896.474	
6e7ff6d7decb	commit queue item waiting	0.223	
6e7ff6d7decb	commit queue item executing	1096.129	
6e7ff6d7decb	holding transaction lock 113	14.892	
6e7ff6d7decb	check for read-write conflicts 113	0.111	
6e7ff6d7decb	applying service meta-data 113	0.516	
6e7ff6d7decb	write-start 113	1.847	
6e7ff6d7decb	match subscribers 113	0.051	
6e7ff6d7decb	create pre commit running 113	0.234	
6e7ff6d7decb	write changeset 113	0.201	
6e7ff6d7decb	check data kickers 113	0.039	
6e7ff6d7decb	prepare 113	2.865	
6e7ff6d7decb	push configuration	1883.937	
6e7ff6d7decb	commit 113	8.264	
6e7ff6d7decb	switch to new running 113	0.832	
6e7ff6d7decb	commit through queue	1690.734	
6e7ff6d7decb	commit queue item waiting	0.217	
6e7ff6d7decb	commit queue item executing	1090.378	
6e7ff6d7decb	push configuration	1077.231	

The above progress trace visualization is truncated to fit, but notice how the t3:validated state action callbacks, t3:configured state service creation callbacks, and configuration push from the commit queues are running concurrently (on separate CPU cores) when initiating the service deployment with a single transaction started by the CLI commit.

A sequence diagram describing the transaction t1 deploying service configuration to the devices using the NSO CLI:

```

      CFS           validate   service   push config      config
      create       Nano config   create    ndtrans=1   netsim subscriber
CLI     trans=2   RFS   nwork=1   nwork=1   cq=True    device ddelay=1
commit          t1 --> 1s -----> 1s -----[---]---> ex0 ---> 1s
                  t -----> t ---->
                  t2 --> 1s -----> 1s -----[---]---> ex1 ---> 1s
                  wall-clock 1s           1s                         1s=3s

```

The two transactions run concurrently, deploying the service in ~3 seconds (plus some overhead) of wall-clock time. Like the `perf-trans` example, you can play around with the `perf-stack` example by tweaking the parameters.

```

-d NDEVS
The number of netsim (ConfD) devices (network elements) started.
Default 4

-t NTRANS
The number of transactions updating the same service in parallel.
Default: $NDEVS

-w NWORK
Work per transaction in the service creation and validation phases. One
second of CPU time per work item.
Default: 3 seconds of CPU time.

-r NDTRANS
Number of devices the service will configure per service transaction.
Default: 1

-c USECQ
Use device commit queues.
Default: True

-y DEV_DELAY
Transaction delay (simulated by sleeping) on the netsim devices (seconds).
Default: 1 second

```

See the README in the `perf-stack` example for details. For even more details, see the steps in the showcase script.

Stop NSO and the netsim devices:

```
make stop
```

Migrating to and Scale Up Using an LSA Setup

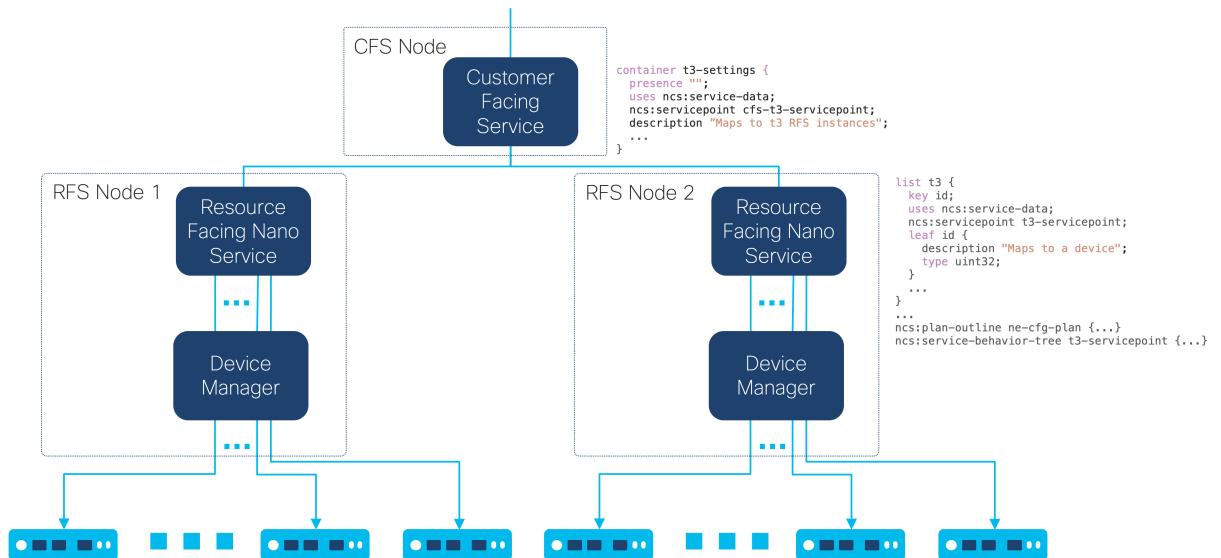
If the processor where NSO runs becomes a severe bottleneck, the CFS can migrate to a layered service architecture (LSA) setup. The `perf-stack` example implements stacked services, a CFS abstracting the RFS. It allows for easy migration to an LSA setup to scale with the number of devices or network elements participating in the service deployment. While adding complexity, LSA allows exposing a single CFS instance for all processors instead of one per processor.

**Note**

Before considering taking on the complexity of a multi-NSO node LSA setup, make sure you have done the following:

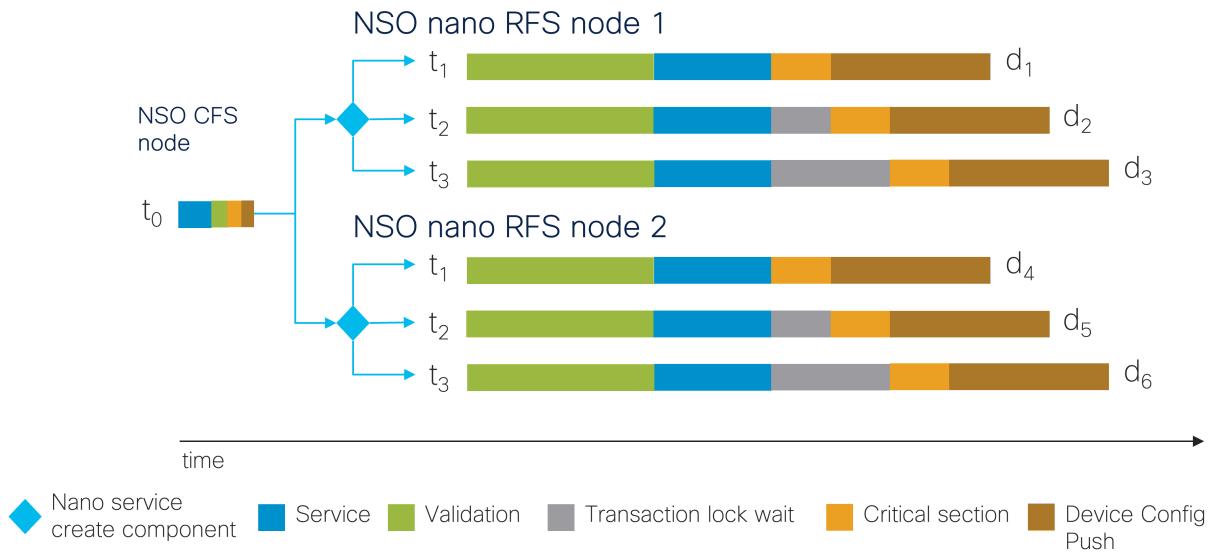
- Explored all possible avenues of design and optimization improvements described so far in this section.
- Measured the transaction performance to find bottlenecks.
- Optimized any bottlenecks to reduce their overhead as much as possible.
- Observe that the available processor cores are all fully utilized.
- Explored running NSO on a more powerful processor with more CPU cores and faster clock speed.
- If there are more devices and RFS instances created at one point than available CPU cores, verify that increasing the number of CPU cores will result in a significant improvement. I.e., if the CPU processing spent on service creation and validation is substantial, the bottleneck, compared to writing the configuration to CDB and the commit queues and pushing the configuration to the devices.

Migrating to an LSA setup should only be considered after checking all boxes for the above items.



Running the LSA-enabled perf-lsa Example

The `perf-lsa` example builds on the `perf-stack` example and showcases an LSA setup using two RFS NSO instances, `lower-nso-1` and `lower-nso-2`, with a CFS NSO instance, `upper-nso`.



You can imagine adding more RFS NSO instances, lower-nso-3, lower-nso-4, etc., to the existing two as the number of devices increases. One NSO instance per multi-core processor and at least one CPU core per device (network element) is likely the most performant setup for this simulated work example. See Chapter 1, *LSA Overview* in *Layered Service Architecture* for more.

As an example, a variant that starts four RFS transactions with a 1-second CPU time workload per transaction in both the service and validation callbacks, each RFS transaction pushing the device configuration to 1 device using synchronous commit queues, where each device simulates taking 1 second to make the configuration changes to the device:

```
cd $NCS_DIR/examples.ncs/development-guide/concurrency-model/perf-lsa
./showcase.sh -d 2 -t 2 -w 1 -r 1 -q 'True' -y 1
```

The three NSO progress trace visualizations show NSO on the CFS and the two RFS nodes. Notice how the CLI commit starts a transaction on the CFS node and configures four service instances with two transactions on each RFS node to push the resulting configuration to four devices.

Figure 194. NSO CFS node

Trace ID	Event [Transaction ID]	Duration	Span 3435.883 ms
47a2ca4122ab	applying transaction 90	154.853	
47a2ca4122ab	waiting to apply 90	0.026	
47a2ca4122ab	validate 90	27.182	
47a2ca4122ab	creating rollback checkpoint 90	0.106	
47a2ca4122ab	creating rollback file 90	2.035	
47a2ca4122ab	creating pre-transform checkpoint 90	0.242	
47a2ca4122ab	run pre-transform validation 90	0.387	
47a2ca4122ab	creating transform checkpoint 90	0.101	
47a2ca4122ab	run transforms and transaction hooks 90	19.633	
47a2ca4122ab	taking service write lock 90	0.068	
47a2ca4122ab	holding service write lock 90	149.624	
47a2ca4122ab	creating service 90	17.737	
47a2ca4122ab	creating validation checkpoint 90	0.137	
47a2ca4122ab	mark inactive 90	0.296	
47a2ca4122ab	pre validate 90	0.289	
47a2ca4122ab	run validation over the changeset 90	0.271	
47a2ca4122ab	run dependency-triggered validation 90	0.262	
47a2ca4122ab	check configuration policies 90	0.057	
47a2ca4122ab	check for read-write conflicts 90	0.154	
47a2ca4122ab	taking transaction lock 90	0.062	
47a2ca4122ab	holding transaction lock 90	128.793	
47a2ca4122ab	check for read-write conflicts 90	0.146	
47a2ca4122ab	applying service meta-data 90	0.867	
47a2ca4122ab	write-start 90	2.895	
47a2ca4122ab	match subscribers 90	0.142	
47a2ca4122ab	create pre commit running 90	0.232	
47a2ca4122ab	write changeset 90	0.496	
47a2ca4122ab	check data kickers 90	0.054	
47a2ca4122ab	prepare 90	103.009	
47a2ca4122ab	push configuration 90	94.890	
47a2ca4122ab	push configuration 90	94.812	
47a2ca4122ab	commit 90	21.466	
47a2ca4122ab	switch to new running 90	1.973	

Running the LSA-enabled perf-lsa Example

Figure 195. NSO RFS node 1 (truncated to fit)

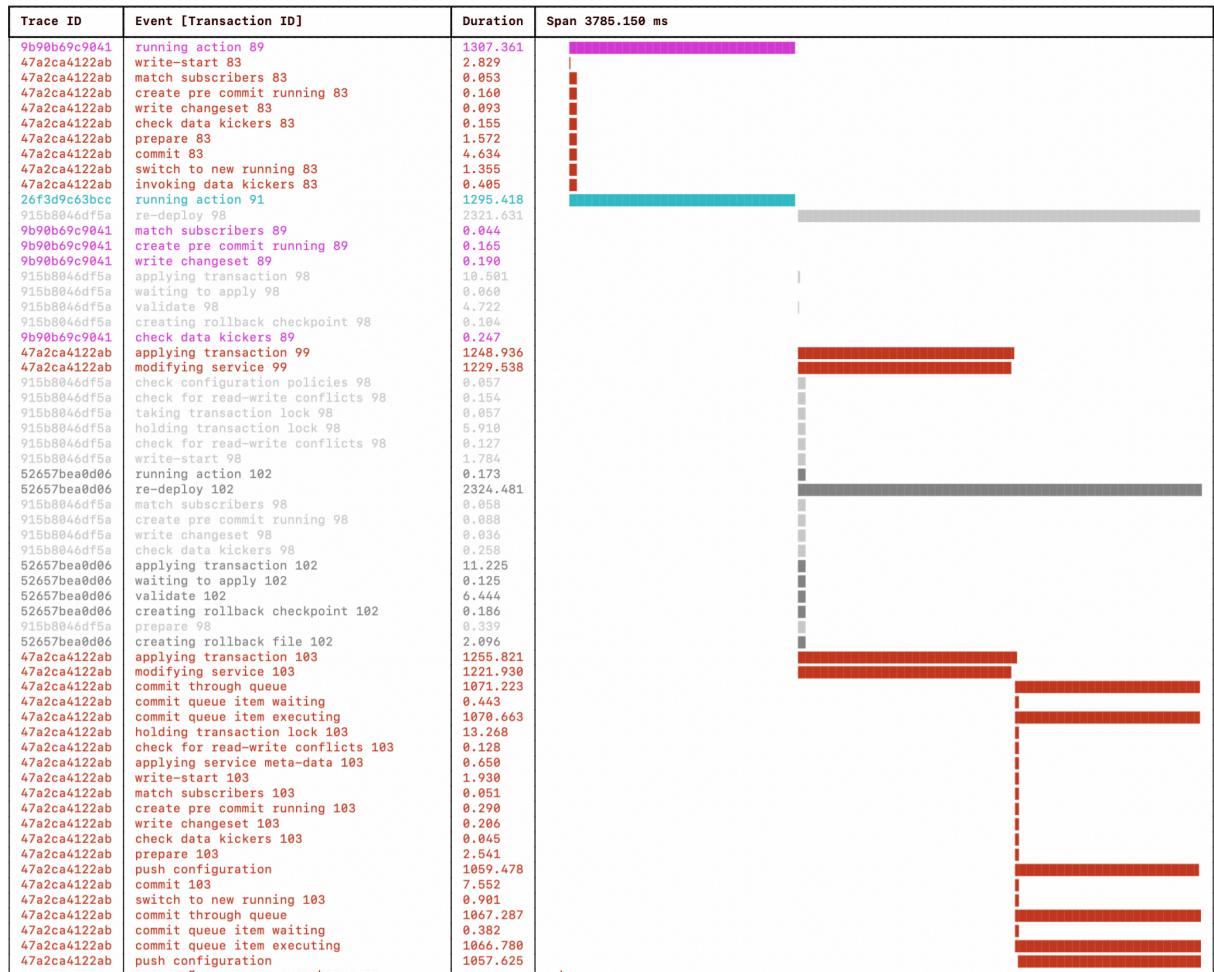
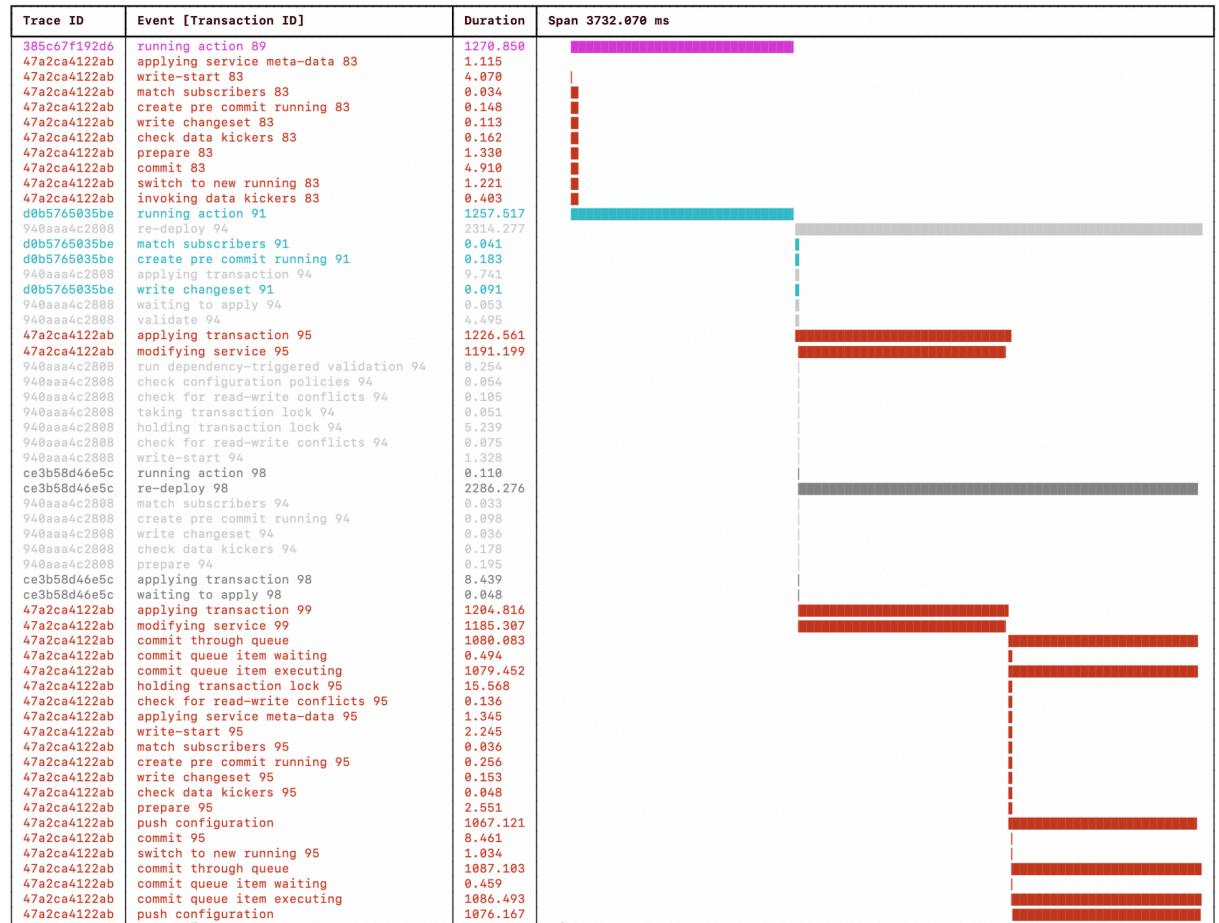
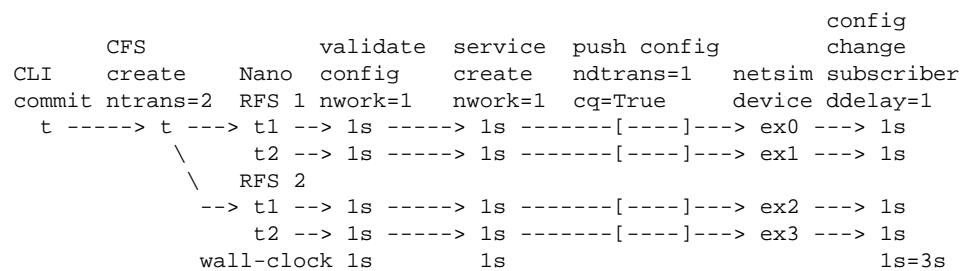


Figure 196. NSO RFS node 2 (truncated to fit)



A sequence diagram describing the transactions on RFS 1 t1 t2 and RFS 2 t1 t2. The transactions deploy service configuration to the devices using the NSO CLI:



The four transactions run concurrently, two per RFS node, performing the work and configuring the four devices in ~3 seconds (plus some overhead) of wall-clock time.

You can play with the perf-lsa example by tweaking the parameters.

```

-d LDEVS
Number of netsim (ConfD) devices (network elements) started per RFS
NSO instance.
Default 2 (4 total)

-t NTRANS

```

Number of transactions updating the same service in parallel per RFS NSO instance. Here, one per device.
 Default: \$LDEVS (\$LDEVS * 2 total)

-w NWORK
 Work per transaction in the service creation and validation phases. One second of CPU time per work item.
 Default: 3 seconds of CPU time.

-r NDTRANS
 Number of devices the service will configure per service transaction.
 Default: 1

-q USECQ
 Use device commit queues.
 Default: True

-y DEV_DELAY
 Transaction delay (simulated by sleeping) on the netsim devices (seconds).
 Default: 1 second

See the README in the `perf-lsa` example for details. For even more details, see the steps in the showcase script.

Stop NSO and the netsim devices:

```
make stop
```

Scaling RAM and Disk

NSO contains an internal database called CDB, which stores both configuration and operational state data. Understanding the resource consumption of NSO at a steady state is mostly about understanding CDB, as it typically stands for the vast majority of resource usage.

CDB

Optimized for fast access, CDB is an in-memory database that holds all data in RAM. It also keeps the data on disk for persistence. The in-memory data structure is optimized for navigating tree data but is still a compact and efficient memory structure. The on-disk format uses a log structure, making it fast to write and very compact.

The in-memory structure usually consumes 2 - 3x more than the size of the on-disk format. The on-disk log will grow as more changes are performed in the system. A periodic compaction process compacts the write log and reduces its size. Upon startup of NSO, the on-disk version of CDB will be read, and the in-memory structure will be recreated based on the log. A recently compacted CDB will thus start up faster.

By default, NSO automatically determines when to compact CDB. It is visible in the `devel.log` when CDB compaction takes place. Compaction may require significant time, during which write transactions cannot be performed. In certain use cases, it may be preferable to disable automatic compaction by CDB and instead trigger compaction manually according to the specific needs. See the section called “Compaction” in *Administration Guide* for more details.

Services and Devices in CDB

CDB is a YANG-modeled database. By writing a YANG model, it is possible to store any kind of data in NSO and access it via one of the northbound interfaces of NSO. From this perspective, a service or a device's configuration is like most other YANG-modeled data. The number of service instances in NSO in the steady state affects how much space the data consumes in RAM and on disk.

But keep in mind that services tend to be modified from time to time, and with a higher total number of service instances, changes to those services are more likely. A higher number of service instances means more transactions to deploy changes, which means an increased need for optimizing transactional throughput, available CPU processing, RAM, and disk. See the section called “[Designing for Maximal Transaction Throughput](#)” for details.

CDB Stores the YANG Model Schema

In addition to storing instance data, CDB also stores the schema (the YANG models), on disk and reads it into memory on startup. Having a large schema (many or large YANG models) loaded means both disk and RAM will be used, even when starting up an “empty” NSO, i.e., no instance data is stored in CDB.

In particular, device YANG models can be of considerable size. For example, the YANG models in recent versions of Cisco IOS XR have over 750,000 lines. Loading one such NED will consume about 1GB GB of RAM and slightly less disk space. In a mixed vendor network, you would load NEDs for all or some of these device types. With CDM, you can have multiple XR NEDs loaded to support communicating with different versions of XR and similarly for other devices, further consuming resources.

In comparison, most CLI NEDs only model a subset of a device and, are as a result, much smaller, most often under 100,000 lines of YANG.

For small NSO systems, the schema will usually consume more resources than the instance data, and NEDs, in particular, are the most significant contributors to resource consumption. As the system grows and more service and device configuration is added, the percentage of the total resource usage used for NED YANG models will decrease.

Note that the schema is memory mapped into shared memory, so even though multiple Python VMs might be started, memory usage will not increase as it shares memory between different clients. The Java VM uses its own copy of the schema, which is also why we can see that the JVM memory consumption follows the size of the loaded YANG schema.

The Size of CDB

Accurately predicting the size of CDB means accurately modeling its internal data structure. Since the result will depend on the YANG models and what actual values are stored in the database, the easiest way to understand of how the size grows is to start NSO with the schema and data in question and then measure the resource usage.

Performing accurate measurements can be a tedious process or sometimes impossible. When impossible, an estimate can be reached by extrapolating from known data, which is usually much more manageable and accurate enough.

We can look at the disk and RAM used for the running datastore, which stores configuration. On a freshly started NSO, it doesn't occupy much space at all:

```
# show ncs-state internal cdb datastore running | select ram-size | select disk-size
      DISK
NAME      SIZE      RAM SIZE
-----
running   3.83 KiB  26.27 KiB
```

Devices, Small and Large

Adding a device with a small configuration, in this case, a Cisco NXOS switch with about 700 lines of CLI configuration, there is a clear increase:

```
# show ncs-state internal cdb datastore running | select ram-size | select disk-size
NAME      DISK SIZE   RAM SIZE
-----
running  28.51 KiB  240.99 KiB
```

Compared to the size of CDB before we added the device, we can deduce that the device with its configuration takes up ~214 kB in RAM and 25 kB on disk. Adding 1000 such devices, we see how CDB resource consumption increases linearly with more devices. This graph shows the RAM and memory usage of the running datastore in CDB over time. We perform a sequential **sync-from** operation on the 1000 devices, and while it is executing, we see how resource consumption increases. At the end, resource consumption has reached about 150 MB of RAM and 25 MB of disk, equating to ~150 KiB of RAM and ~25 KiB of disk per device.

```
# request devices device * sync-from
```


Note

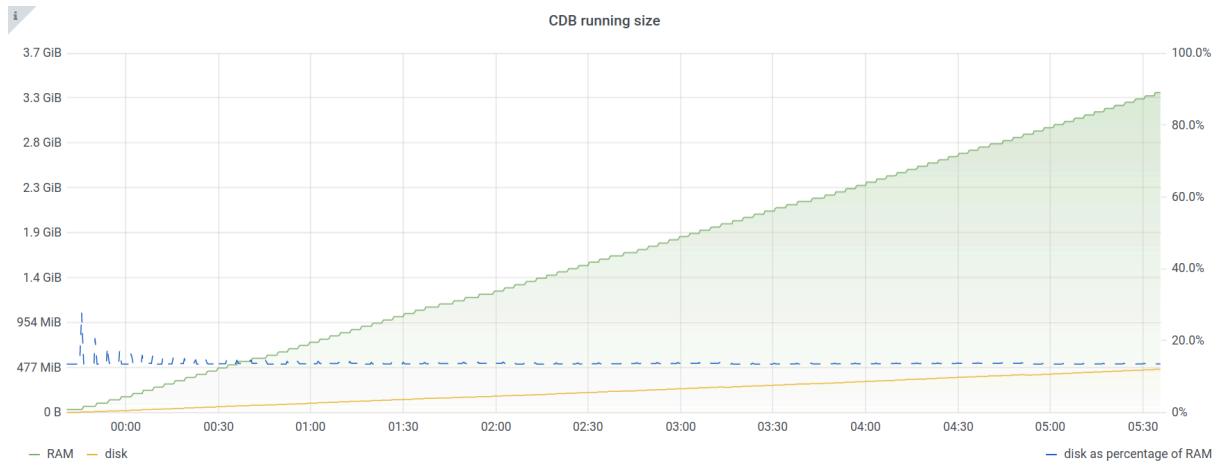
The wildcard expansion in **request devices device * sync-from** is processed by the CLI, which will iterate over the devices sequentially. This is inefficient and can be sped up by using **devices sync-from** which instead processes the devices concurrently. The sequential mode better produces a graph that better illustrates how this scales, which is why it is used here.



A device with a larger configuration will consume more space. With a single Juniper MX device that has a configuration with close to half a million lines of configuration, there's a substantial increase:

```
# show ncs-state internal cdb datastore running | select ram-size | select disk-size
NAME      DISK SIZE   RAM SIZE
-----
running  4.59 MiB  33.97 MiB
```

Similarly, adding more such devices allows monitoring of how it scales linearly. In the end, with 100 devices, CDB consumes 3.35 GB of RAM and 450 MB of disk, or ~33.5 MiB of RAM and ~4.5 MiB disk space per device.



Thus, you must do more than dimension your NSO installation based on the number of devices. You must also understand roughly how much resources each device will consume.

Unless a device uses NETCONF, NSO will not store the configuration as retrieved from the device. When configuration is retrieved, it is parsed by the NED into a structured format.

For example, here is a basic BGP stanza from a Cisco IOS device:

```
router bgp 64512
address-family ipv4 vrf TEST
no synchronization
redistribute connected metric 123 route-map IPV4-REDISTRIBUTE-CONNECTED-TO-BGP
!
```

After being parsed by the IOS CLI NED, the equivalent configuration looks like this in NSO:

```
<router xmlns="urn:ios">
  <bgp>
    <as-no>64512</as-no>
    <address-family>
      <with-vrf>
        <ipv4>
          <af>unicast</af>
          <vrf>
            <name>TEST</name>
            <redistribute>
              <connected>
                <metric>123</metric>
                <route-map>IPV4-REDISTRIBUTE-CONNECTED-TO-BGP</route-map>
              </connected>
              <static/>
            </redistribute>
            </vrf>
          </ipv4>
          </with-vrf>
        </address-family>
      </bgp>
    </router>
```

A single line, such as **redistribute connected metric 123 route-map IPV4-REDISTRIBUTE-CONNECTED-TO-BGP**, is parsed into a structure of multiple nodes / YANG leaves. There is no exact correlation between number of lines of configuration with the space it consumes in NSO. The easiest way to determine the resource consumption of a device's configuration is thus to load it into NSO and check the size of CDB before and after.

Planning Resource Consumption

Forming a rough estimate of CDB resource consumption for planning can be helpful.

Divide your devices into categories. Get a rough measurement for an exemplar in each category, add a safety margin, e.g., double the resource consumption, and multiply by the number of devices in that category.

Table 197. Example

Device Type	RAM	Disk	Number of Devices	Margin	Total RAM	Total Disk
FTTB access switch	200KiB	25KiB	30000	100%	11718MiB	1464MiB
Mobile Base Station	120KiB	11KiB	15000	100%	3515MiB	322MiB
Business CPE	50KiB	4KiB	50000	50%	3662MiB	292MiB
PE / Edge Router	10MiB	1MiB	1000	25%	12GiB	1.2GiB
Total					20.6GiB	3.3GiB

The Size of a Service

A YANG model describes the input to services, and just like any other data in CDB, it consumes resources. Compared to the typical device configuration, where even small devices often have a few hundred lines of configuration, a small service might only have a handful of configurable inputs. Even extensive services rarely have more than 50 inputs.

When services write configuration, a reverse diff set is generated and saved as part of the service's private data. The more configuration a service writes, the larger its reverse diff set will be and, thus, the more resources it will consume. What appears as a small service with just a handful of inputs could consume considerable resources if it writes a lot of configuration. Similarly, we save a forward diff set by default, contributing to the size. Service metadata attributes, the back pointer list, and the recount are also added to the written configuration, which consumes some resources. For example, if 50 services all (share)create a node, there will be 50 backpointers in the database, which consumes some space.

Implications of a Large CDB

As shown above, CDB scales linearly. Modern servers commonly support multiple terabytes of RAM, making it possible to support 50,000 - 100,000 such large router devices in NSO, well beyond the size of any currently existing network. However, beyond consuming RAM and disk space, the size of the CDB also affects the startup time of NSO and certain other operations like upgrades. In the previous example, 100 devices were used, which resulted in a CDB size of 461 MB on disk. Starting that on a standard laptop takes about 100 seconds. With 50,000 devices, CDB on-disk would be over 230 GB, which would take around 6 hours to load on the same laptop, if it had enough RAM. The typical server is considerably faster than the average laptop here, but loading a large CDB will take considerable time.

This also affects the sync/resync time in high availability setups, where the database size increases the data transfer needed.

A working system needs more than just storing the data. It must also be possible to use the devices and services and apply the necessary operations to these for the environment in which they operate. For

example, it is common in brownfield environments to frequently run the **sync-from** action. Most device-related operations, including **sync-from**, can run concurrently across multiple devices in NSO. Syncing an extensive device configuration will take a few minutes or so. With 50,000 such large devices, we are looking at a total time of tens of hours or even days. Many environments require higher throughput, which could be handled using an LSA setup and spreading the devices over many NSO RFS nodes. **sync-from** is an example of an action that is easy to scale up and runs concurrently. For example, spreading the 50,000 devices over 5 NSO RFS nodes, each with 10,000 devices, would lead to a speedup close to 5x.

Using LSA, multiple Resource Facing Service (RFS) nodes can be employed to spread the devices across multiple NSO instances. This allows increasing the parallelism in sync-from and other operations, as described in [the section called “Designing for Maximal Transaction Throughput”](#), making it possible to scale to an almost arbitrary number of devices. Similarly, the services associated with each device are also spread across the RFS nodes, making it possible to operate on them in parallel. Finally, a top CFS node communicates with all RFS nodes, making it possible to administrate the entire setup as one extensive system.

Checklists

For smooth operation of NSO instances consider all of the following:

- Ensure there is enough RAM for NSO to run, with ample headroom.
- `create()` should normally run in a few hundred milliseconds, perhaps a few seconds for extensive services.
 - Consider splitting into smaller services.
 - Stacked services allow the composition of many smaller services into a larger service. A common best-practice design pattern is to have one Resource Facing Service (RFS) instance map to one device or network element.
 - Avoid conflicts between service instances.
 - Improves performance compared to a single large service for typical modifications.
 - Only services with changed input will have their `create()` called.
 - A small change to the Customer Facing Service (CFS) that results in changes to a subset of the lower services avoids running `create()` for all lower services.
 - No external calls or **sync-from** in `create()` code.
 - Use nano-services to do external calls asynchronously.
 - Never run **sync-from** from `create()` code.
 - Carefully consider the complexity of XPath constraints, in particular around lists.
 - Avoid XPath expressions with linear scaling or worse.
 - For example, avoid checking something for every element in a list, as performance will drop radically as the list grows.
 - XPath expressions involving nested lists or comparisons between lists can lead to quadratic scaling.
 - Make sure you have an efficient transaction ID method for NEDs.
 - In the worst case, the NED will compute the transaction ID based on a config hash, which means it will fetch the entire config to compute the transaction ID.
 - Enable commit-queues and ensure transactions utilize as many CPU cores in a multi-core system as possible to increase transactional throughput.
 - Ensure there are enough file descriptors available.
 - In many Linux systems the default limit is 1024.

- If we, for example, assume that there are 4 northbound interface ports, CLI, RESTCONF, SNMP, JSON-RPC, or similar, plus a few hundred IPC ports, $x 1024 = 5120$. But one might as well use the next power of two, 8192, to be on the safe side.
- See the section called “Disable Memory Overcommit” in *Getting Started*.

Hardware Sizing

Lab Testing and Development

While a minimal setup with a single CPU core and 1 GB of RAM is enough to start NSO for lab testing and development, it is recommended to have at least 2 CPU cores to avoid CPU contention and to run at least two transactions concurrently, and 4 GB of RAM to be able to load a few NEDs.

Contemporary laptops typically work well for NSO service development.

Production

For production systems it is recommended to have at least 8 CPU cores and with as high clock frequency as possible. This ensures all NSO processes can run without contending for the same CPU cores. More CPU cores enable more transactions to run in parallel on the same processor. For higher scale systems, an LSA setup should be investigated together with a technical expert. See [the section called “Designing for Maximal Transaction Throughput”](#).

NSO is not very disk intensive since CDB is loaded into RAM. On startup, CDB is read from disk into memory. Therefore, for fast startups of NSO, rapid backups, and other similar administrative operations, it is recommended to use a fast disk, for example, an NVMe SSD.

Network management protocols typically consume little network bandwidth. It is often less than 10 Mbps but can burst many times that. While 10 Gbps is recommended, 1 Gbps network connectivity will usually suffice. If you use High Availability (HA), the continuous HA updates are typically relatively small and do not consume a lot of bandwidth. A low latency, preferably below 1 ms and well within 10 ms, will significantly impact performance more than increasing bandwidth beyond 1 Gbps. 10 Gbps or more can make a difference for the initial synchronization in case the nodes are not in sync and avoid congestion when doing backups over the network or similar.

The in-memory portion of CDB needs to fit in RAM, and NSO needs working memory to process queries. This is a hard requirement. NSO can only function with enough memory. Less than the required amount of RAM does not lead to performance degradation - it prevents NSO from working. For example, if CDB consumes 50 GB, ensure you have at least 64 GB of RAM. There needs to be some headroom for RAM to allow temporary usage during, for example, heavy queries.

Swapping is a way to use disk space as RAM, and while it can make it possible to start an NSO instance that otherwise would not fit in RAM, it would lead to terrible performance. See the section called “Disable Memory Overcommit” in *Getting Started*.

Provide at least 32GB of RAM and increase with the growth of CDB. As described in [the section called “Scaling RAM and Disk”](#), the consumption of memory and disk resources for devices and services will vary greatly with the type and size of the service or device.



CHAPTER 24

NSO Concurrency Model

Since version 6.0, NSO uses so-called *optimistic concurrency*, which greatly improves parallelism. With this approach, NSO avoids the need for serialization and a global lock to run user code which would otherwise limit the number of requests the system can process in a given time unit.

Using this concurrency model, your code, such as a service mapping or custom validation code, can run in parallel, either with another instance of the same service or an entirely different service (or any other provisioning code, for that matter). As a result, the system can take better advantage of available resources, especially the additional CPU cores, making it a lot more performant.

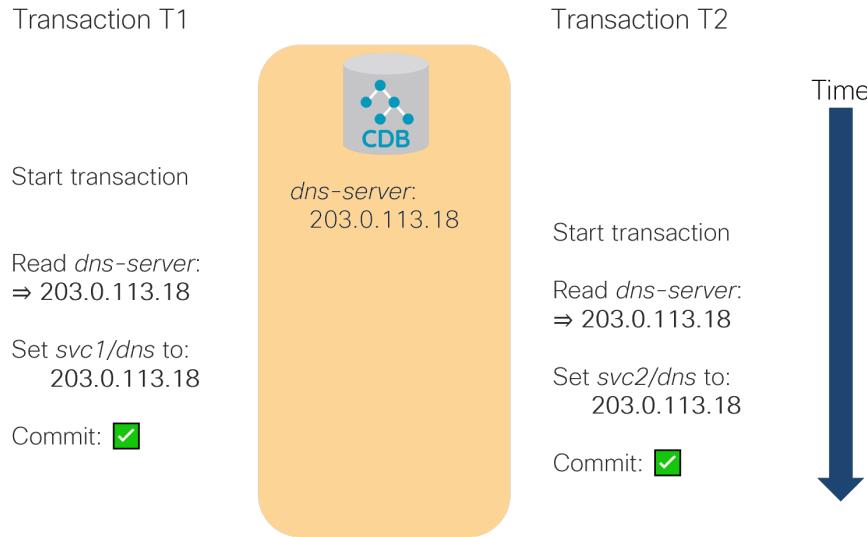
- [Optimistic Concurrency, page 495](#)
- [Identifying Conflicts, page 498](#)
- [Automatic Retries, page 499](#)
- [Handling Conflicts, page 500](#)
- [Example Retrying Code in Python, page 501](#)
- [Example Retrying Code in Java, page 502](#)
- [Designing for Concurrency, page 504](#)

Optimistic Concurrency

Transactional systems, such as NSO, must process each request in a way that preserves what are known as the *ACID properties*, such as atomicity and isolation of requests. A traditional approach to ensure this behavior is by using locking to apply requests or transactions one by one. The main downside is that requests are processed sequentially and may not be able to fully utilize the available resources.

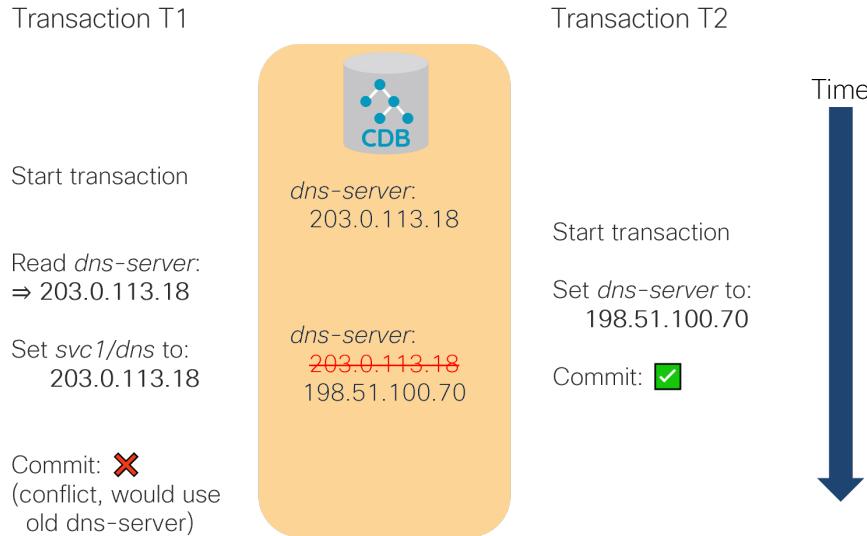
Optimistic concurrency, on the other hand, allows transactions to run in parallel. It works on the premise that data conflicts are rare, so most of the time the transactions can be applied concurrently and will retain the required properties. NSO ensures this by checking that there are no conflicts with other transactions just before each transaction is committed. In particular, NSO will verify that all the data accessed as part of the transaction is still valid when applying changes. Otherwise, the system will reject the transaction.

Such model makes sense because a lot of the time concurrent transactions deal with separate sets of data. Even if multiple transactions share some data in a read-only fashion, it is fine as they still produce the same result.

Figure 198. Nonconflicting Concurrent Transactions

In the figure, *svc1* in the T1 transaction and *svc2* in the T2 transaction both read (but do not change) the same, shared piece of data and can proceed as usual, unperturbed.

On the other hand, a conflict is when a piece of data, that has been read by one transaction, is changed by another transaction before the first transaction is committed. In this case, at the moment the first transaction completes, it is already working with stale data and must be rejected, as the following figure shows.

Figure 199. Conflicting Concurrent Transactions

In the figure, transaction T1 reads *dns-server* to use in provisioning of *svc1* but transaction T2 changes *dns-server* value in the meantime. The two transactions conflict and T1 is rejected because T2 completed first.

To be precise, for a transaction to experience a conflict, both of the following has to be true:

- 1 It reads some data that is changed after being read and before the transaction is completed.
- 2 It commits a set of changes in NSO.

This means a set of read-only transactions or transactions, where nothing is really changed, will never conflict. It is also possible that multiple write-only transactions won't conflict even when they update the same data nodes.

Allowing multiple concurrent transactions to write (and only write, not read) to the same data without conflict may seem odd at first. But from a transaction's standpoint, it does not depend on the current value because it was never read. Suppose the value changed the previous day, the transaction would do the exact same thing and you wouldn't consider it a conflict. So, the last write wins, regardless of the time elapsed between the two transactions.

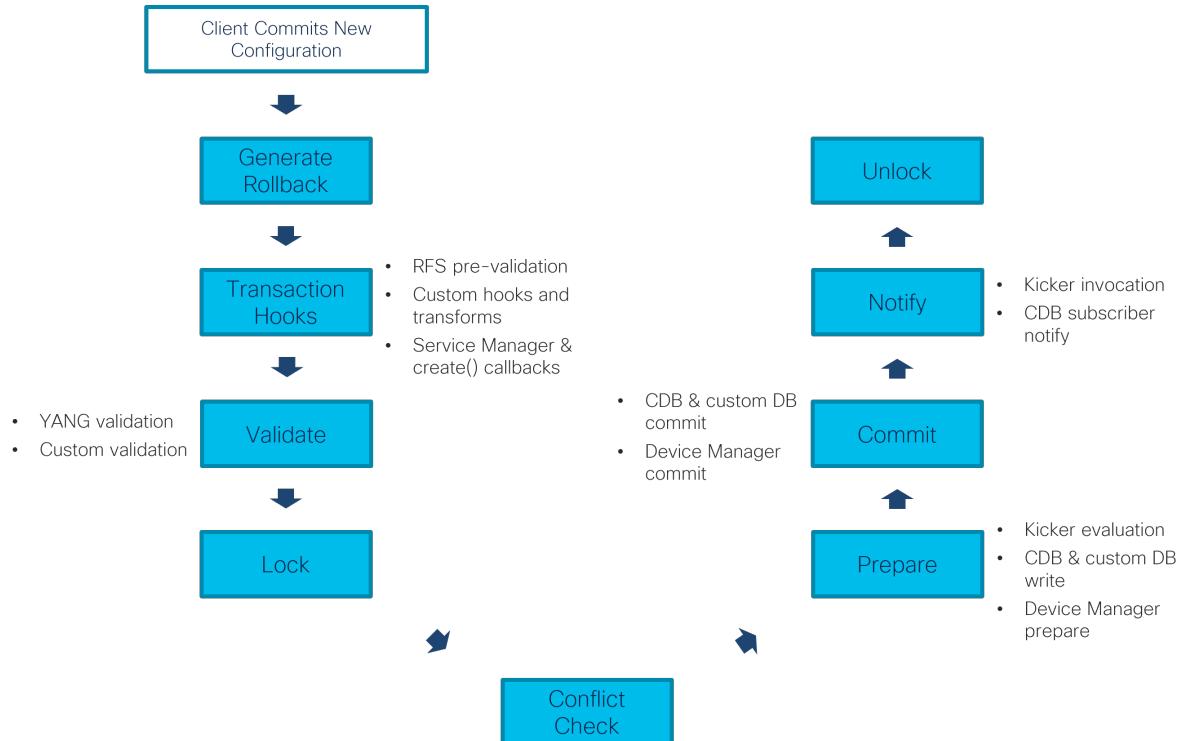


Caution

It is extremely important that you do not mix multiple transactions, because it will prevent NSO from detecting conflicts properly. For example, starting multiple separate transactions and using one to write data, based on what was read from a different one, can result in subtle bugs that are hard to troubleshoot.

While the optimistic concurrency model allows transactions to run concurrently most of the time, ultimately some synchronization (a global lock) is still required to perform the conflict checks and serialize data writes to the CDB and devices. The following figure shows everything that happens after a client tries to apply a configuration change, including acquiring and releasing the lock. This process takes place, for example, when you enter the **commit** command on the NSO CLI or when a PUT request of the RESTCONF API is processed.

Figure 200. Stages of a Transaction Commit



As the figure shows and you can also observe it in the progress trace output, service mapping, validation, and transforms all happen in the transaction before taking a (global) transaction lock.

At the same time, NSO tracks all of the data reads and writes from the start of the transaction, right until the lock and conflict check. This includes service mapping callbacks and XML templates, as well as transform and custom validation hooks, if you are using any. It even includes reads done as part of the YANG validation and rollback creation that NSO performs automatically.

If reads do not overlap with writes from other transactions, the conflict check passes. The change is written to the CDB and disseminated to the affected network devices, through the *prepare* and *commit* phases. Kickers and subscribers are called and, finally, the global lock can be released.

On the other hand, if there is overlap and the system detects a conflict, the transaction obviously cannot proceed. To recover if this happens, the transaction should be retried. Sometimes the system can do it automatically and sometimes the client itself must be prepared to retry it.



Note

An ingenious developer might consider avoiding the need for retries by using explicit locking, in the way the NETCONF `lock` command does. However, be aware that such an approach is likely to significantly degrade throughput of the whole system and is discouraged. If explicit locking is required, it should be considered with caution and sufficient testing.

In general, what affects the chance of conflict is the actual data that is read and written by each transaction. So, if there is more data, the surface for potential conflict is bigger. But you can minimize this chance by accounting for it in the application design.

Identifying Conflicts

When a transaction conflict occurs, NSO logs an entry in the developer log, often found at `logs/devel.log` or a similar path. Suppose you have the following code in Python:

```
with ncs.maapi.single_write_trans('admin', 'system') as t:
    root = ncs.maagic.get_root(t)
    # Read a value that can change during this transaction
    dns_server = root.mysvc_dns
    # Now perform complex work... or time.sleep(10) for testing
    # Finally, write the result
    root.some_data = 'the result'
    t.apply()
```

If the `/mysvc-dns` leaf changes while the code is executing, the `t.apply()` line fails and the developer log contains an entry similar to the following example:

```
<INFO> 23-Aug-2022::03:31:17.029 linux-ns0 ncs[<0.18350.3>]: ncs writeset collector:
check conflict tid=3347 min=234 seq=237 wait=0ms against=[3346] elapsed=1ms
-> conflict on: /mysvc-dns read: <<"10.1.2.2">> (op: get_delem tid: 3347)
write: <<"10.1.1.138">> (op: write tid: 3346 user: admin) phase(s): work
write tids: 3346
```

Here the transaction with id 3347 reads a value of `/mysvc-dns` as “10.1.2.2” but that value was changed by the transaction with id 3346 to “10.1.1.138” by the time the first transaction called `t.apply()`. The entry also contains some additional data, such as the user that initiated the other transaction and the low-level operations that resulted in the conflict.

At the same time, the Python code raises an `ncs.error.Error` exception, with `confd_errno` set to the value of `ncs.ERR_TRANSACTION_CONFLICT` and error text, such as the following:

```
Conflict detected (70): Transaction 3347 conflicts with transaction 3346 started by
user admin: /mysvc:mysvc-dns read-op get_delem write-op write in work phase(s)
```

In Java code, a matching `com.tailf.conf.ConfException` is thrown, with `errorCode` set to the `com.tailf.conf.ErrorCode.ERR_TRANSACTION_CONFLICT` value.

A thing to keep in mind when examining conflicts is that the transaction that did the read operations is the one that gets the error and causes the log entry, while the other transaction, doing the write operations to the same path, already completed successfully.

The error includes a reference to the `work` phase. The phase tells which part of the transaction encountered a conflict. The `work` phase signifies changes in an open transaction before it is applied. In practice, this is a direct read in the code that started the transaction before calling the `apply()` or `applyTrans()` function: the example reads the value of the leaf into `dns_server`.

On the other hand, if two transactions configure two service instances and the conflict arises in the mapping code, then the phase shows `transform` instead. It is also possible for a conflict to occur in more than one place, such as the phase `transform`, `work` denoting a conflict in both, the service mapping code as well as the initial transaction.

The complete list of conflict sources, that is, the possible values for the phase, is as follows:

- `work`: read in an open transaction before it is applied
- `rollback`: read during rollback file creation
- `pre-transform`: read while validating service input parameters according to the service YANG model
- `transform`: read during service (FASTMAP) or another transform invocation
- `validation`: read while validating the final configuration (YANG validation)

For example, `pre-transform` indicates that the service YANG model validation is the source of the conflict. This can help tremendously when you try to narrow down the conflicting code in complex scenarios. In addition, the phase information is useful when you troubleshoot automatic transaction retries in case of conflict: when the phase includes `work`, automatic retry is not possible.

Automatic Retries

In some situations, it is possible for NSO to retry a transaction that first failed to apply due to a conflict. A prerequisite is that NSO knows which code caused the conflict and that it can run that code again.

Changes done in the `work` phase are changes made directly by an external agent, such as a Python script connecting to the NSO or a remote NETCONF client. Since NSO is not in control of and is not aware of the logic in the external agent, it can only reject the conflicting transaction.

However, for the phases that follow the `work` phase, all the logic is implemented in NSO and NSO can run it on demand. For example, NSO is in charge of calling the service mapping code and the code can be run as many times as needed (a requirement for service re-deploy and similar). So, in case of a conflict, NSO can rerun all of the necessary logic to provision or deprovision a service.

In fact, NSO keeps checkpoints for each transaction, to restart it from the conflicting phase and save itself redoing the work from the preceding phases if possible. NSO automatically checks if the transaction checkpoint read- or write-set grows too large. This allows for larger transactions to go through without memory exhaustion. When all checkpoints are skipped, no transaction retries are possible, and the transaction fails. When later-stage checkpoints are skipped, the transaction retry will take more time.

Moreover, in case of conflicts during service mapping, NSO optimizes the process even further. It tracks the conflicting services in order to not schedule them concurrently in the future. This automatic retry behavior is enabled by default.

For services, retries can be configured further or even disabled under `/services/global-` settings. You can also find the service conflicts NSO knows about by running the **show services scheduling conflict** command. For example:

```
admin@ncs# unhide debug
admin@ncs# show services scheduling conflict | notab
services scheduling conflict mysvc-servicepoint mysvc-servicepoint
  type          dynamic
  first-seen    2022-08-27T17:15:10+00:00
  inactive-after 2022-08-27T17:15:09+00:00
  expires-after  2022-08-27T18:05:09+00:00
  ttl-multiplier 1
admin@ncs#
```

Since a given service may not always conflict and can evolve over time, NSO reverts to default scheduling after expiry time, unless new conflicts occur.

Sometimes, you know in advance that a service will conflict, either with itself or another service. You can encode this information in the service YANG model using the `conflicts-with` parameter under the `servicepoint` definition:

```
list mysvc {
  uses ncs:service-data;
  ncs:servicepoint mysvc-servicepoint {
    ncs:conflicts-with "mysvc-servicepoint";
    ncs:conflicts-with "some-other-servicepoint";
  }
  // ...
}
```

The parameter ensures NSO will never schedule and execute this service concurrently with another service using the specified servicepoint. It adds a non-expiring static scheduling conflict entry. This way, you can avoid the unnecessary occasional retry when the dynamic scheduling conflict entry expires.

Declaring a conflict with itself is especially useful when you have older, non-thread-safe service code that cannot be easily updated to avoid threading issues.

For the NSO CLI and JSON-RPC (WebUI) interfaces, a commit of a transaction that results in a conflict will trigger an automatic rebase and retry when the resulting configuration is the same despite the conflict. If the rebase does not resolve the conflict, the transaction will fail. The conflict can, in some CLI cases, be resolved manually. A successful automatic rebase and retry will generate something like the following pseudo log entries in the developer log (trace log level):

```
<INFO> ... check for read-write conflicts: conflict found
<INFO> ... rebase transaction
...
<INFO> ... rebase transaction: ok
<INFO> ... retrying transaction after rebase
```

Handling Conflicts

When a transaction fails to apply due to a read-write conflict in the work phase, NSO rejects the transaction and returns a corresponding error. In such a case, you must start a new transaction and redo all the changes.

Why is this necessary? Suppose you have code, let's say as part of a CDB subscriber or a standalone program, similar to the following Python snippet:

```
with ncs.maapi.single_write_trans('admin', 'system') as t:
```

```

if t.get_elem('/mysvc-use-dhcp') == True:
    # do something
else:
    # do something entirely different that breaks
    # your network if mysvc-use-dhcp happens to be true
t.apply()

```

If `mysvc-use-dhcp` has one value when your code starts provisioning but is changed mid-process, your code needs to restart from the beginning or you can end up with a broken system. To guard against such a scenario, NSO needs to be conservative and return an error.

Since there is a chance of a transaction failing to apply due to a conflict, robust code should implement a retry scheme. You can implement the retry algorithm yourself, or you can use one of the provided helpers.

In Python, `Maapi` class has a `run_with_retry()` method, which creates a new transaction and calls a user supplied function to perform the work. On conflict, `run_with_retry()` will recreate the transaction and call the user function again. For details, please see the relevant API documentation.

The same functionality is available in Java as well, as the `Maapi.ncsRunWithRetry()` method. Where it differs from the Python implementation is that it expects the function to be implemented inside a `MaapiRetryableOp` object.

As an alternative option, available only in Python, you can use the `retry_on_conflict()` function decorator.

Example code for each of these approaches is shown next. In addition, the `examples.ncs/development-guide/concurrency-model/retry` example showcases this functionality as part of a concrete service.

Example Retrying Code in Python

Suppose you have some code in Python, such as the following:

```

with ncs.maapi.single_write_trans('admin', 'python') as t:
    root = ncs.maagic.get_root(t)
    # First read some data, then write some too.
    # Finally, call apply.
    t.apply()

```

Since the code performs reads and writes of data in NSO through a newly established transaction, there is a chance of encountering a conflict with another, concurrent transaction.

On the other hand, if this was service mapping code, you wouldn't be creating a new transaction yourself because the system would already provide one for you. You wouldn't have to worry about the retry because, again, the system would handle it for you through the automatic mechanism described earlier.

Yet, you may find such code in CDB subscribers, standalone scripts, or action implementations. As a best practice, the code should handle conflicts.

If you have an existing `ncs.maapi.Maapi` object already available, the simplest option might be to refactor the actual logic into a separate function and call it through `run_with_retry()`. For the current example, this might look like the following:

```

def do_provisioning(t):
    """Function containing the actual logic"""
    root = ncs.maagic.get_root(t)
    # First read some data, then write some too.
    # ...

```

Example Retrying Code in Java

```

# Finally, return True to signal apply() has to be called.
return True

# Need to replace single_write_trans() with a Maapi object
with ncs.maapi.Maapi() as m:
    with ncs.maapi.Session(m, 'admin', 'python'):
        m.run_with_retry(do_provisioning)

```

If the new function is not entirely independent and needs additional values passed as parameters, you can wrap it inside an anonymous (lambda) function:

```
m.run_with_retry(lambda t: do_provisioning(t, one_param, another_param))
```

An alternative implementation with a decorator is also possible and might be easier to implement if the code relies on the `single_write_trans()` or similar function. Here, the code does not change unless it has to be refactored into a separate function. The function is then adorned with the `@ncs.maapi.retry_on_conflict()` decorator. For example:

```

from ncs.maapi import retry_on_conflict

@retry_on_conflict()
def do_provisioning():
    # This is the same code as before but in a function
    with ncs.maapi.single_write_trans('admin', 'python') as t:
        root = ncs.maagic.get_root(t)
        # First read some data, then write some too.
        #
        # ...
        # Finally, call apply().
        t.apply()

do_provisioning()

```

The major benefit of this approach is when the code is already in a function and only a decorator needs to be added. It can also be used with methods of the Action class and alike.

```

class MyAction(ncs.dp.Action):
    @ncs.dp.Action.action
    @retry_on_conflict()
    def cb_action(self, uinfo, name, kp, input, output, trans):
        with ncs.maapi.single_write_trans('admin', 'python') as t:
            ...

```

For actions in particular, please note that the order of decorators is important and the decorator is only useful when you start your own write transaction in the wrapped function. This is what `single_write_trans()` does in the preceding example because the old transaction cannot be used any longer in case of conflict.

Example Retrying Code in Java

Suppose you have some code in Java, such as the following:

```

public class MyProgram {
    public static void main(String[] args) throws Exception {
        Socket socket = new Socket("127.0.0.1", Conf.NCS_PORT);
        Maapi maapi = new Maapi(socket);
        maapi.startUserSession("admin", InetAddress.getByName(null),
                              "system", new String[]{},
                              MaapiUserSessionFlag.PROTO_TCP);
        NavuContext context = new NavuContext(maapi);
        int tid = context.startRunningTrans(Conf.MODE_READ_WRITE);
    }
}

```

```

    // Your code here that reads and writes data.

    // Finally, call apply.
    context.applyClearTrans();
    maapi.endUserSession();
    socket.close();
}
}

```

In order to read and write some data in NSO, the code starts a new transaction with the help of `NavuContext.startRunningTrans()` but could have called `Maapi.startTrans()` directly as well. Regardless of the way such a transaction is started, there is a chance of encountering a read-write conflict. To handle those cases, the code can be rewritten to use `Maapi.ncsRunWithRetry()`.

The `ncsRunWithRetry()` call creates and manages a new transaction, then delegates work to an object implementing the `com.tailf.maapi.MaapiRetryableOp` interface. So, you need to move the code that does the work into a new class, let's say `MyProvisioningOp`:

```

public class MyProvisioningOp implements MaapiRetryableOp {
    public boolean execute(Maapi maapi, int tid)
        throws IOException, ConfException, MaapiException
    {
        // Create context for the provided, managed transaction;
        // note the extra parameter compared to before and no calling
        // context.startRunningTrans() anymore.
        NavuContext context = new NavuContext(maapi, tid);

        // Your code here that reads and writes data.

        // Finally, return true to signal apply() has to be called.
        return true;
    }
}

```

This class does not start its own transaction anymore but uses the transaction handle `tid`, provided by the `ncsRunWithRetry()` wrapper.

You can create the `MyProvisioningOp` as an inner or nested class if you wish so but note that, depending on your code, you may need to designate it as a `static class` to use it directly as shown here.

If the code requires some extra parameters when called, you can also define additional properties on the new class and use them for this purpose. With the new class ready, you instantiate and call into it with the `ncsRunWithRetry()` function. For example:

```

public class MyProgram {
    public static void main(String[] args) throws Exception {
        Socket socket = new Socket("127.0.0.1", Conf.NCS_PORT);
        Maapi maapi = new Maapi(socket);
        maapi.startUserSession("admin", InetAddress.getByName(null),
            "system", new String[]{}, MaapiUserSessionFlag.PROTO_TCP);
        // Delegate work to MyProvisioningOp, with retry.
        maapi.ncsRunWithRetry(new MyProvisioningOp());
        // No more calling applyClearTrans() or friends,
        // ncsRunWithRetry() does that for you.
        maapi.endUserSession();
        socket.close();
    }
}

```

And what if your use-case requires you to customize how the transaction is started or applied? `ncsRunWithRetry()` can take additional parameters that allow you to control those aspects. Please see the relevant API documentation for the full reference.

Designing for Concurrency

In general, transaction conflicts in NSO cannot be avoided altogether, so your code should handle them gracefully with retries. Retries are required to ensure correctness but do take up additional time and resources. Since a high percentage of retries will notably decrease throughput of the system, you should endeavor to construct your data models and logic in a way that minimizes the chance of conflicts.

A conflict arises when one transaction changes a value that one or more other ongoing transactions rely on. From this, you can make a couple of observations that should help guide your implementation.

First, if the shared data changes infrequently, it will rarely cause a conflict (regardless of the amount of reads) because it only affects the transactions happening at the time it is changed. Conversely, a frequent change can clash with other transactions much more often and warrants spending some effort to analyze and possibly make conflict-free.

Next, if a transaction runs a long time, a greater number of other write transactions can potentially run in the mean time, increasing the chances of a conflict. For this reason, you should avoid long-running read-write transactions.

Likewise, the more data nodes and the different parts of the data tree the transaction touches, the more likely it is to run into a conflict. Limiting the scope and the amount of the changes to shared data is an important design aspect.

Also, when considering possible conflicts, you must account for all the changes in the transaction. This includes changes propagated to other parts of the data model through dependencies. For example, consider the following YANG snippet. Changing a single `provision-dns` leaf also changes every `mysvc` list item because of the `when` statement.

```
leaf provision-dns {
    type boolean;
}
list mysvc {
    container dns {
        when ".../provision-dns";
        // ...
    }
}
```

Ultimately, what matters is the read-write overlap with other transactions. Thus, you should avoid needless reads in your code: if there are no reads of the changed values, there can't be any conflicts.

Avoiding Needless Reads

A technique used in some existing projects, in service mapping code and elsewhere, is to first prepare all the provisioning parameters by reading in a number of things from the CDB. But some of these parameters, or even most, may not really be needed for that particular invocation.

Consider the following service mapping code:

```
def cb_create(self, tctx, root, service, proplist):
    device = root.devices.device[service.device]

    # Search device interfaces and CDB for mgmt IP
    device_ip = find_device_ip(device)
```

```

# Find the best server to use for this device
ntp_servers = root.my_settings.ntp_servers
use_ntp_server = find_closest_server(device_ip, ntp_servers)

if service.do_ntp:
    device.ntp.servers.append(use_ntp_server)

```

Here, a service performs NTP configuration when enabled through the `do_ntp` switch. But even if the switch is off, there are still a lot of reads performed. If one of the values changes during provisioning, such as the list of the available NTP servers in `ntp_servers`, it will cause a conflict and a retry.

An improved version of the code only calculates the NTP server value if it is actually needed:

```

def cb_create(self, tctx, root, service, proplist):
    device = root.devices.device[service.device]

    if service.do_ntp:
        # Search device interfaces and CDB for mgmt IP
        device_ip = find_device_ip(device)

        # Find the best server to use for this device
        ntp_servers = root.my_settings.ntp_servers
        use_ntp_server = find_closest_server(device_ip, ntp_servers)

        device.ntp.servers.append(use_ntp_server)

```

Handling Dependent Services

Another thing to consider in addition to the individual service implementation is the placement and interaction of the service within the system. What happens if one service is used to generate input for another service? If the two services run concurrently, writes of the first service will invalidate reads of the other one, pretty much guaranteeing a conflict. Then it is wasteful to run both services concurrently and they should really run serially.

A way to achieve this is through a design pattern called stacked services. You create a third service that instantiates the first service (generating the input data) before the second one (dependent on the generated data).

Searching and Enumerating Lists

When there is a need to search or filter a list for specific items, you will often find for-loops or similar constructs in the code. For example, to configure NTP, you might have the following:

```

for ntp_server in root.my_settings.ntp_servers:
    # Only select active servers
    if ntp_server.is_active:
        # Do something

```

This approach is especially prevalent in ordered-by-user lists, since the order of the items and their processing is important.

The interesting bit is that such code reads every item in the list. If the list is changed while the transaction is ongoing, you get a conflict with the message identifying the `get_next` operation (which is used for list traversal). This is not very surprising: if another active item is added or removed, it changes the result of your algorithm. So, this behavior is expected and desirable to ensure correctness.

However, you can observe the same conflict behavior in less obvious scenarios. If the list model contains a unique YANG statement, NSO performs the same kind of enumeration of list items for you to verify the

unique constraint. Likewise, a `must` or `when` statement can also trigger evaluation of every item during validation, depending on the XPath expression.

Basically, NSO knows how to discern between access to specific list items based on the key value, where it tracks reads only to those particular items, and enumerating the list, where no key value is supplied and a list with all elements is treated as a single item. This works for your code as well as for the XPath expressions (in YANG and otherwise). As you can imagine, adding or removing items in the first case doesn't cause conflicts, while in the second one it does.

In the end, it depends on the situation whether list enumeration can affect throughput or not. In the example, the NTP servers could be configured manually, by the operator, so they would rarely change, making it a non-issue. But your use case might differ.

Python Assigning to Self

As several service invocations may run in parallel, Python self-assignment in service handling code can cause difficult-to-debug issues. Therefore, NSO checks for such patterns and issues an alarm (default) or a log entry containing a warning and a keypath to the service instance that caused the warning. See [Chapter 11, *The NSO Python VM*](#) for details.



CHAPTER 25

Developing Alarm Applications

- [Introduction, page 507](#)
- [Using the Alarms Sink, page 508](#)
- [Using the Alarms Source, page 510](#)
- [Extending the alarm manager, adding user defined alarm types and fields, page 511](#)
- [Mapping alarms to objects, page 513](#)

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 *User Guide* are well understood before reading this section.

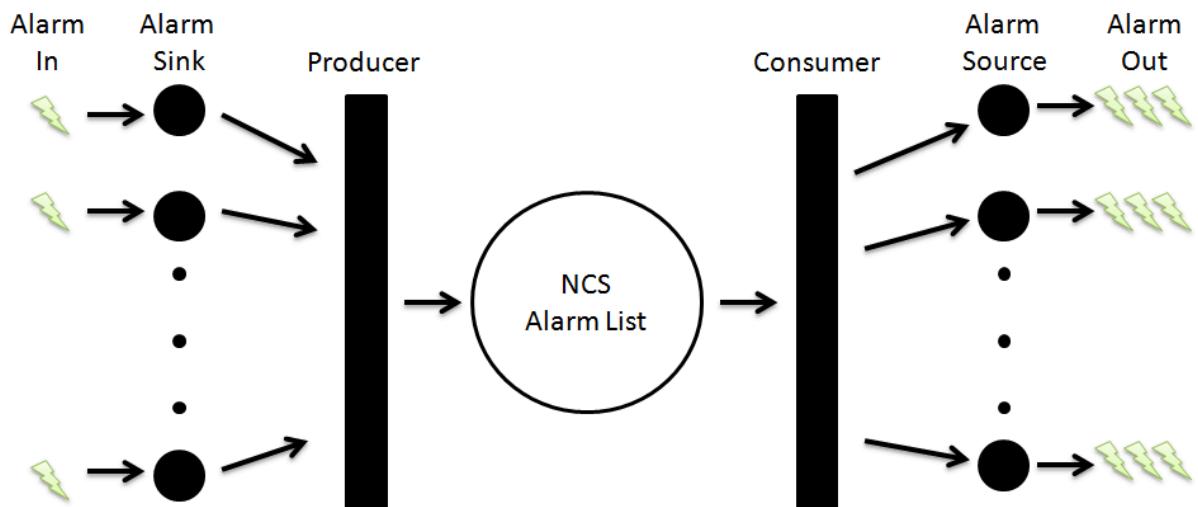
The Alarm API provides a simplified way of managing your alarms for the most common alarm management use cases. The API is divided into a producer and a consumer part.

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 201. 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 202. 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 203. 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 204. 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 205. 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 206. 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 207. 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 208. 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 209. 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 210. 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 211. 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 212. 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 213. 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;
}
```



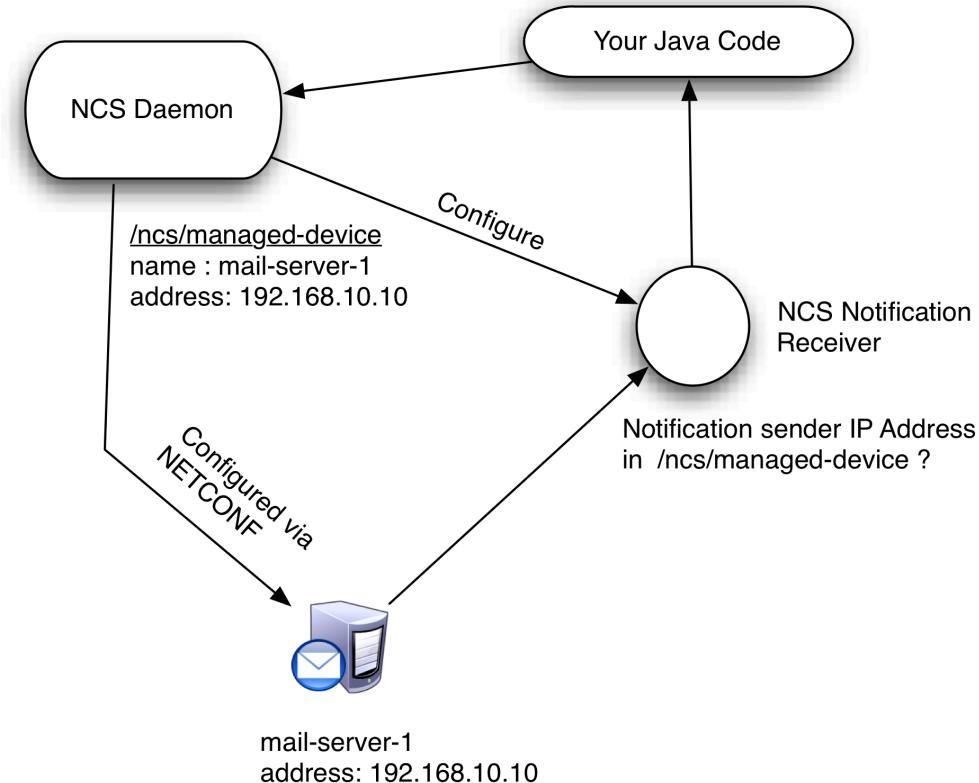
CHAPTER 26

SNMP Notification Receiver

- [Introduction, page 515](#)
- [Configuring NSO to Receive SNMP Notifications, page 516](#)
- [Built-in Filters, page 516](#)
- [Notification Handlers, page 517](#)

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.

```
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,
        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:

```

/**
 * 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 27

The web server

- [Introduction, page 521](#)
- [Web server capabilities, page 521](#)
- [CGI support, page 522](#)
- [Storing TLS data in database, page 523](#)
- [Package upload, page 525](#)

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 *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 *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 list 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 list 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_INTO":
- "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 214. 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 215. 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-----
> 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 216. 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
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
[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
...

```

Package upload

The web server includes support for uploading packages to "/package-upload" using HTTP POST from the local host to the NSO host, making them *installable* there. It is disabled by default, but can be enabled in ncs.conf - see the manual pages of the section called "CONFIGURATION PARAMETERS" in *Manual Pages*.

By default only upload 1 file per request will be processed and any remaining file parts after that will result in an error and its content will be ignored. To allow multiple files in a request you can increase /ncs-config/webui/package-upload/max-files.

Example 217. Valid package example

```
curl \
--cookie 'sessionid=sess12541119146799620192;' \
-X POST \
-H "Cache-Control: no-cache" \
-F "upload=@path/to/some-valid-package.tar.gz" \
http://127.0.0.1:8080/package-upload

[
{
  "result": {
    "filename": "some-valid-package.tar.gz"
  }
}]
```

Example 218. Invalid package example

```
curl \
--cookie 'sessionid=sess12541119146799620192;' \
-X POST \
-H "Cache-Control: no-cache" \
-F "upload=@path/to/some-invalid-package.tar.gz" \
http://127.0.0.1:8080/package-upload

[
{
  "error": {
    "filename": "some-invalid-package.tar.gz",
    "data": {
      "reason": "Invalid package contents"
    }
}]
```

```
        }
    }
]
```

The AAA infrastructure can be used to restrict access to library functions using command rules:

```
<cmdrule xmlns="http://tail-f.com/yang/acm">
<name>deny-package-upload</name>
<context>webui</context>
<command>::webui:: package-upload</command>
<access-operations>exec</access-operations>
<action>deny</action>
</cmdrule>
```

Note how the command is prefixed with "::webui:: ". This tells the AAA engine to apply the command rule to webui API functions.

You can read more about command rules in "The AAA infrastructure" chapter in this User Guide.



CHAPTER 28

Kicker

- [Introduction, page 527](#)
- [Kicker action invocation, page 527](#)
- [Data Kicker Concepts, page 528](#)
- [Notification Kicker Concepts, page 534](#)
- [Nano Services Reactive FastMap with Kicker, page 538](#)
- [Debugging kickers, page 538](#)

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:

<code>kicker-id</code>	The id (name) of the invoking kicker.
<code>path</code>	The path of the current monitor triggering the kicker
<code>tid</code>	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 a 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:

- Kickers are triggered by changes in the sub-tree indicated by the `monitor` parameter.
- Actions are invoked during the commit phase. Hence an aborted transactions never trigger kickers.
- Kickers process both, configuration and operational data changes, but can be configured to react to a certain type of change only.
- 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.
```

The trigger-expression can in some cases be used to refine the monitor of the kicker, as to avoid unnecessary evaluations. Let's change something below the monitor that doesn't touch the nodes in the trigger-expression:

```
admin(config)# sys ifc port-0 speed ten
admin(config-ifc-port-0)# commit | debug kicker
Commit complete.
```

Notice there was no evaluation done.

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 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
```

Notification Kicker Concepts

```
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 a1 \
> monitor /services/properties/wsp:web-site/profile \
> kick-node /services/wse:actions action-name diffcheck
```

```
admin@ncs(config-data-kicker-a1)# commit
admin@ncs(config-data-kicker-a1)# top
admin@ncs(config)# show full-configuration kickers data-kicker a1
kickers data-kicker a1
monitor      /services/properties/wsp:web-site/profile
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 ordering of kicker invocations is generally not guaranteed. That is, a kicker triggered at a later time might execute before a kicker that was triggered earlier, and kickers triggered for the same

subscription may execute in any order. A priority and a serializer value can be used to modify this behavior.

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:

DEVICE	The name of the device emitting the current notification.
SUBSCRIPTION_NAME	The name of the current subscription from which the notification was received. the kicker
NOTIFICATION_NAME	The name of the current notification.
NOTIFICATION_NS	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.

Serializer and priority values

These values are used to ensure order of kicker execution. Priority orders kickers for the same notification event, while serializer orders kickers chronologically for different notification events. By default, when no serializer or priority value is given, kickers may be triggered in any order and in parallel. However, some situations may require stricter ordering and setting serializer and priority in kicker configuration allows you to achieve it.

If priority for a set of kickers is specified, for each individual notification event, the kickers that match are executed in order, going from priority 0 to 255. For example, kicker K1 with priority 5 is executed before kicker K2 with priority 8, which triggered for the same notification.

Parallel execution of kickers can also result in a situation where a kicker for a notification is executed after the kicker for a later notification. That is, even though the trigger for the first kicker came first, this kicker might have a priority set and must wait for other kickers to execute first, while kicker for the next notification can execute right away. If there is a dependency between these two kickers, serializer value can ensure chronological ordering.

A serializer is a simple integer value between 0 and 255. Notification kickers configured with the same value will be executed in the order in which they were triggered, relative to each other. For example, suppose there are three kickers configured: T1 and T2 with serializer set to 10, and T3 with serializer of 20. NSO receives two notifications, the first triggering T1 and T3, and the second triggering T2. Because of the serializer, NSO guarantees T1 will be invoked before T2. But T2, even though it came in later, could potentially be invoked before T3 because they are not serialized (have different serializer value).

When using both, serializer and priority, only kickers with the same serializer value are priority ordered, that is, serializer value takes precedence. For example, kicker Q1 with serializer 10 and priority 15 may execute before or after kicker Q2 with serializer 20 and priority 4. The reason is Q1 may need to wait for other kickers with serializer 10 from previous events. The same is true for Q2 and previous kickers with serializer 20.

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.

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}/extension
op = MOP_CREATED
```

```

newValue = 4668
path = /ncs:devices/device{www0}/notifications.../data/notif:linkUp/linkProperty{0}/extensions{1}
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

```

Nano Services Reactive FastMap with Kicker

Nano services use kickers to trigger executing state callback code, run templates and execute actions according to a plan when pre-conditions are met. For more information see the section called “ Nano Services for Provisioning with Side Effects ” and Chapter 31, *Nano Services for Staged Provisioning*.

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:name]
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 219. 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 29

Scheduler

- [Introduction, page 541](#)
- [Scheduling Periodic Work, page 541](#)
- [Scheduling Non-recurring Work, page 542](#)
- [Scheduling in a HA Cluster, page 542](#)
- [Troubleshooting, page 543](#)

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 220. 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.

Periodic compaction

Compaction in NSO can take a considerable amount of time, during which transactions are blocked. To avoid disruption, it might be advantageous to schedule compaction during times of low NSO utilization. This can be done using the NSO scheduler and a service. See examples.ncs/development-guide/periodic-compaction for an example that demonstrates how to create a periodic compaction service that can be scheduled using the NSO scheduler.

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 secondary 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 *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.

Suspending the Scheduler

```
admin# scheduler suspend
```

When ready the scheduler can be resumed.

```
admin# scheduler resume
```



CHAPTER 30

Progress Trace

- [Introduction, page 545](#)
- [Configuring Progress Trace, page 546](#)
- [Report Progress Events from User Code, page 548](#)

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.

```
applying transaction for running datastore usid=41 tid=1761 trace-id=d7f06482-41ad-4151-938d-
entering validate phase
  2021-05-25T17:28:12.267 taking transaction lock... ok (0.000 s)
  2021-05-25T17:28:12.267 holding transaction lock...
  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-manager: service /ordserv[name='o2']: run service... ok (0.000 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 (0.045 s)
entering write-start phase
  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 (0.001 s)
entering prepare phase
  2021-05-25T17:28:12.314 cdb: prepare
```

```
2021-05-25T17:28:12.314 device-manager: prepare
leaving prepare phase (0.003 s)
entering commit phase
2021-05-25T17:28:12.317 cdb: commit
2021-05-25T17:28:12.318 service-manager: commit
2021-05-25T17:28:12.318 device-manager: commit
2021-05-25T17:28:12.320 holding transaction lock: ok (0.033 s)
leaving commit phase (0.002 s)
applying transaction for running datastore usid=41 tid=1761 trace-id=d7f06482-41ad-4151-938d-7a8
```

Some actions (usually those involving device communication) also produces progress data.

```
admin@ncs% request devices device ce0 sync-from dry-run | details very-verbose
running action /devices/device\[name='ce0'\]/sync-from usid=41 tid=1800 trace-id=fff4d4b0-5688-4
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)
2021-05-25T17:28:12.267 device ce0: holding device lock...
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)
...
2021-05-25T17:28:12.320 device ce0: holding device lock: ok (0.033 s)
2021-05-25T17:31:31.249 device ce0: sync-from: ok (0.026 s)
running action /devices/device\[name='ce0'\]/sync-from usid=41 tid=1800 trace-id=fff4d4b0-5688-4
```

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 Python, and Java, Erlang and C.

Python ncs.maapi Example

```
class ServiceCallbacks(Service):
    @Service.create
    def cb_create(self, tctx, root, service, proplist):
        maapi = ncs.maagic.get_maapi(root)
        trans = maapi.attach(tctx)

        with trans.start_progress_span("service create()", path=service._path):
            ipv4_addr = None
            with trans.start_progress_span("allocate IP address") as sp1:
                self.log.info('alloc trace-id: ' + sp1.trace_id + \
                    ' span-id: ' + sp1.span_id)
                ipv4_addr = alloc_ipv4_addr('192.168.0.0', 24)
                trans.progress_info('got IP address ' + ipv4_addr)
            with trans.start_progress_span("apply template",
                attrs={'ipv4_addr':ipv4_addr}) as sp12:
                self.log.info('templ trace-id: ' + sp12.trace_id + \
                    ' span-id: ' + sp12.span_id)
                vars = ncs.template.Variables()
                vars.add('IPV4_ADDRESS', ipv4_addr)
                template = ncs.template.Template(service)
                template.apply('ipv4-addr-template', vars)
```

Further details can be found in the NSO Python API reference under `ncs.maapi.start_progress_span` and `ncs.maapi.progress_info`.

Java com.tailf.progress.ProgressTrace Example

```
@ServiceCallback(servicePoint="...", callType=ServiceCBType.CREATE)
public Properties create(ServiceContext context,
                        NavuNode service,
                        NavuNode ncsRoot,
                        Properties opaque)
                        throws DpCallbackException {
    try {
        Maapi maapi = service.context().getMaapi();
        int tid = service.context().getMaapiHandle();
        ProgressTrace progress = new ProgressTrace(maapi, tid,
                                                    service.getConfPath());
        Span sp1 = progress.startSpan("service create()");

        Span sp11 = progress.startSpan("allocate IP address");
        LOGGER.info("alloc trace-id: " + sp11.getTraceId() +
                    " span-id: " + sp11.getSpanId());
        String ipv4Addr = allocIpv4Addr("192.168.0.0", 24);
        progress.event("got IP address " + ipv4Addr);
        progress.endSpan(sp11);

        Attributes attrs = new Attributes();
        attrs.set("ipv4_addr", ipv4Addr);
        Span sp12 = progress.startSpan(Maapi.Verbosity.NORMAL,
                                      "apply template", attrs, null);
```

```
LOGGER.info("temp1 trace-id: " + sp12.getTraceId() +  
           " span-id: " + sp12.getSpanId());  
TemplateVariables ipVar = new TemplateVariables();  
ipVar.putQuoted("IPV4_ADDRESS", ipv4Addr);  
Template ipTemplate = new Template(context, "ipv4-addr-template");  
ipTemplate.apply(service, ipVar);  
progress.endSpan(sp12);  
  
progress.endSpan(sp1);
```

Further details can be found in the NSO Java API reference under
`com.tailf.progress.ProgressTrace` and `com.tailf.progress.Span`.



CHAPTER 31

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 552](#)
- [Benefits and Use Cases, page 558](#)
- [Backtracking and Staged Delete, page 559](#)
- [Managing Side Effects, page 562](#)
- [Multiple and Dynamic Plan Components, page 564](#)
- [Netsim Router Provisioning Example, page 566](#)
- [Zombie Services, page 569](#)
- [Using Notifications to Track the Plan and its Status, page 570](#)
- [Developing and Updating a Nano Service, page 573](#)
- [Implementation Reference, page 574](#)

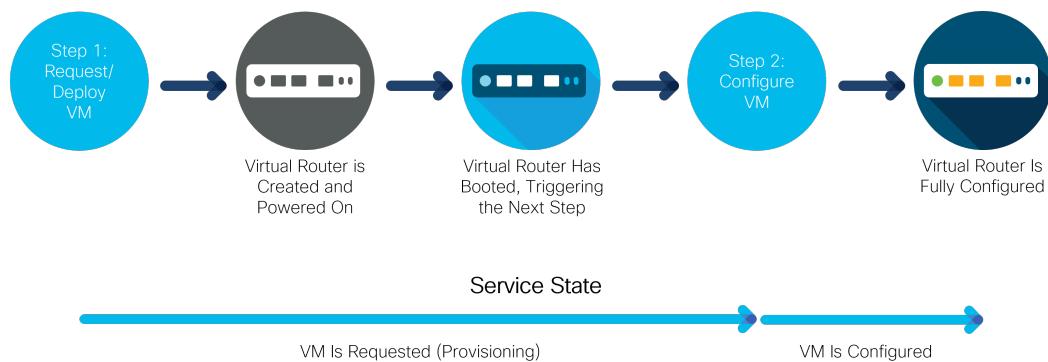
- Graceful Link Migration Example, page 588

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 221. 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

By default, the plan outline consists of a single component, the `self` component, with the two states `init` and `ready`. It can be used to track the progress of the service as a whole. You can add any number of additional components and states to form the nano service.

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;
    }

    identity vrouter {
        base ncs:plan-component-type;
    }

    ncs:plan-outline vrouter-plan {
        description "Plan for configuring a VM-based router";

        ncs:component-type "vr:vrouter" {
            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. 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 one component and 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="vr:vrouter"
                  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
                                  'vr:vrouter', # 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 "vr:vrouter" {
        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";
    }
}
```

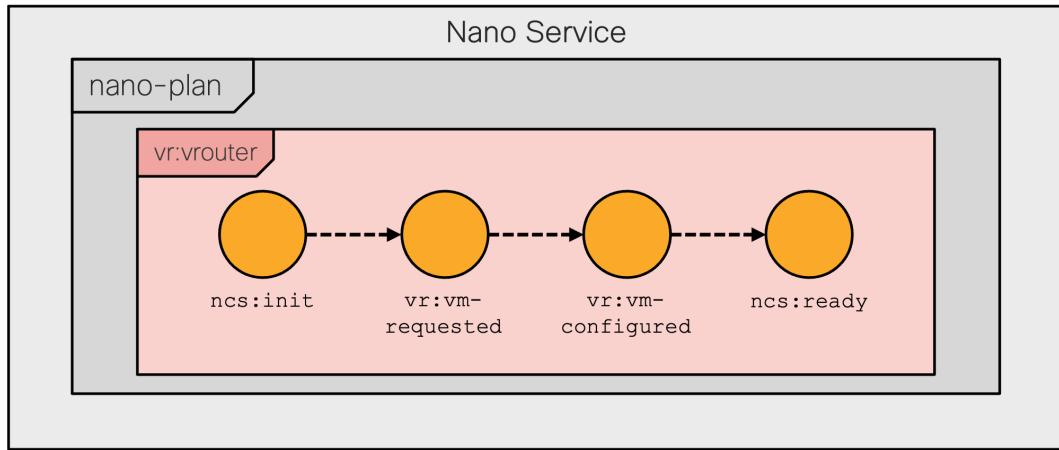
For the service to use it, the plan outline must be linked to a service point with the help of a *behavior tree*. The main purpose of a behavior tree is to allow a service to dynamically instantiate components, based on service parameters. Dynamic instantiation is not always required and the behavior 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 "'vrouter'" {
            ncs:component-type-ref "vr:vrouter";
        }
    }
}
```

This behavior tree always creates a single “vrouter” 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 222. 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

POST
ACTION

TYPE	NAME	TRACK	GOAL	STATE	STATUS	WHEN	ref	STATUS
self	self	false	-	init	reached	2023-08-11T07:45:20	-	-
				ready	not-reached	-	-	-
vrouter	vrouter	false	-	init	reached	2023-08-11T07:45:20	-	-
				vm-requested	reached	2023-08-11T07:45:20	-	-
				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# vrouter 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 28, 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.

TYPE	NAME	TRACK	GOAL	STATE	STATUS	WHEN	POST ACTION	
							ref	STATUS
self	self	false	-	init	reached	2023-08-11T07:45:20	-	-
				ready	reached	2023-08-11T07:47:36	-	-
vrouter	vrouter	false	-	init	reached	2023-08-11T07:45:20	-	-
				vm-requested	reached	2023-08-11T07:45:20	-	-
				vm-configured	reached	2023-08-11T07:47:36	-	-
				ready	reached	2023-08-11T07:47:36	-	-

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

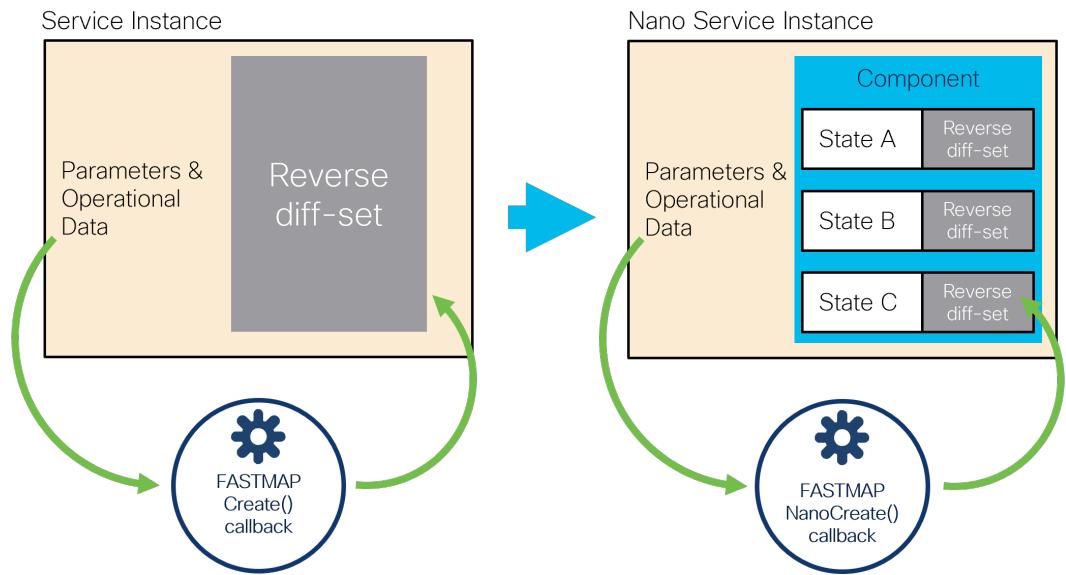
Every service in NSO has a YANG definition of the service parameters, a service point name, and an implementation of the service point `create()` callback. Normally, when a service is committed, the FASTMAP algorithm removes all previous data changes internally, and presents the service data to the `create()` 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 223. 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.

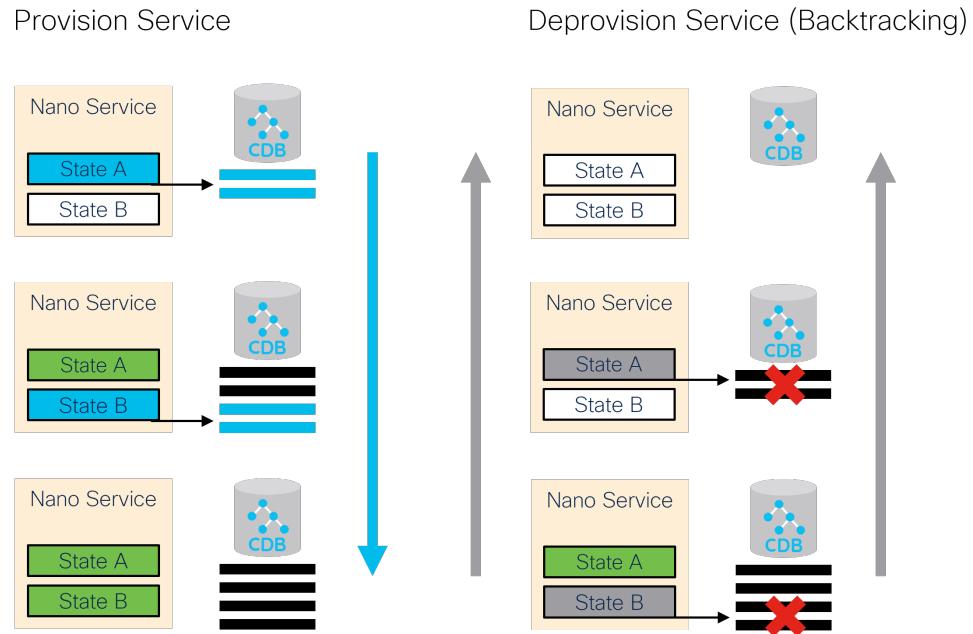
Backtracking and Staged Delete

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 224. Staged delete with backtracking

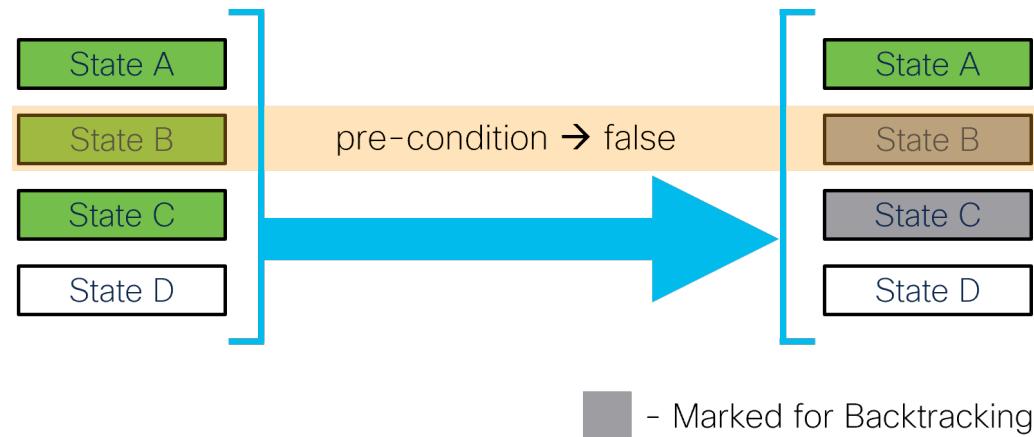


In doing so, the back-tracking process checks for a *delete pre-condition* of a state. A delete pre-condition is similar to the create pre-condition, but only relevant when back-tracking. If the condition is not fulfilled, the back-tracking process stops and waits until it becomes satisfied. Behind the 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 225. 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.

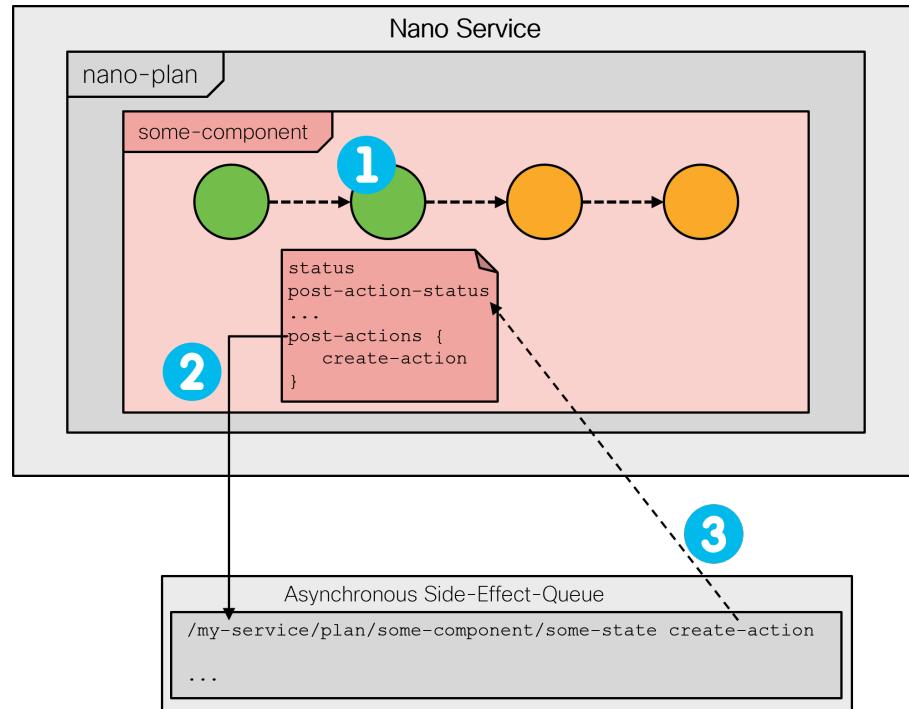
A common solution is to perform these actions outside of the configuration transaction. Nano services provide this functionality through the *post-actions* mechanism, using a `post-action-node` statement for a state. It is a definition of an action that should be invoked after the state has been reached 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 226. 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 *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, you can perform the action **reschedule**.

```
$ ncs_cli -u admin
admin@ncs> show side-effect-queue side-effect status
ID  STATUS
-----
2   failed

[ok][2023-08-15 11:01:10]
admin@ncs> request side-effect-queue side-effect 2 reschedule
side-effect-id 2
[ok][2023-08-15 11:01:18]
```

Or 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 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 actually an XPath expression. The quotes ensure the name is used verbatim when the expression is evaluated.

With multiple components in place, the implicit `self` component reflects the cumulative status of the service. 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.

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 "'router'" { ... }
        ncs:selector {
            ncs:pre-condition {
                ncs:monitor "$SERVICE" {
                    ncs:trigger-exp "use-virtual-devices = 'true'";
                }
            }
            ncs:create-component "'vm1'" { ... }
            ncs:create-component "'vm2'" { ... }
        }
    }
}
```

The described behavior tree always synthesizes the router 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, onboard 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.

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:

```
// vr:router name
ncs:variable "NAME" {
    ncs:value-expr "current()/name";
}
// vr:router component name
ncs:variable "DONAME" {
    ncs:value-expr "concat(current()/name, '-day0')";
}
// vr:router 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-router` 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 vrouter 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 configure 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-exp "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.

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, 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 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>nanol</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 *Northbound APIs* for further reference.

NETCONF create subscription using netconf-console:

```
$ netconf-console create-subscription=service-state-changes
```

The trace-id in the Notification

```
<?xml version="1.0" encoding="UTF-8"?>
<notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
  <eventTime>2021-11-16T20:36:06.324322+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 *Northbound APIs* for further reference.

CLI show received notifications using **ncs_cli**:

```
$ ncs_cli -u admin -C <<<'show notification stream service-state-changes'

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 component. All components are expected to have `ncs:init` as its first state and `ncs:ready` as its last state. A component type 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.

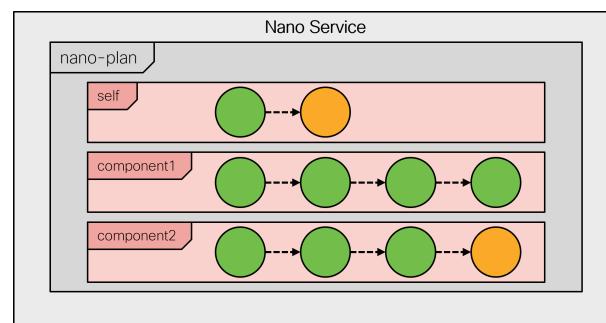
Plan component is removed

When a plan component is removed and has reached states that contain reverse diff-sets.

Service is deleted

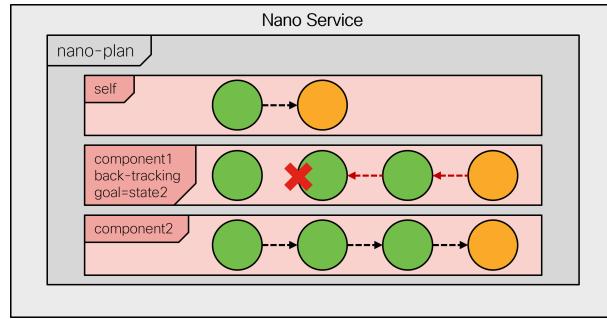
When a service is deleted, NSO will set all plan components to back-tracking mode before deleting the service.

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.

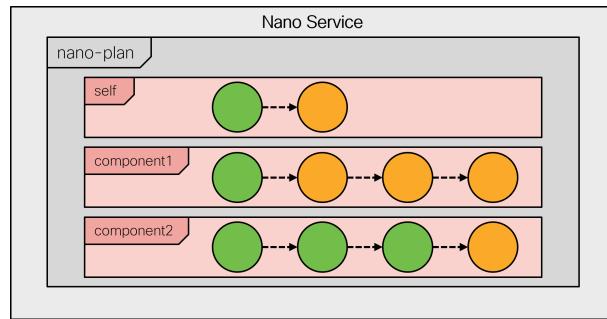


While traversing the states, a create pre-condition that was previously satisfied may become un-satisfied. If there are subsequent reached states that contain reverse diff-sets, then the component must be set to back-tracking mode. The back-tracking mode has as its goal to revert all changes up to the 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.

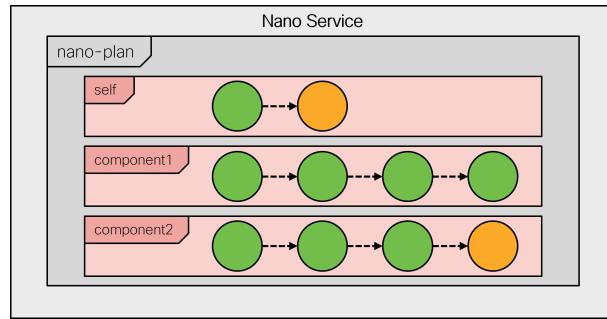
Back-Tracking



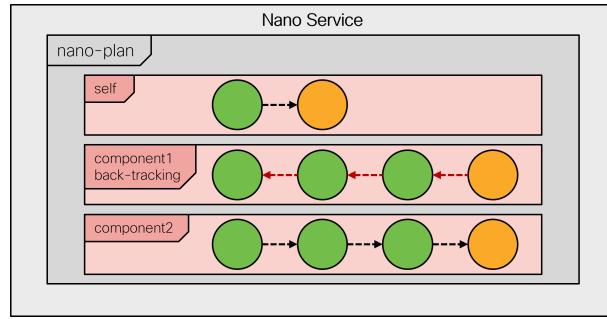
When the back-tracking plan component has reached its goal state, the component is set to normal mode again. The state's create pre-condition is evaluated and if it is satisfied the state is entered or otherwise a kicker is created as described above.



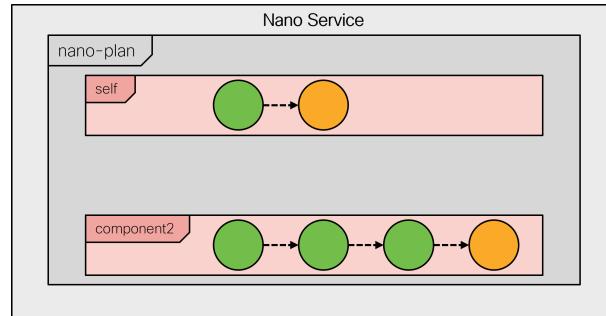
In some circumstances a complete plan component is removed (for example, if the service input parameters are changed). If this happens, the plan component is checked if it contains reached states that contain reverse diff-sets.



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.

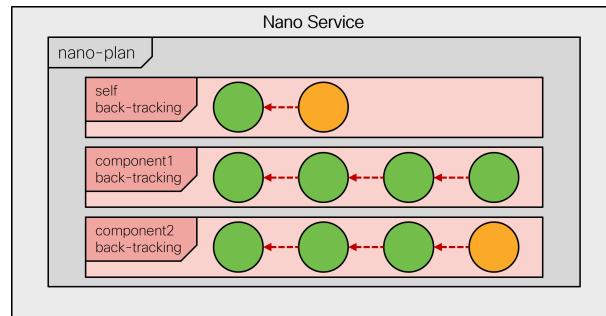


In this case, there is no specified goal state for the back-tracking. This means that when all the states have been reverted, the component is automatically deleted.

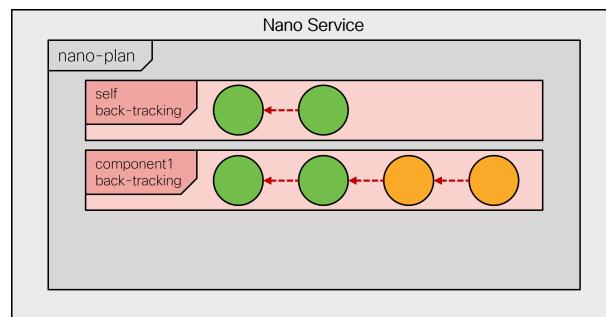


If a service is deleted, all components are set to back-tracking mode. The service becomes a zombie, storing away its plan states so that the service configuration can be removed.

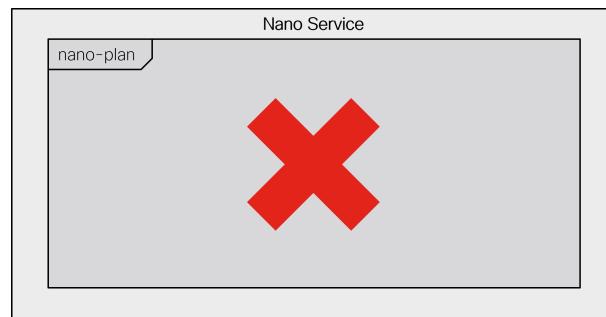
All components of a deleted service are set in backtracking mode.



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 the 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:

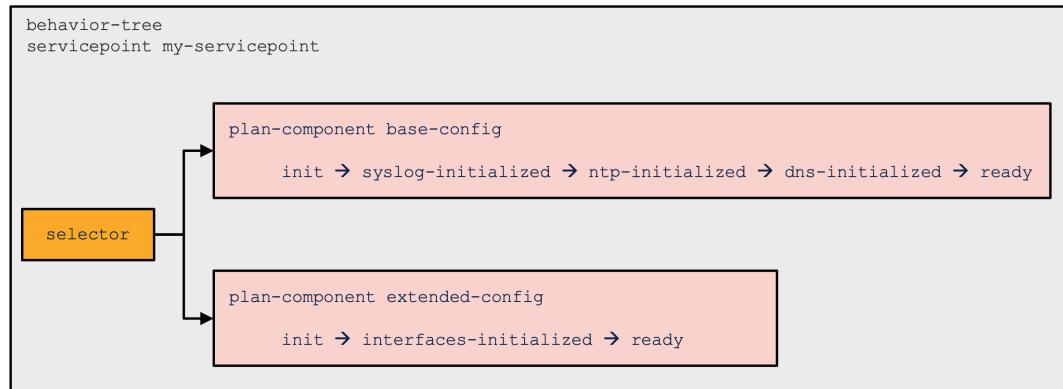
Selector A *selector* node has a set of children which are synthesized as described above.

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:

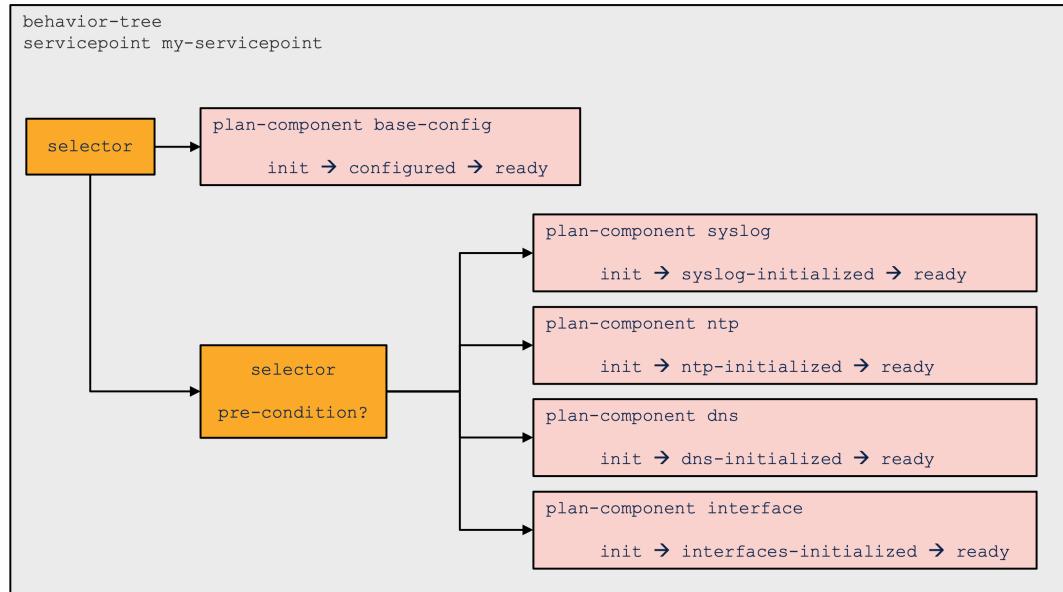
Create component The *create-component* execution node creates an instance of the component type that it refers to in the plan.

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:



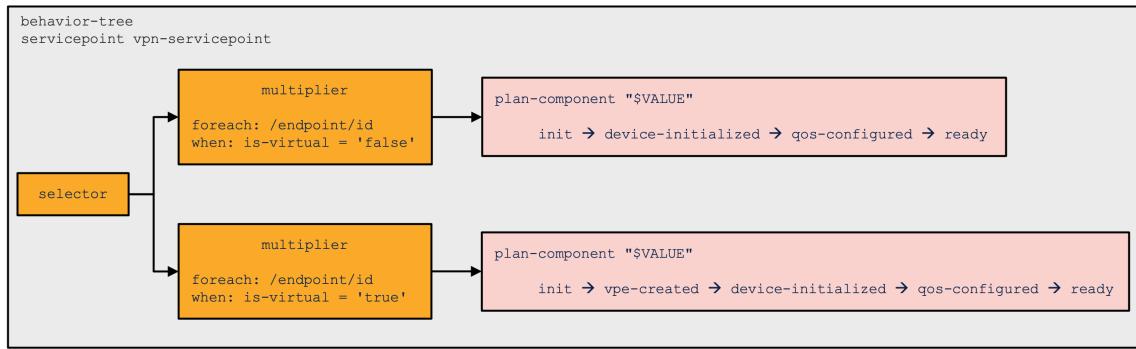
Having a selector on root implies that all plan-components are created if they don't have any pre-conditions, or for which the pre-conditions are satisfied.

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 "base-config" 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 "base-config" {
    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 more 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 "base-config" {
    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 "'base-config' {
        ncs:component-type-ref "t:base-config";
    }
    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 “base-config” 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'") )
```

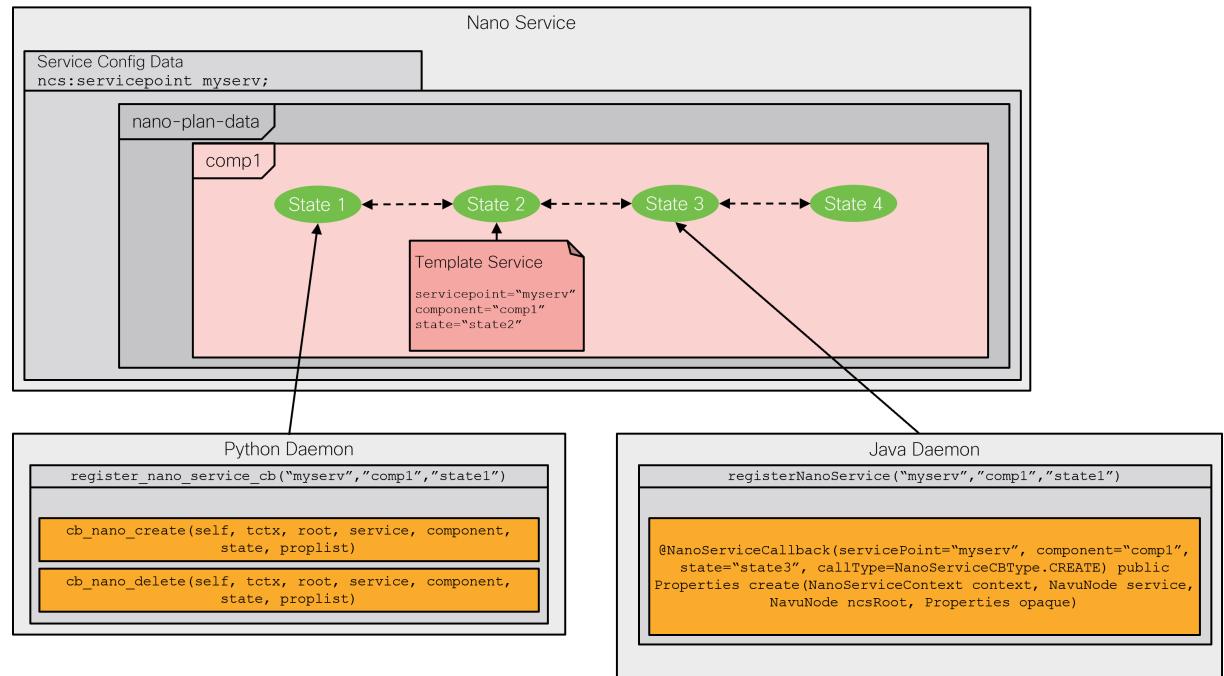
Using this scheme at an early state, such as the “base-config” component’s “ncs:init”, you can have a callback that sets name-value pairs for all other states that are then implemented solely with templates and preconditions.

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 precondition. 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()` will not fail for a nano service. But they will never be 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 "base-config" {
    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 is stored under the service path, as `ncs:plan-data` is 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. 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 227. Nano services custom plan location example

```

identity base-config {
    base ncs:plan-component-type;
}

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;
}

ncs:plan-outline custom-plan {
    description
        "Custom plan location example outline";

    ncs:component-type "p:base-config" {
        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 "'base-config'" {
            ncs:component-type-ref "p:base-config";
        }
    }
}

```

Nano Services and Commit Queue

The commit queue feature, described in the section called “Commit Queue” in *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 *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 228. 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 "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[type = 'vlan-link'][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[type = 'vlan-link'][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 vlan-link 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 229. 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: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.

The "multiplier" control node 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:

Graceful Link Migration Example

```
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
      BACK
NAME      TYPE   TRACK  GOAL  STATE    STATUS
-----  ...
self      self   false  -     init    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:

Graceful Link Migration Example

```

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  vlan-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 32

Encryption Keys

- [Introduction, page 595](#)
- [Reading encryption keys using an external command, page 595](#)

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 230. 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 33

External Logging

- [Introduction, page 597](#)
- [Enabling external log processing, page 597](#)
- [Processing logs using an external command, page 598](#)

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*.



CHAPTER 34

NSO Developer Studio

- [Introduction, page 599](#)
- [NSO Developer Studio - Developer extension, page 599](#)
- [NSO Developer Studio - Explorer extension, page 606](#)

Introduction

NSO Developer Studio provides an integrated framework for developing NSO services using Visual Studio (VS) Code extensions. The extensions come with a core feature set to help you create services and connect to running CDB instances from within the VS Code environment. The following extensions are available as part of the NSO Developer Studio:

- **NSO Developer Studio - Developer:** Used for creating NSO services. Also referred to as NSO Developer extension in this guide.
- **NSO Developer Studio - Explorer:** Used for connecting to and inspecting NSO instance. Also referred to as NSO Explorer extension in this guide.



Note

Throughout this guide, references to the VS Code GUI elements are made. It is recommended that you understand the GUI terminology before proceeding. To familiarize yourself with the VS Code GUI terminology, refer to [VS Code UX Guidelines](#).

CodeLens is a VS Code feature to facilitate performing inline contextual actions. See [Extensions using CodeLens](#) for more information.

Contribute

If you feel certain code snippets would be helpful or would like to help contribute to enhancing the extension, please get in touch: jwycoff@cisco.com

NSO Developer Studio - Developer extension

This section describes the installation and functionality of the NSO Developer extension.

The purpose of the NSO Developer extension is to provide a base framework for developers to create their own NSO services. The focus of this guide is to manifest creation of a simple NSO service package using the NSO Developer extension. At this time, reactive FastMAP and Nano services are not supported with this extension.

System requirements

In terms of an NSO package, the extension supports YANG, XML, and Python to bring together various elements required to create a simple service.

After the installation, you can use the extension to create services and perform additional functions described below.

System requirements

To get started with development using the NSO Developer extension, ensure that the following prerequisites are met on your system. The prerequisites are not a requirement to install the NSO Developer extension, but for NSO development after the extension is installed.

- Visual Studio Code.
- Java JDK 11 or higher.
- Python 3.9 or higher (recommended).

Install the extension

Installation of the NSO Developer extension is done via the VS Code marketplace.

To install the NSO Developer extension in your VS Code environment:

Step 1 Open VS Code and click the **Extensions** icon on the **Activity Bar**.

Step 2 Search for the extension using the keywords "nso developer studio" in the **Search Extensions in Marketplace** field.

Step 3 In the search results, locate the extension (**NSO Developer Studio - Developer**) and click **Install**.

Step 4 Wait while the installation completes. A notification at the bottom-right corner indicates that the installation has finished. After the installation, an NSO icon is added to the **Activity Bar**.

Make a new service package (Python only)

Use the **Make Package** command in VS Code to create a new Python package. The purpose of this command is to provide functionality similar to the `ncs-make-package` CLI command, that is, to create a basic structure for you to start developing a new Python service package. The `ncs-make-package` command however, comes with several additional options to create a package.

To make a new Python service package:

Step 1 In the VS Code menu, go to **View**, and choose **Command Palette**.

Step 2 In the **Command Palette**, type or pick the command **NSO: Make Package**. This brings up the **Make Package** dialog where you can configure package details.

Step 3 In the **Make Package** dialog, specify the following package details:

- **Package Name:** Name of the package.
- **Package Location:** Destination folder where the package is to be created.
- **Namespace:** Namespace of the YANG module, e.g. `http://www.cisco.com/myModule`.
- **Prefix:** The prefix to be given to the YANG module, e.g. `msp`.
- **Yang Version:** The YANG version that this module follows.

Step 4 Click **Create Package**. This creates the required package and opens up a new instance of VS Code with the newly created NSO package.

- Step 5** If the **Workspace Trust** dialog is shown, click **Yes, I Trust the Authors**.
-

Open an existing package

Use the **Open Existing Package** command to open an already existing package.

To open an existing package:

- Step 1** In the VS Code menu, go to **View**, then choose **Command Palette**.
- Step 2** In the **Command Palette**, type or pick the command **NSO: Open Existing Package**.
- Step 3** Browse for the package on your local disk and open it. This brings up a new instance of VS Code and opens the package in it.
-

Edit YANG files

Opening a YANG file for edit results in VS Code detecting syntax errors in the YANG file. The errors show up due to missing path to YANG files and can be resolved using the following procedure.

Add YANG models for Yangster

For YANG support, a third-party extension called Yangster is used. Yangster is able to resolve imports for core NSO models but requires additional configuration.

To add YANG models for Yangster:

- Step 1** Create a new file named `yang.settings` by right-clicking in the blank area of the **Explorer** view and choosing **New File** from the pop-up.
- Step 2** Locate the NSO source YANG files on your local disk and copy the path.
- Step 3** In the file `yang.settings`, enter the path in the JSON format: `{ "yangPath": "<path to Yang files>" }`, for example, `{ "yangPath": "/home/my-user-name/nsx-6.0/src/ncs/yang" }`.
-  **Note** On Microsoft Windows, make sure that the backslash (\) is escaped, e.g., "C:\\user\\folder\\src\\yang".
- Step 4** Save the file.
- Step 5** Wait while the Yangster extension indexes and parses the YANG file to resolve NSO imports. After the parsing is finished, errors in the YANG file will disappear.
-

View YANG diagram

YANG diagram is a feature provided by the Yangster extension.

To view the YANG diagram:

- Step 1**  Update the YANG file.
-



Note Pressing **Ctrl+space** brings up auto-completion where applicable.

- Step 2** Right-click anywhere in the VS Code **Editor** area and select **Open in Diagram** in the pop-up.
-

Add a new YANG module

To add a new YANG module:

-
- Step 1** In the **Explorer** view, navigate to the **yang** folder and select it.
- Step 2** Right-click on the **yang** folder and select **NSO: Add Yang Module** from the pop-up menu. This brings up the **Create Yang Module** dialog where you can configure module details.
- Step 3** In the **Create Yang Module** dialog, fill in the following details:
- **Module Name:** Name of the module.
 - **Namespace:** Namespace of the module, e.g., `http://www.cisco.com/myModule`.
 - **Prefix:** Prefix for the YANG module.
 - **Yang Version:** Version of YANG for this module.
- Step 4** Click **Finish**. This creates and opens up the newly created module.
-

Add a service point

Often while working on a package, there is a requirement to create a new service. This usually involves adding a service point. Adding a service point also requires other parts of the files to be updated, for example, Python.

Service points are usually added to lists.

To add a service point:

-
- Step 1** Update your YANG model as required. The extension automatically detects the list elements and displays a CodeLens called **Add Service Point**. An example is shown below.

```
container users {
    list user {
        key "name";
        description
            "This is a list of users in the system.";
        leaf name {
            type string;
        }
        leaf type {
            type string;
        }
        leaf full-name {
            type string;
        }
    }
}
```

- Step 2** Click the **Add Service Point** CodeLens. This brings up the **Add Service Point** dialog.
- Step 3** Fill in the **Service Point ID** that is used to identify the service point, for example, `mySimpleService`.
- Step 4** Next, in the **Python Details** section, select using the **Python Module** field if you want to create a new Python module or use an existing one.
- If you opt to create a new Python file, relevant sections are automatically updated in `package-metadata.xml`.

- If you select an existing Python module from the list, it is assumed that you are selecting the correct module and that, it has been created correctly, i.e., the package-meta-data.xml file is updated with the component definition.

Step 5 Enter the **Service CB Class**, for example, SimpleServiceCB.

Step 6 Finish creating the service by clicking **Add Service Point**.

Register an action point

All action points in a YANG model must be registered in NSO. Registering an action point also requires other parts of the files to be updated, for example, Python (`register_action`), and update package-meta-data if needed.

Action points are usually defined to lists or containers.

To register an action point:

Step 1 Update your YANG model as required. The extension automatically detects the action point elements in YANG and displays a CodeLens called **Add Action Point**. An example is shown below.

```
...
container server {
    tailf:action ping {
        tailf:actionpoint pingaction;
        input {
            leaf destination {
                type inet:ip-address;
            }
        }
        output {
            leaf packet-loss {
                type uint8;
            }
        }
    }
}
```



Note Note that it is mandatory to specify `tailf:actionpoint <actionpointname>` under `tailf:action <actionname>`. This is a known limitation.



Note The action point CodeLens at this time only works for the `tailf:action` statement, and not for the YANG `rpc` or YANG 1.1 action statements.

Step 2 Click the **Add Action Point** CodeLens. This brings up the **Register Action Point** dialog.

Step 3 Next, in the **Python Details** section, select using the **Python Module** field if you want to create a new Python module or use an existing one.

- If you opt to create a new Python file, relevant sections are automatically updated in package-meta-data.xml.
- If you select an existing Python module from the list, it is assumed that you are selecting the correct module, and that it has been created correctly, i.e., the package-meta-data.xml file is updated with the component definition.

- Step 4** Enter the action class name in the **Main Class name used as entry point** field, for example, `MyAction`.
- Step 5** Finish by clicking **Register Action Point**.

Edit Python files

Opening a Python file uses the Microsoft Pylance extension. This extension provides syntax highlighting and other features such as code completion.



Note

To resolve NCS import errors with the Pylance extension, you need to configure the path to NSO Python API in VS Code settings. To do this, go to VS Code **Preferences > Settings** and type `python.analysis.extraPaths` in the **Search settings** field. Next, click **Add Item**, and enter the path to NSO Python API, for example, `/home/my-user-name/nsos-6.0/src/ncs/pyapi`. Press **OK** when done.

Add a new Python module

To add a new Python module:

- Step 1** In the **Primary Sidebar, Explorer** view, right-click on the `python` folder.
- Step 2** Select **NSO: Add Python Module** from the pop-up. This brings up the **Create Python Module** dialog.
- Step 3** In the **Create Python Module** dialog, fill in the following details:
- **Module Name:** Name of the module, for example, `MyServicePackage.service`.
 - **Component Name:** Name of the component that will be used to identify this module, for example, `service`.
 - **Class Name:** Name of the class to be invoked, for example, `Main`.
- Step 4** Click **Finish**.

Use Python code completion snippets

Pre-defined snippets in VS Code allow for NSO Python code completion.

To use a Python code completion snippet:

- Step 1** Open a Python file for editing.
- Step 2** Type in one of the following pre-defined texts to display snippet options:
- `maapi`: to view options for creating a `maapi` write transaction.
 - `ncs`: to view options for snippet for `ncs` template and variables.
- Step 3** Select a snippet from the pop-up to insert its code. This also highlights config items that can be changed. Press the **Tab** key to cycle through each value.

Edit XML template files

The final part of a typical service development is creating and editing the XML configuration template.

Add a new XML template

To add a new XML template:

-
- Step 1** In the **Primary Sidebar**, **Explorer** view, right-click on the **templates** folder.
- Step 2** Select **NSO: Add XML Template** from the pop-up. This brings up the **Add XML Template** dialog.
- Step 3** In the **Add XML Template** dialog, fill in the **XML Template** name, for example, `mspSimpleService`.
- Step 4** Click **Finish**.
-

Use XML code completion snippets

Pre-defined snippets in VS Code allow for NSO XML code completion of processing instructions and variables.

To use an XML code completion snippet:

-
- Step 1** Open an XML file for editing.
- Step 2** Type in one of the following pre-defined texts to display snippet options:
- For processing instructions: `<?` followed by a character, for example `<?i` to view snippets for an `if` statement. All supported processing instructions are available as snippets.
 - For variables: `$` followed by character(s) matching the variable name, for example, `$VA` to view variable snippet. Variables defined in the XML template via the `<?set` processing instruction or defined in Python code are displayed.
-  **Note** Auto-completion can also be triggered by pressing the **Ctrl+Space** keys.
- Step 3** Select an option from the pop-up to insert the relevant XML processing instruction or variable. Items that require further configuration are highlighted. Press the **Tab** key to cycle through the items.
-

XML code validation

The NSO Developer extension also performs code validation wherever possible. The following warning and error messages are shown if the extension is unable to validate code:

- A warning is shown if a user enters a variable in an XML template that is not detected by the NSO Developer extension.
- An error message is shown if the ending tags in a processing instruction do not match.

Limitations

The extension provides help on a best-effort basis by showing error messages and warnings wherever possible. Still, in certain situations code validation is not possible. An example of such a limitation is when the extension is not able to detect a template variable that is defined elsewhere and passed indirectly (i.e., the variable is not directly called).

Consider the following code for example, where the extension will successfully detect that a template variable `IP_ADDRESS` has been set.

```
vars.add('IP_ADDRESS', '192.168.0.1')
```

Now consider the following code. While it serves the same purpose, it will fail to be detected.

```
ip_add_var_name = 'IP_ADDRESS' vars.add(ip_add_var_name, '192.168.0.1')
```

NSO Developer Studio - Explorer extension

This section describes the installation and functionality of the NSO Explorer extension.

The purpose of the NSO Explorer extension is to allow the user to connect to running instance of NSO and navigate the CDB from within VS Code.

System requirements

To get started with the NSO Explorer extension, ensure that the following prerequisites are met on your system. The prerequisites are not a requirement to install the NSO Explorer extension, but for NSO development after the extension is installed.

- Visual Studio Code.
- Java JDK 11 or higher.
- Python 3.9 or higher (recommended).

Install the extension

Installation of the NSO Explorer extension is done via the VS Code marketplace.

To install NSO Explorer extension in your VS Code environment:

Step 1 Open VS Code and click the **Extensions** icon on the **Activity Bar**.

Step 2 Search for the extension using the keywords "nso developer studio" in the **Search Extensions in Marketplace** field.

Step 3 In the search results, locate the extension (**NSO Developer Studio - Explorer**) and click **Install**.

Step 4 Wait while the installation completes. A notification at the bottom-right corner indicates that the installation has finished. After the installation, an NSO icon is added to the **Activity Bar**.

Connect to NSO instance

The NSO Explorer extension allows you to connect to and inspect a live NSO instance from within the VS Code. This procedure assumes that you have not previously connected to an NSO instance.

To connect to an NSO instance:

Step 1 In the **Activity Bar**, click the **NSO** icon to open **NSO Explorer**.

Step 2 If no NSO instance is already configured, a welcome screen is displayed with an option to add a new NSO instance.

Step 3 Click the **Add NSO Instance** button to open the **Settings** editor.

Step 4 In the **Settings** editor, click the link **Edit in settings.json**. This opens the `settings.json` file for editing.

Step 5 Next, edit the `settings.json` file as shown below:

```
"NSO.Instance": [
  {
    "host": "<hostname/ip>",
    "port": "<port>",
    "scheme": "http|https",
    "username": "<username>",
    "password": "<password>"
  }
]
```

Step 6 Save the file when done.

If settings have been configured correctly, NSO Explorer will attempt to connect to running NSO instance and display the NSO configuration.

Inspect the CDB tree

Once the NSO Explorer extension is configured, the user can inspect the CDB tree.

To inspect the CDB tree, use the following functions:

- **Get Element Info:** Click the **i** (info) icon on the **Explorer** bar, or alternatively inline next to an element in the **Explorer** view.
- **Copy KeyPath:** Click the **{ KP }** icon to copy the keypath for the selected node.
- **Copy XPath:** Click the **{ XP }** icon to copy the XPath for the selected node.
- **Get XML Config:** Click the **XML** icon to retrieve the XML configuration for the selected node and copy it to clipboard.

If data has changed in NSO, click the refresh button at the top of the **Explorer** pane to fetch it.

Inspect the CDB tree