



Administration 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

Introduction 1

CHAPTER 2

NSO System Management 3

Introduction 3

Configuring NSO 3

Overview 3

Configuration file 3

Exposed interfaces 4

Dynamic configuration 4

Built-in or external SSH server 5

Starting NSO 5

Licensing NSO 5

Monitoring NSO 5

NSO status 5

Monitoring the NSO daemon 7

Logging 7

syslog 8

Log messages and formats 9

Trace ID 28

Backup and restore 28

Backup 28

NSO Restore 28

Disaster management 29

NSO fails to start 29

NSO failure after startup 30

Transaction commit failure 30

Troubleshooting 31

Installation Problems 31

Problems Starting NSO	31
Problems Running Examples	32
Problems Using and Developing Services	32
General Troubleshooting Strategies	32

CHAPTER 3

Cisco Smart Licensing	35
Introduction	35
Smart Accounts and Virtual Accounts	35
Request a Smart Account	35
Adding users to a Smart Account	37
Create a License Registration Token	38
Notes on Configuring Smart Licensing	41
Validation and Troubleshooting	41
Available Show Commands	41
Available Show Commands	41

CHAPTER 4

NSO Alarms	43
Overview	43
Alarm type structure	43
Alarm type descriptions	44

CHAPTER 5

NSO Packages	53
Package Overview	53
Loading Packages	54
Redeploying Packages	55
Adding NED Packages	56
Managing Packages	56
Actions	57

CHAPTER 6

Advanced Topics	59
Locks	59
Global locks	59
Transaction locks	60
Northbound agents and global locks	60
External data providers	60
CDB	60

Lock impact on user sessions	61
Compaction	61
Automatic Compaction	61
Manual Compaction	61
Delayed Compaction	62
IPC ports	62
Restricting access to the IPC port	63
Restart strategies for service manager	63
Security issues	63
Running NSO as a non privileged user	65
Using IPv6 on northbound interfaces	65
CHAPTER 7	High Availability 67
Introduction to NSO High Availability	67
NSO HA Raft	68
Overview of Raft Operation	69
Node Names and Certificates	70
Recipe for a Self-signed CA	71
Network and <code>ncs.conf</code> Prerequisites	72
Connected Nodes and Node Discovery	72
Initial Cluster Setup	73
Cluster Management	75
Migrating From Existing Rule-based HA	75
Security Considerations	77
Packages Upgrades in Raft Cluster	77
Version Upgrade of Cluster Nodes	78
NSO Rule-based HA	79
Prerequisites	79
HA Member configuration	79
HA Roles	80
Failover	80
Startup	82
Actions	83
Status Check	84
Tail-f HCC Package	84

Overview	84
Dependencies	84
Running the HCC Package with NSO as a Non-Root User	85
Tail-f HCC Compared with HCC Version 4.x and Older	85
Upgrading	86
Layer-2	86
Layer-3 BGP	87
Layer-3 DNS update	89
Usage	91
Data Model	94
Setup with an External Load Balancer	99
NB listen addresses on HA primary for Load Balancers	102
HA framework requirements	102
Mode of operation	103
Security aspects	105
API	105
Ticks	105
Relay secondaries	106
CDB replication	107

CHAPTER 8
Rollbacks 109

Introduction	109
Configuration	109

CHAPTER 9
The AAA infrastructure 111

The problem	111
Structure - data models	111
Data model contents	112
AAA related items in ncs.conf	112
Authentication	114
Public Key Login	115
Password Login	116
PAM	117
External authentication	118
External token validation	120

External multi factor authentication	122
Package Authentication	124
Restricting the IPC port	127
Group Membership	127
Authorization	128
Command authorization	129
Rpc, notification, and data authorization	132
NACM Rules and Services	137
Device Group Authorization	138
Authorization Examples	140
The AAA cache	143
Populating AAA using CDB	143
Hiding the AAA tree	144

CHAPTER 10

Upgrade	145
Preparing for Upgrade	145
Single Instance Upgrade	147
Recover from Failed Upgrade	148
NSO HA Version Upgrade	149
Package Upgrade	153
Patch Management	155

CHAPTER 11

Deployment Example	157
Initial NSO Installation	158
The <code>ncs.conf</code> Configuration	161
The <code>aaa_init.xml</code> Configuration	162
The High Availability and VIP Configuration	163
Global Settings and Timeouts	164
Cisco Smart Licensing	164
Log Management	164
Log Rotate	164
Syslog	164
Audit Network Log and NED Traces	165
Python Logs	165
Java Logs	165

Internal NSO Log	165
Monitoring the Installation	165
Alarms	166
Metric - Counters, Gauges and Rate of Change Gauges	166
Security Considerations	167

CHAPTER 12

Administration 169

User Management	169
Packages	170
Adding and upgrading a package	171
Simulating the new device	172
Adding the new devices to NSO	172
Configuring NSO	173
ncs.conf	173
Run-time configuration	173
Monitoring NSO	173
Backup and Restore	173
Backup	174
NSO Restore	174

CHAPTER 13

Containerized NSO 175

Overview of NSO Images	175
Production Image	176
Development Image	176
Downloading and Extracting the Images	176
System Requirements	177
Administrative Information	177
Migration to Containerized NSO	177
ncs.conf File Configuration and Preference	179
Pre- and Post-Start Scripts	179
Admin User Creation	179
Exposing Ports	180
Backup and Restore	180
SSH Host Key	181
HTTPS TLS Certificate	181

YANG Model Changes (destructive)	181
Health Check	182
NSO System Dump and Disable Memory Overcommit	182
Startup Arguments	182
Examples	183
Running the Production Image using Docker CLI	183
Upgrading NSO using Docker CLI	185
Running the NSO Images using Docker Compose	186
Upgrading NSO using Docker Compose	190
 CHAPTER 14	
NED Administration	193
Introduction	193
Types of NED Packages	193
CLI NED	194
Generic NED	195
NED Settings	197
Purpose of NED ID	197
NED Versioning Scheme	198
NED Installation in NSO	199
Local Install of NED in NSO	199
System Install of Cisco-provided NED in NSO	200
Configuring a device with the new Cisco-provided NED	201
CLI NED Setup	201
Cisco-provided Generic NED Setup	202
Managing Cisco-provided third Party YANG NEDs	203
Downloading with the NED Built-in Download Tool	203
Rebuilding NED with Downloaded YANG Files	206
Reloading the NED Package into NSO	207
Rebuilding the NED with a Unique NED ID	207
Upgrading a Cisco-provided Third Party YANG NED to a Newer Version	209
NED Migration	211
Revision Merge Functionality	213



CHAPTER 1

Introduction

Cisco Network Service Orchestrator (NSO) version 6.2.5 is an evolution of the Tail-f Network Control System (NCS). Tail-f was acquired by Cisco in 2014. The product has been enhanced and forms the base for Cisco NSO. Note that the terms 'ncs' and 'tail-f' are used extensively in file names, command-line command names, YANG models, application programming interfaces (API), etc. Throughout this document we will use NSO to mean the product, which consists of a number of tools and executables. These executable components will be referred to by their command line name, e.g. **ncs**, **ncs-netsim**, **ncs_cli**, etc.



CHAPTER 2

NSO System Management

- [Introduction, page 3](#)
- [Configuring NSO, page 3](#)
- [Starting NSO, page 5](#)
- [Licensing NSO, page 5](#)
- [Monitoring NSO, page 5](#)
- [Backup and restore, page 28](#)
- [Disaster management, page 29](#)
- [Troubleshooting, page 31](#)

Introduction

Cisco Network Service Orchestrator enabled by Tail-f (NSO) version 6.2.5 is an evolution of the Tail-f Network Control System (NCS). Tail-f was acquired by Cisco in 2014. The product has been enhanced and forms the base for Cisco NSO. Note that the 'ncs' and 'tail-f' terms are used extensively in file names, command-line command names, YANG models, application programming interfaces (API), etc. Throughout this document we will use NSO to mean the product, which consists of a number of modules and executable components. These executable components will be referred to by their command line name, e.g. **ncs**, **ncs-netsim**, **ncs_cli**, etc. **ncs** is used to refer to the executable, the running daemon.

Configuring NSO

Overview

NSO is configured in two different ways. Its configuration file, `ncs.conf`, and also through whatever data that is configured at run-time over any northbound, for example turning on trace using the CLI.

Configuration file

The `ncs.conf` file is described by the the section called “CONFIGURATION PARAMETERS” in *Manual Pages* manual page. There is a large number of configuration items in `ncs.conf`, most of them have sane default values. The `ncs.conf` file is an XML file that must adhere to the `tailf-ncs-config.yang` model. If we start the NSO daemon directly we must provide the path to the `ncs` config file as in:

```
# ncs -c /etc/ncs/ncs.conf
```

However in a "system install", the init script must be used to start NSO, and it will pass the appropriate options to the **ncs** command. Thus NSO is started with the command:

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

It is possible to edit the `ncs.conf` file, and then tell NSO to reload the edited file without restarting the daemon as in:

```
# ncs --reload
```

This command also tells NSO to close and reopen all log files, which makes it suitable to use from a system like **logrotate**.

In this section some of the important configuration settings will be described and discussed.

Exposed interfaces

NSO allows access through a number of different interfaces, depending on the use case. In the default configuration, clients can access the system locally through an unauthenticated IPC socket (with the **ncs*** family of commands, port 4569) and plain (non-HTTPS) HTTP web server (port 8080). Additionally, the system enables remote access through SSH-secured NETCONF and CLI (ports 2022 and 2024).

We strongly encourage you to review and customize the exposed interfaces to your needs in the `ncs.conf` configuration file. In particular, set:

- `/ncs-config/webui/match-host-name` to `true`.
- `/ncs-config/webui/server-name` to the hostname of the server.

If you decide to allow remote access to the web server, also make sure you use TLS-secured HTTPS instead of HTTP. Not doing so exposes you to security risks.



Note

Using `/ncs-config/webui/match-host-name = true` requires you to use the configured hostname when accessing the server. Web browsers do this automatically but you may need to set the `Host` header when performing requests programmatically using an IP address instead of the hostname.

To additionally secure IPC access, refer to [the section called “Restricting access to the IPC port”](#).

For more details on individual interfaces and their use, see Northbound APIs .

Dynamic configuration

In this section all settings that can be manipulated through the NSO northbound interfaces are briefly described. NSO itself has a number of built-in YANG modules. These YANG modules describe structure that is stored in CDB. Whenever we change anything under, say `/devices/device`, it will change the CDB, but it will also change the configuration of NSO. We call this dynamic config since it can be changed at will through all northbound APIs.

We summarize the most relevant parts below:

```
ncs@ncs(config)#
Possible completions:
  aaa                AAA management, users and groups
  cluster            Cluster configuration
  devices            Device communication settings
  java-vm            Control of the NCS Java VM
  nacm               Access control
```

packages	Installed packages
python-vm	Control of the NCS Python VM
services	Global settings for services, (the services themselves might be ob
session	Global default CLI session parameters
snmp	Top-level container for SNMP related configuration and status ob
snmp-notification-receiver	Configure reception of SNMP notifications
software	Software management
ssh	Global SSH connection configuration

tailf-ncs.yang

This is the most important YANG module that is used to control and configure NSO. The module can be found at: `$NCS_DIR/src/ncs/yang/tailf-ncs.yang` in the release. Everything in that module is available through the northbound APIs. The YANG module has descriptions for everything that can be configured.

`tailf-common-monitoring2.yang` and `tailf-ncs-monitoring2.yang` are two modules that are relevant to monitoring NSO.

Built-in or external SSH server

NSO has a built-in SSH server which makes it possible to SSH directly into the NSO daemon. Both NSO northbound NETCONF agent and the CLI need SSH. To configure the built-in SSH server we need a directory with server SSH keys - it is specified via `/ncs-config/aaa/ssh-server-key-dir` in `ncs.conf`. We also need to enable `/ncs-config/netconf-north-bound/transport/ssh` and `/ncs-config/cli/ssh` in `ncs.conf`. In a "system install", `ncs.conf` is installed in the "config directory", by default `/etc/ncs`, with the SSH server keys in `/etc/ncs/ssh`.

Starting NSO

When NSO is started, it reads its configuration file and starts all subsystems configured to start (such as NETCONF, CLI etc.).

By default, NSO starts in the background without an associated terminal. It is recommended to use a "system install" when installing NSO for production deployment, see the section called "System Install Steps" in *Getting Started*. This will create an init script that starts NSO when the system boots, and make NSO start the service manager.

Licensing NSO

NSO is licensed using Cisco Smart Licensing. To register your NSO instance, you need to enter a token from your Cisco Smart Software Manager account. For more information on this topic, please see [Chapter 3, Cisco Smart Licensing](#)

Monitoring NSO

This section describes how to monitor NSO. Also read the dedicated session on alarms, [the section called "Overview"](#)

NSO status

Checking the overall status of NSO can be done using the shell:

```
$ ncs --status
```

or in the CLI

```
ncs# show ncs-state
```

For details on the output see `$NCS_DIR/src/yang/tailf-common-monitoring2.yang` and

Below follows an overview of the output:

- *daemon-status* You can see the NSO daemon mode, starting, phase0, phase1, started, stopping. The phase0 and phase1 modes are schema upgrade modes and will appear if you have upgraded any data-models.
- *version* The NSO version.
- *smp* Number of threads used by the daemon.
- *ha* The High-Availability mode of the ncs daemon will show up here: secondary, primary, relay-secondary.
- *internal/callpoints* Next section is call-points. Make sure that any validation points etc are registered. (The ncs-rfs-service-hook is an obsolete call-point, ignore this one).
 - *UNKNOWN* code tries to register a call-point that does not exist in a data-model.
 - *NOT-REGISTERED* a loaded data-model has a call-point but no code has registered.

Of special interest is of course the servicepoints. All your deployed service models should have a corresponding service-point. For example:

```
servicepoints:
  id=l3vpn-servicepoint daemonId=10 daemonName=ncs-dp-6-l3vpn:L3VPN
  id=nsr-servicepoint daemonId=11 daemonName=ncs-dp-7-nsd:NSRService
  id=vm-esc-servicepoint daemonId=12 daemonName=ncs-dp-8-vm-manager-
    esc:ServiceforVMstarting
  id=vnf-catalogue-esc daemonId=13 daemonName=ncs-dp-9-vnf-catalogue-
    esc:ESCVNFCatalogueService
```

- *internal/cdb* The cdb section is important. Look for any locks. This might be a sign that a developer has taken a CDB lock without releasing it. The subscriber section is also important. A design pattern is to register subscribers to wait for something to change in NSO and then trigger an action. Reactive FASTMAP is designed around that. Validate that all expected subscribers are OK.
- *loaded-data-models* The next section shows all namespaces and YANG modules that are loaded. If you for example are missing a service model, make sure it is really loaded.
- *cli, netconf, rest, snmp, webui* All northbound agents like CLI, REST, NETCONF, SNMP etc are listed with their IP and port. So if you want to connect over REST for example, you can see the port number here.
- *patches* Lists any installed patches.
- *upgrade-mode* If the node is in upgrade mode, it is not possible to get any information from the system over NETCONF. Existing CLI sessions can get system information.

It is also important to look at the packages that are loaded. This can be done in the CLI with:

```
admin> show packages
packages package cisco-asa
package-version 3.4.0
description "NED package for Cisco ASA"
ncs-min-version [ 3.2.2 3.3 3.4 4.0 ]
directory ./state/packages-in-use/1/cisco-asa
component upgrade-ned-id
  upgrade java-class-name com.tailf.packages.ned.asa.UpgradeNedId
component ASADp
  callback java-class-name [ com.tailf.packages.ned.asa.ASADp ]
component cisco-asa
  ned cli ned-id cisco-asa
  ned cli java-class-name com.tailf.packages.ned.asa.ASANedCli
```


ned device vendor Cisco

Monitoring the NSO daemon

NSO runs following processes:

- *The daemon:* **ncs.smp**: this is the ncs process running in the Erlang VM.
- *Java VM:* **com.tailf.ncs.NcsJVMLauncher**: service applications implemented in Java runs in this VM. There are several options on how to start the Java VM, it can be monitored and started/restarted by NSO or by an external monitor. See `ncs.conf(5)` man page and the **java-vm** settings in the CLI.
- *Python VMs:* NSO packages can be implemented in Python. The individual packages can be configured to run a VM each or share Python VM. Use the **show python-vm status current** to see current threads and **show python-vm status start** to see which threads where started at startup-time.

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/Python 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.

NSO logs in `/logs` in your running directory, (depends on your settings in `ncs.conf`). You might want the log files to be stored somewhere else. See `man ncs.conf` for details on how to configure the various logs. Below follows a list of the most useful log files:

- *ncs.log*: ncs daemon log. See [the section called “Log messages and formats”](#). Can be configured to syslog.
- *ncserr.log.1*, *ncserr.log.idx*, *ncserr.log.siz*: if the NSO daemon has a problem. this contains debug information relevant for support. The content can be displayed with "`ncs --printlog ncserr.log`".
- *audit.log*: central audit log covering all northbound interfaces. See [the section called “Log messages and formats”](#) for formats. Can be configured to syslog.
- *localhost:8080.access*: all HTTP requests to the daemon. This an access log for the embedded Web server. This file adheres to the Common Log Format, as defined by Apache and others. This log is not enabled by default and is not rotated, i.e., use `logrotate(8)`. Can be configured to syslog.
- *devel.log*: developer-log is a debug log for troubleshooting user-written code. This log is enabled by default and is not rotated, i.e., use `logrotate(8)`. This log shall be used in combination with the `java-vm` or `python-vm` logs. The user code logs in the VM logs and corresponding library logs in `devel.log`. Disable this log in production systems. Can be configured to syslog. You can manage this log and set its logging level in `ncs.conf`.

```
<developer-log>
  <enabled>true</enabled>
  <file>
    <name>${NCS_LOG_DIR}/devel.log</name>
    <enabled>>false</enabled>
  </file>
  <syslog>
    <enabled>true</enabled>
  </syslog>
</developer-log>
<developer-log-level>trace</developer-log-level>
```

- *ncs-java-vm.log*, *ncs-python-vm.log*: logger for code running in Java or Python VM, for example service applications. Developers writing Java and Python code use this log (in combination with

devel.log) for debugging. Both Java and Python log levels can be set from their respective VM settings in, for example, the CLI.

```
admin@ncs(config)# python-vm logging level level-info
admin@ncs(config)# java-vm java-logging logger com.tailf.maapi level level-info
```

- *netconf.log*, *snmp.log*: log for northbound agents. Can be configured to Syslog.
- *rollbackNNNNN*: all NSO commits generates a corresponding rollback file. The maximum number of rollback files and file numbering can be configured in `ncs.conf`.
- *xpath.trace*: XPATH is used in many places, for example XML templates. This log file shows the evaluation of all XPATH expressions and can be enabled in the `ncs.conf`.

```
<xpathTraceLog>
  <enabled>true</enabled>
  <filename>${NCS_LOG_DIR}/xpath.trace</filename>
</xpathTraceLog>
```

To debug XPATH for a template, use the pipe-target debug in the CLI instead.

```
admin@ncs(config)# commit | debug template
```

- *ned-cisco-ios-xr-pe1.trace* (for example): if device trace is turned on a trace file will be created per device. The file location is not configured in `ncs.conf` but is configured when device trace is turned on, for example in the CLI.
- *admin@ncs(config)# devices device r0 trace pretty*
- *Progress trace log*: When a transaction or action is applied, NSO emits specific progress events. These events can be displayed and recorded in a number of different ways, either in CLI with the pipe-target details on a commit, or by writing it to a log file. You can read more about progress trace log in Chapter 30, *Progress Trace in Development Guide*.
- *Transaction error log*: log for collecting information on failed transactions that lead to either CDB boot error or a runtime transaction failure. The default is 'false' (disabled). More information about the log is available in the section called “CONFIGURATION PARAMETERS” in *Manual Pages* under `logs/transaction-log`.
- *Upgrade log*: log containing information about CDB upgrade. The log is enabled by default and not rotated (i.e., use logrotate). With the NSO example set, the following examples populate the log in the `logs/upgrade.log` file: `examples.ncs/development-guide/ned-upgrade/yang-revision`, `examples.ncs/development-guide/high-availability/upgrade-basic`, `examples.ncs/development-guide/high-availability/upgrade-cluster`, and `examples.ncs/getting-started/developing-with-ncs/14-upgrade-service`. More information about the log is available in the section called “CONFIGURATION PARAMETERS” in *Manual Pages* under `logs/upgrade-log`.

syslog

NSO can syslog to a local syslog. See **man ncs.conf** how to configure the syslog settings. All syslog messages are documented in “Log messages”. The `ncs.conf` also lets you decide which of the logs should go into syslog: `ncs.log`, `devel.log`, `netconf.log`, `snmp.log`, `audit.log`, `WebUI access log`. There is also a possibility to integrate with **rsyslog** to log the `ncs`, `developer`, `audit`, `netconf`, `snmp`, and `webui access` logs to syslog with facility set to `daemon` in `ncs.conf`. For reference, see the *upgrade-l2* example, located in `examples.ncs/development-guide/high-availability/hcc`.

Below follows an example of syslog configuration:

```
<syslog-config>
```

```

    <facility>daemon</facility>
</syslog-config>

<ncs-log>
  <enabled>true</enabled>
  <file>
    <name>./logs/ncs.log</name>
    <enabled>true</enabled>
  </file>
  <syslog>
    <enabled>true</enabled>
  </syslog>
</ncs-log>

```

Log messages and formats

Table 1. Syslog Messages

Symbol	Severity
Comment	
Format String	
AAA_LOAD_FAIL	CRIT
Failed to load the AAA data, it could be that an external db is misbehaving or AAA is mounted/populated badly	
"Failed to load AAA: ~s"	
ABORT_CAND_COMMIT	INFO
Aborting candidate commit, request from user, reverting configuration.	
"Aborting candidate commit, request from user, reverting configuration."	
ABORT_CAND_COMMIT_REBOOT	INFO
ConfD restarted while having a ongoing candidate commit timer, reverting configuration.	
"ConfD restarted while having a ongoing candidate commit timer, reverting configuration."	
ABORT_CAND_COMMIT_TERM	INFO
Candidate commit session terminated, reverting configuration.	
"Candidate commit session terminated, reverting configuration."	
ABORT_CAND_COMMIT_TIMER	INFO
Candidate commit timer expired, reverting configuration.	
"Candidate commit timer expired, reverting configuration."	
ACCEPT_FATAL	CRIT
ConfD encountered an OS-specific error indicating that networking support is unavailable.	
"Fatal error for accept() - ~s"	
ACCEPT_FDLIMIT	CRIT
ConfD failed to accept a connection due to reaching the process or system-wide file descriptor limit.	
"Out of file descriptors for accept() - ~s limit reached"	
AUTH_LOGIN_FAIL	INFO

Symbol	Severity
Comment	
Format String	
A user failed to log in to ConfD. "login failed via ~s from ~s with ~s: ~s"	
AUTH_LOGIN_SUCCESS	INFO
A user logged into ConfD. "logged in via ~s from ~s with ~s using ~s authentication"	
AUTH_LOGOUT	INFO
A user was logged out from ConfD. "logged out <~s> user"	
BADCONFIG	CRIT
confd.conf contained bad data. "Bad configuration: ~s:~s: ~s"	
BAD_DEPENDENCY	ERR
A dependency was not found "The dependency node '~s' for node '~s' in module '~s' does not exist"	
BAD_NS_HASH	CRIT
Two namespaces have the same hash value. The namespace hashvalue MUST be unique. You can pass the flag --nshash <value> to confdc when linking the .xso files to force another value for the namespace hash. "~s"	
BIND_ERR	CRIT
ConfD failed to bind to one of the internally used listen sockets. "~s"	
BRIDGE_DIED	ERR
ConfD is configured to start the confd_aaa_bridge and the C program died. "confd_aaa_bridge died - ~s"	
CAND_COMMIT_ROLLBACK_DONE	INFO
Candidate commit rollback done "Candidate commit rollback done"	
CAND_COMMIT_ROLLBACK_FAILURE	ERR
Failed to rollback candidate commit "Failed to rollback candidate commit due to: ~s"	
CANDIDATE_BAD_FILE_FORMAT	WARNING
The candidate database file has a bad format. The candidate database is reset to the empty database. "Bad format found in candidate db file ~s; resetting candidate"	
CANDIDATE_CORRUPT_FILE	WARNING

Symbol	Severity
Comment	
Format String	
The candidate database file is corrupt and cannot be read. The candidate database is reset to the empty database. "Corrupt candidate db file ~s; resetting candidate"	
CDB_BOOT_ERR	CRIT
CDB failed to start. Some grave error in the cdb data files prevented CDB from starting - a recovery from backup is necessary. "CDB boot error: ~s"	
CDB_CLIENT_TIMEOUT	ERR
A CDB client failed to answer within the timeout period. The client will be disconnected. "CDB client (~s) timed out, waiting for ~s"	
CDB_CONFIG_LOST	INFO
CDB found it's data files but no schema file. CDB recovers by starting from an empty database. "CDB: lost config, deleting DB"	
CDB_DB_LOST	INFO
CDB found it's data schema file but not it's data file. CDB recovers by starting from an empty database. "CDB: lost DB, deleting old config"	
CDB_FATAL_ERROR	CRIT
CDB encountered an unrecoverable error "fatal error in CDB: ~s"	
CDB_INIT_LOAD	INFO
CDB is processing an initialization file. "CDB load: processing file: ~s"	
CDB_OP_INIT	ERR
The operational DB was deleted and re-initialized (because of upgrade or corrupt file) "CDB: Operational DB re-initialized"	
CDB_UPGRADE_FAILED	ERR
Automatic CDB upgrade failed. This means that the data model has been changed in a non-supported way. "CDB: Upgrade failed: ~s"	
CGI_REQUEST	INFO
CGI script requested. "CGI: '~s' script with method ~s"	
CLI_CMD_ABORTED	INFO
CLI command aborted. "CLI aborted"	
CLI_CMD_DONE	INFO

Symbol	Severity
Comment	
Format String	
CLI command finished successfully. "CLI done"	
CLI_CMD User executed a CLI command. "CLI '~s'"	INFO
CLI_DENIED User was denied to execute a CLI command due to permissions. "CLI denied '~s'"	INFO
COMMIT_INFO Information about configuration changes committed to the running data store. "commit ~s"	INFO
COMMIT_QUEUE_CORRUPT Failed to load commit queue. ConfD recovers by starting from an empty commit queue. "Resetting commit queue due do inconsistent or corrupt data."	ERR
CONFIG_CHANGE A change to ConfD configuration has taken place, e.g., by a reload of the configuration file "ConfD configuration change: ~s"	INFO
CONFIG_TRANSACTION_LIMIT Configuration transaction limit reached, rejected new transaction request. "Configuration transaction limit of type '~s' reached, rejected new transaction request"	INFO
CONSULT_FILE ConfD is reading its configuration file. "Consulting daemon configuration file ~s"	INFO
DAEMON_DIED An external database daemon closed its control socket. "Daemon ~s died"	CRIT
DAEMON_TIMEOUT An external database daemon did not respond to a query. "Daemon ~s timed out"	CRIT
DEVEL_AAA Developer aaa log message "~s"	INFO
DEVEL_CAPI Developer C api log message	INFO

Symbol	Severity
Comment	
Format String	
"~S"	
DEVEL_CDB Developer CDB log message "~S"	INFO
DEVEL_CONFD Developer ConfD log message "~S"	INFO
DEVEL_ECONFD Developer econfd api log message "~S"	INFO
DEVEL_SLS Developer smartlicensing api log message "~S"	INFO
DEVEL_SNMPA Developer snmp agent log message "~S"	INFO
DEVEL_SNMPGW Developer snmp GW log message "~S"	INFO
DEVEL_WEBUI Developer webui log message "~S"	INFO
DUPLICATE_NAMESPACE Duplicate namespace found. "The namespace ~s is defined in both module ~s and ~s."	CRIT
DUPLICATE_PREFIX Duplicate prefix found. "The prefix ~s is defined in both ~s and ~s."	CRIT
ERRLOG_SIZE_CHANGED Notify change of log size for error log "Changing size of error log (~s) to ~s (was ~s)"	INFO
EVENT_SOCKET_TIMEOUT An event notification subscriber did not reply within the configured timeout period "Event notification subscriber with bitmask ~s timed out, waiting for ~s"	CRIT

Symbol	Severity
Comment	
Format String	
EVENT_SOCKET_WRITE_BLOCK	CRIT
Write on an event socket blocked for too long time	
"~s"	
EXEC_WHEN_CIRCULAR_DEPENDENCY	WARNING
An error occurred while evaluating a when-expression.	
"When-expression evaluation error: circular dependency in ~s"	
EXT_AUTH_2FA_FAIL	INFO
External challenge authentication failed for a user.	
"external challenge authentication failed via ~s from ~s with ~s: ~s"	
EXT_AUTH_2FA	INFO
External challenge sent to a user.	
"external challenge sent to ~s from ~s with ~s"	
EXT_AUTH_2FA_SUCCESS	INFO
An external challenge authenticated user logged in.	
"external challenge authentication succeeded via ~s from ~s with ~s, member of groups: ~s~s"	
EXTAUTH_BAD_RET	ERR
Authentication is external and the external program returned badly formatted data.	
"External auth program (user=~s) ret bad output: ~s"	
EXT_AUTH_FAIL	INFO
External authentication failed for a user.	
"external authentication failed via ~s from ~s with ~s: ~s"	
EXT_AUTH_SUCCESS	INFO
An externally authenticated user logged in.	
"external authentication succeeded via ~s from ~s with ~s, member of groups: ~s~s"	
EXT_AUTH_TOKEN_FAIL	INFO
External token authentication failed for a user.	
"external token authentication failed via ~s from ~s with ~s: ~s"	
EXT_AUTH_TOKEN_SUCCESS	INFO
An externally token authenticated user logged in.	
"external token authentication succeeded via ~s from ~s with ~s, member of groups: ~s~s"	
EXT_BIND_ERR	CRIT
ConfD failed to bind to one of the externally visible listen sockets.	
"~s"	

Symbol	Severity
Comment	
Format String	
FILE_ERROR File error "~s: ~s"	CRIT
FILE_LOAD System loaded a file. "Loaded file ~s"	DEBUG
FILE_LOAD_ERR System tried to load a file in its load path and failed. "Failed to load file ~s: ~s"	CRIT
FILE_LOADING System starts to load a file. "Loading file ~s"	DEBUG
FXS_MISMATCH A secondary connected to a primary where the fxs files are different "Fxs mismatch, secondary is not allowed"	ERR
GROUP_ASSIGN A user was assigned to a set of groups. "assigned to groups: ~s"	INFO
GROUP_NO_ASSIGN A user was logged in but wasn't assigned to any groups at all. "Not assigned to any groups - all access is denied"	INFO
HA_BAD_VSN A secondary connected to a primary with an incompatible HA protocol version "Incompatible HA version (~s, expected ~s), secondary is not allowed"	ERR
HA_DUPLICATE_NODEID A secondary arrived with a node id which already exists "Nodeid ~s already exists"	ERR
HA_FAILED_CONNECT An attempted library become secondary call failed because the secondary couldn't connect to the primary "Failed to connect to primary: ~s"	ERR
HA_SECONDARY_KILLED A secondary node didn't produce its ticks "Secondary ~s killed due to no ticks"	ERR
INTERNAL_ERROR A ConfD internal error - should be reported to support@tail-f.com.	CRIT

Symbol	Severity
Comment	
Format String	
"Internal error: ~s"	
JIT_ENABLED	INFO
Show if JIT is enabled.	
"JIT ~s"	
JSONRPC_LOG_MSG	INFO
JSON-RPC traffic log message	
"JSON-RPC traffic log: ~s"	
JSONRPC_REQUEST_ABSOLUTE_TIMEOUT	INFO
JSON-RPC absolute timeout.	
"Stopping session due to absolute timeout: ~s"	
JSONRPC_REQUEST_IDLE_TIMEOUT	INFO
JSON-RPC idle timeout.	
"Stopping session due to idle timeout: ~s"	
JSONRPC_REQUEST	INFO
JSON-RPC method requested.	
"JSON-RPC: '~s' with JSON params ~s"	
JSONRPC_WARN_MSG	WARNING
JSON-RPC warning message	
"JSON-RPC warning: ~s"	
KICKER_MISSING_SCHEMA	INFO
Failed to load kicker schema	
"Failed to load kicker schema"	
LIB_BAD_SIZES	ERR
An application connecting to ConfD used a library version that can't handle the depth and number of keys used by the data model.	
"Got connect from library with insufficient keypath depth/keys support (~s/~s, needs ~s/~s)"	
LIB_BAD_VSN	ERR
An application connecting to ConfD used a library version that doesn't match the ConfD version (e.g. old version of the client library).	
"Got library connect from wrong version (~s, expected ~s)"	
LIB_NO_ACCESS	ERR
Access check failure occurred when an application connected to ConfD.	
"Got library connect with failed access check: ~s"	
LISTENER_INFO	INFO
ConfD starts or stops to listen for incoming connections.	

Symbol	Severity
Comment	
Format String	
"~s to listen for ~s on ~s:~s"	
LOCAL_AUTH_FAIL_BADPASS	INFO
Authentication for a locally configured user failed due to providing bad password. "local authentication failed via ~s from ~s with ~s: ~s"	
LOCAL_AUTH_FAIL	INFO
Authentication for a locally configured user failed. "local authentication failed via ~s from ~s with ~s: ~s"	
LOCAL_AUTH_FAIL_NOUSER	INFO
Authentication for a locally configured user failed due to user not found. "local authentication failed via ~s from ~s with ~s: ~s"	
LOCAL_AUTH_SUCCESS	INFO
A locally authenticated user logged in. "local authentication succeeded via ~s from ~s with ~s, member of groups: ~s"	
LOGGING_DEST_CHANGED	INFO
The target logfile will change to another file "Changing destination of ~s log to ~s"	
LOGGING_SHUTDOWN	INFO
Logging subsystem terminating "Daemon logging terminating, reason: ~s"	
LOGGING_STARTED	INFO
Logging subsystem started "Daemon logging started"	
LOGGING_STARTED_TO	INFO
Write logs for a subsystem to a specific file "Writing ~s log to ~s"	
LOGGING_STATUS_CHANGED	INFO
Notify a change of logging status (enabled/disabled) for a subsystem "~s ~s log"	
LOGIN_REJECTED	INFO
Authentication for a user was rejected by application callback. "~s"	
MAAPI_LOGOUT	INFO
A maapi user was logged out. "Logged out from maapi ctx=~s (~s)"	

Symbol	Severity
Comment	
Format String	
MAAPI_WRITE_TO_SOCKET_FAIL	INFO
maapi failed to write to a socket. "maapi server failed to write to a socket. Op: ~s Ecode: ~s Error: ~s~s"	
MISSING_AES256CFB128_SETTINGS	ERR
AES256CFB128 keys were not found in confd.conf "AES256CFB128 keys were not found in confd.conf"	
MISSING_AESCFB128_SETTINGS	ERR
AESCFB128 keys were not found in confd.conf "AESCFB128 keys were not found in confd.conf"	
MISSING_DES3CBC_SETTINGS	ERR
DES3CBC keys were not found in confd.conf "DES3CBC keys were not found in confd.conf"	
MISSING_NS2	CRIT
While validating the consistency of the config - a required namespace was missing. "The namespace ~s (referenced by ~s) could not be found in the loadPath."	
MISSING_NS	CRIT
While validating the consistency of the config - a required namespace was missing. "The namespace ~s could not be found in the loadPath."	
MMAP_SCHEMA_FAIL	ERR
Failed to setup the shared memory schema "Failed to setup the shared memory schema"	
NCS_PACKAGE_AUTH_BAD_RET	ERR
Package authentication program returned badly formatted data. "package authentication using ~s program ret bad output: ~s"	
NCS_PACKAGE_AUTH_FAIL	INFO
Package authentication failed. "package authentication using ~s failed via ~s from ~s with ~s: ~s"	
NCS_PACKAGE_AUTH_SUCCESS	INFO
A package authenticated user logged in. "package authentication using ~s succeeded via ~s from ~s with ~s, member of groups: ~s~s"	
NCS_PACKAGE_CHAL_2FA	INFO
Package authentication challenge sent to a user. "package authentication challenge sent to ~s from ~s with ~s"	

Symbol	Severity
Comment	
Format String	
NCS_PACKAGE_CHAL_FAIL	INFO
Package authentication challenge failed. "package authentication challenge using ~s failed via ~s from ~s with ~s: ~s"	
NETCONF_HDR_ERR	ERR
The cleartext header indicating user and groups was badly formatted. "Got bad NETCONF TCP header"	
NETCONF	INFO
NETCONF traffic log message "~s"	
NIF_LOG	INFO
Log message from NIF code. "~s: ~s"	
NOAAA_CLI_LOGIN	INFO
A user used the --noaaa flag to confd_cli "logged in from the CLI with aaa disabled"	
NO_CALLPOINT	CRIT
ConfD tried to populate an XML tree but no code had registered under the relevant callpoint. "no registration found for callpoint ~s of type=~s"	
NO_SUCH_IDENTITY	CRIT
The fxs file with the base identity is not loaded "The identity ~s in namespace ~s refers to a non-existing base identity ~s in namespace ~s"	
NO_SUCH_NS	CRIT
A nonexistent namespace was referred to. Typically this means that a .fxs was missing from the loadPath. "No such namespace ~s, used by ~s"	
NO_SUCH_TYPE	CRIT
A nonexistent type was referred to from a ns. Typically this means that a bad version of an .fxs file was found in the loadPath. "No such simpleType '~s' in ~s, used by ~s"	
NOTIFICATION_REPLAY_STORE_FAILURE	CRIT
A failure occurred in the builtin notification replay store "~s"	
NS_LOAD_ERR2	CRIT
System tried to process a loaded namespace and failed. "Failed to process namespaces: ~s"	

Symbol	Severity
Comment	
Format String	
NS_LOAD_ERR	CRIT
System tried to process a loaded namespace and failed. "Failed to process namespace ~s: ~s"	
OPEN_LOGFILE	INFO
Indicate target file for certain type of logging "Logging subsystem, opening log file '~s' for ~s"	
PAM_AUTH_FAIL	INFO
A user failed to authenticate through PAM. "PAM authentication failed via ~s from ~s with ~s: phase ~s, ~s"	
PAM_AUTH_SUCCESS	INFO
A PAM authenticated user logged in. "pam authentication succeeded via ~s from ~s with ~s"	
PHASE0_STARTED	INFO
ConfD has just started its start phase 0. "ConfD phase0 started"	
PHASE1_STARTED	INFO
ConfD has just started its start phase 1. "ConfD phasel1 started"	
READ_STATE_FILE_FAILED	CRIT
Reading of a state file failed "Reading state file failed: ~s: ~s (~s)"	
RELOAD	INFO
Reload of daemon configuration has been initiated. "Reloading daemon configuration."	
REOPEN_LOGS	INFO
Logging subsystem, reopening log files "Logging subsystem, reopening log files"	
REST_AUTH_FAIL	INFO
Rest authentication for a user failed. "rest authentication failed from ~s"	
REST_AUTH_SUCCESS	INFO
A rest authenticated user logged in. "rest authentication succeeded from ~s , member of groups: ~s"	
RESTCONF_REQUEST	INFO
RESTCONF request	

Symbol	Severity
Comment	
Format String	
"RESTCONF: request with ~s: ~s"	
RESTCONF_RESPONSE	INFO
RESTCONF response	
"RESTCONF: response with ~s: ~s duration ~s us"	
REST_REQUEST	INFO
REST request	
"REST: request with ~s: ~s"	
REST_RESPONSE	INFO
REST response	
"REST: response with ~s: ~s duration ~s ms"	
ROLLBACK_FAIL_CREATE	ERR
Error while creating rollback file.	
"Error while creating rollback file: ~s: ~s"	
ROLLBACK_FAIL_DELETE	ERR
Failed to delete rollback file.	
"Failed to delete rollback file ~s: ~s"	
ROLLBACK_FAIL_RENAME	ERR
Failed to rename rollback file.	
"Failed to rename rollback file ~s to ~s: ~s"	
ROLLBACK_FAIL_REPAIR	ERR
Failed to repair rollback files.	
"Failed to repair rollback files."	
ROLLBACK_REMOVE	INFO
Found half created rollback0 file - removing and creating new.	
"Found half created rollback0 file - removing and creating new"	
ROLLBACK_REPAIR	INFO
Found half created rollback0 file - repairing.	
"Found half created rollback0 file - repairing"	
SESSION_CREATE	INFO
A new user session was created	
"created new session via ~s from ~s with ~s"	
SESSION_LIMIT	INFO
Session limit reached, rejected new session request.	
"Session limit of type '~s' reached, rejected new session request"	
SESSION_MAX_EXCEEDED	INFO

Symbol	Severity
Comment	
Format String	
A user failed to create a new user sessions due to exceeding sessions limits "could not create new session via ~s from ~s with ~s due to session limits"	
SESSION_TERMINATION	INFO
A user session was terminated due to specified reason "terminated session (reason: ~s)"	
SKIP_FILE_LOADING	DEBUG
System skips a file. "Skipping file ~s: ~s"	
SNMP_AUTHENTICATION_FAILED	INFO
An SNMP authentication failed. "SNMP authentication failed: ~s"	
SNMP_CANT_LOAD_MIB	CRIT
The SNMP Agent failed to load a MIB file "Can't load MIB file: ~s"	
SNMP_MIB_LOADING	DEBUG
SNMP Agent loading a MIB file "Loading MIB: ~s"	
SNMP_NOT_A_TRAP	INFO
An UDP package was received on the trap receiving port, but it's not an SNMP trap. "SNMP gateway: Non-trap received from ~s"	
SNMP_READ_STATE_FILE_FAILED	CRIT
Read SNMP agent state file failed "Read state file failed: ~s: ~s"	
SNMP_REQUIRES_CDB	WARNING
The SNMP agent requires CDB to be enabled in order to be started. "Can't start SNMP. CDB is not enabled"	
SNMP_TRAP_NOT_FORWARDED	INFO
An SNMP trap was to be forwarded, but couldn't be. "SNMP gateway: Can't forward trap from ~s; ~s"	
SNMP_TRAP_NOT_RECOGNIZED	INFO
An SNMP trap was received on the trap receiving port, but its definition is not known "SNMP gateway: Can't forward trap with OID ~s from ~s; There is no notification with this OID in the loaded models."	
SNMP_TRAP_OPEN_PORT	ERR
The port for listening to SNMP traps could not be opened.	

Symbol	Severity
Comment	
Format String	
"SNMP gateway: Can't open trap listening port ~s: ~s"	
SNMP_TRAP_UNKNOWN_SENDER	INFO
An SNMP trap was to be forwarded, but the sender was not listed in confd.conf. "SNMP gateway: Not forwarding trap from ~s; the sender is not recognized"	
SNMP_TRAP_V1	INFO
An SNMP v1 trap was received on the trap receiving port, but forwarding v1 traps is not supported. "SNMP gateway: V1 trap received from ~s"	
SNMP_WRITE_STATE_FILE_FAILED	WARNING
Write SNMP agent state file failed "Write state file failed: ~s: ~s"	
SSH_HOST_KEY_UNAVAILABLE	ERR
No SSH host keys available. "No SSH host keys available"	
SSH_SUBSYS_ERR	INFO
Typically errors where the client doesn't properly send the \"subsystem\" command. "ssh protocol subsys - ~s"	
STARTED	INFO
ConfD has started. "ConfD started vsn: ~s"	
STARTING	INFO
ConfD is starting. "Starting ConfD vsn: ~s"	
STOPPING	INFO
ConfD is stopping (due to e.g. confd --stop). "ConfD stopping (~s)"	
TOKEN_MISMATCH	ERR
A secondary connected to a primary with a bad auth token "Token mismatch, secondary is not allowed"	
UPGRADE_ABORTED	INFO
In-service upgrade was aborted. "Upgrade aborted"	
UPGRADE_COMMITTED	INFO
In-service upgrade was committed. "Upgrade committed"	

Symbol	Severity
Comment	
Format String	
UPGRADE_INIT_STARTED	INFO
In-service upgrade initialization has started.	
"Upgrade init started"	
UPGRADE_INIT_SUCCEEDED	INFO
In-service upgrade initialization succeeded.	
"Upgrade init succeeded"	
UPGRADE_PERFORMED	INFO
In-service upgrade has been performed (not committed yet).	
"Upgrade performed"	
WEB_ACTION	INFO
User executed a Web UI action.	
"WebUI action '~s'"	
WEB_CMD	INFO
User executed a Web UI command.	
"WebUI cmd '~s'"	
WEB_COMMIT	INFO
User performed Web UI commit.	
"WebUI commit ~s"	
WEBUI_LOG_MSG	INFO
WebUI access log message	
"WebUI access log: ~s"	
WRITE_STATE_FILE_FAILED	CRIT
Writing of a state file failed	
"Writing state file failed: ~s: ~s (~s)"	
XPATH_EVAL_ERROR1	WARNING
An error occurred while evaluating an XPath expression.	
"XPath evaluation error: ~s for ~s"	
XPATH_EVAL_ERROR2	WARNING
An error occurred while evaluating an XPath expression.	
"XPath evaluation error: '~s' resulted in ~s for ~s"	
COMMIT_UN_SYNCED_DEV	INFO
Data was committed toward a device with bad or unknown sync state	
"Committed data towards device ~s which is out of sync"	
NCS_DEVICE_OUT_OF_SYNC	INFO
A check-sync action reported out-of-sync for a device	

Symbol	Severity
Comment	
Format String	
"NCS device-out-of-sync Device '~s' Info '~s'"	
NCS_JAVA_VM_FAIL	ERR
The NCS Java VM failure/timeout	
"The NCS Java VM ~s"	
NCS_JAVA_VM_START	INFO
Starting the NCS Java VM	
"Starting the NCS Java VM"	
NCS_PACKAGE_BAD_DEPENDENCY	CRIT
Bad NCS package dependency	
"Failed to load NCS package: ~s; required package ~s of version ~s is not present (found ~s)"	
NCS_PACKAGE_BAD_NCS_VERSION	CRIT
Bad NCS version for package	
"Failed to load NCS package: ~s; requires NCS version ~s"	
NCS_PACKAGE_CIRCULAR_DEPENDENCY	CRIT
Circular NCS package dependency	
"Failed to load NCS package: ~s; circular dependency found"	
NCS_PACKAGE_COPYING	DEBUG
A package is copied from the load path to private directory	
"Copying NCS package from ~s to ~s"	
NCS_PACKAGE_DUPLICATE	CRIT
Duplicate package found	
"Failed to load duplicate NCS package ~s: (~s)"	
NCS_PACKAGE_SYNTAX_ERROR	CRIT
Syntax error in package file	
"Failed to load NCS package: ~s; syntax error in package file"	
NCS_PACKAGE_UPGRADE_ABORTED	CRIT
The CDB upgrade was aborted implying that CDB is untouched. However the package state is changed	
"NCS package upgrade failed with reason '~s'"	
NCS_PACKAGE_UPGRADE_UNSAFE	CRIT
Package upgrade has been aborted due to warnings.	
"NCS package upgrade has been aborted due to warnings:\n~s"	
NCS_PYTHON_VM_FAIL	ERR
The NCS Python VM failure/timeout	
"The NCS Python VM ~s"	

Symbol	Severity
Comment	
Format String	
NCS_PYTHON_VM_START	INFO
Starting the named NCS Python VM	
"Starting the NCS Python VM ~s"	
NCS_PYTHON_VM_START_UPGRADE	INFO
Starting a Python VM to run upgrade code	
"Starting upgrade of NCS Python package ~s"	
NCS_SERVICE_OUT_OF_SYNC	INFO
A check-sync action reported out-of-sync for a service	
"NCS service-out-of-sync Service '~s' Info '~s'"	
NCS_SET_PLATFORM_DATA_ERROR	ERR
The device failed to set the platform operational data at connect	
"NCS Device '~s' failed to set platform data Info '~s'"	
NCS_SMART_LICENSING_ENTITLEMENT_NOTIFICATION	INFO
Smart Licensing Entitlement Notification	
"Smart Licensing Entitlement Notification: ~s"	
NCS_SMART_LICENSING_EVALUATION_COUNTDOWN	INFO
Smart Licensing evaluation time remaining	
"Smart Licensing evaluation time remaining: ~s"	
NCS_SMART_LICENSING_FAIL	INFO
The NCS Smart Licensing Java VM failure/timeout	
"The NCS Smart Licensing Java VM ~s"	
NCS_SMART_LICENSING_GLOBAL_NOTIFICATION	INFO
Smart Licensing Global Notification	
"Smart Licensing Global Notification: ~s"	
NCS_SMART_LICENSING_START	INFO
Starting the NCS Smart Licensing Java VM	
"Starting the NCS Smart Licensing Java VM"	
NCS_SNMP_INIT_ERR	INFO
Failed to locate snmp_init.xml in loadpath	
"Failed to locate snmp_init.xml in loadpath ~s"	
NCS_SNMPM_START	INFO
Starting the NCS SNMP manager component	
"Starting the NCS SNMP manager component"	
NCS_SNMPM_STOP	INFO
The NCS SNMP manager component has been stopped	

Symbol	Severity
Comment	
Format String	
"The NCS SNMP manager component has been stopped"	
BAD_LOCAL_PASS	INFO
A locally configured user provided a bad password. "Provided bad password"	
EXT_LOGIN	INFO
An externally authenticated user logged in. "Logged in over ~s using externalauth, member of groups: ~s~s"	
EXT_NO_LOGIN	INFO
External authentication failed for a user. "failed to login using externalauth: ~s"	
NO_SUCH_LOCAL_USER	INFO
A non existing local user tried to login. "no such local user"	
PAM_LOGIN_FAILED	INFO
A user failed to login through PAM. "pam phase ~s failed to login through PAM: ~s"	
PAM_NO_LOGIN	INFO
A user failed to login through PAM "failed to login through PAM: ~s"	
SSH_LOGIN	INFO
A user logged into ConfD's builtin ssh server. "logged in over ssh from ~s with authmeth:~s"	
SSH_LOGOUT	INFO
A user was logged out from ConfD's builtin ssh server. "Logged out ssh <~s> user"	
SSH_NO_LOGIN	INFO
A user failed to login to ConfD's builtin SSH server. "Failed to login over ssh: ~s"	
WEB_LOGIN	INFO
A user logged in through the WebUI. "logged in through Web UI from ~s"	
WEB_LOGOUT	INFO
A Web UI user logged out. "logged out from Web UI"	

Trace ID

NSO can issue a unique Trace ID per northbound request, visible in logs and trace headers. This Trace ID can be used to follow the request from service invocation to configuration changes pushed to any device affected by the change. The Trace ID may either be passed in from external client or generated by NSO

Trace ID is enabled by default, and can be turned off by adding the following snippet to NSO.conf:

```
<trace-id>false</trace-id>
```

Trace ID is propagated downwards in LSA setups and is fully integrated with commit queues.

Trace ID can be passed to NSO over NETCONF, RESTCONF, JSON-RPC or CLI as a commit parameter.

If Trace ID is not given as a commit parameter, NSO will generate one if the feature is enabled. This generated Trace ID will be on the form UUID version 4.

For RESTCONF request, this generated Trace ID will be communicated back to the requesting client as a HTTP header called "X-Cisco-NSO-Trace-ID". The trace-id query parameter can also be used with RPCs and actions to relay a trace-id from northbound requests.

For NETCONF, the Trace ID will be returned as an attributed called "trace-id".

Trace ID will appear in relevant log entries and trace file headers on the form "trace-id=...".

Backup and restore

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

In a "system install" of NSO, the most convenient way to do backup and restore is to use the **ncs-backup** command. In that case the following procedure is used.

Backup

NSO Backup backs up the database (CDB) files, state files, config files and rollback files from the installation directory. To take a complete backup (for disaster recovery), use

```
# ncs-backup
```

The backup will be stored in the "run directory", by default `/var/opt/ncs`, as `/var/opt/ncs/backups/ncs-VERSION@DATETIME.backup`

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

NSO Restore

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



Note

NSO must be stopped before performing Restore.

Step 1

Stop NSO if it is running.

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

Step 2 Restore the backup.

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

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

Step 3 Start NSO.

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

Disaster management

This section describes a number of disaster scenarios and recommends various actions to take in the different disaster variants.

NSO fails to start

CDB keeps its data in four files `A.cdb`, `C.cdb`, `O.cdb` and `S.cdb`. If NSO is stopped, these four files can simply be copied, and the copy is then a full backup of CDB.

Furthermore, if neither files exists in the configured CDB directory, CDB will attempt to initialize from all files in the CDB directory with the suffix `".xml"`.

Thus, there exists two different ways to re-initiate CDB from a previous known good state, either from `.xml` files or from a CDB backup. The `.xml` files would typically be used to reinstall "factory defaults" whereas a CDB backup could be used in more complex scenarios.

If the `S.cdb` file has become inconsistent or has been removed, all commit queue items will be removed and devices not yet processed out of sync. For such an event appropriate alarms will be raised on the devices and any service instance that has unprocessed device changes will be set in the failed state.

When NSO starts and fails to initialize, the following exit codes can occur:

- Exit codes `1` and `19` mean that an internal error has occurred. A text message should be in the logs, or if the error occurred at startup before logging had been activated, on standard error (standard output if NSO was started with `--foreground --verbose`). Generally the message will only be meaningful to the NSO developers, and an internal error should always be reported to support.
- Exit codes `2` and `3` are only used for the `ncs` "control commands" (see the section **COMMUNICATING WITH NCS** in the `ncs(1)` in *Manual Pages* manual page), and mean that the command failed due to timeout. Code `2` is used when the initial connect to NSO didn't succeed within 5 seconds (or the `TryTime` if given), while code `3` means that the NSO daemon did not complete the command within the time given by the `--timeout` option.
- Exit code `10` means that one of the init files in the CDB directory was faulty in some way. Further information in the log.
- Exit code `11` means that the CDB configuration was changed in an unsupported way. This will only happen when an existing database is detected, which was created with another configuration than the current in `ncs.conf`.
- Exit code `13` means that the schema change caused an upgrade, but for some reason the upgrade failed. Details are in the log. The way to recover from this situation is either to correct the problem or to re-install the old schema (fxs) files.
- Exit code `14` means that the schema change caused an upgrade, but for some reason the upgrade failed, corrupting the database in the process. This is rare and usually caused by a bug. To recover,

either start from an empty database with the new schema, or re-install the old schema files and apply a backup.

- Exit code *15* means that `A.cdb` or `C.cdb` is corrupt in a non-recoverable way. Remove the files and re-start using a backup or init files.
- Exit code *16* means that CDB ran into an unrecoverable file-error (such as running out of space on the device while performing journal compaction).
- Exit code *20* means that NSO failed to bind a socket.
- Exit code *21* means that some NSO configuration file is faulty. More information in the logs.
- Exit code *22* indicates a NSO installation related problem, e.g. that the user does not have read access to some library files, or that some file is missing.

If the NSO daemon starts normally, the exit code is *0*.

If the AAA database is broken, NSO will start but with no authorization rules loaded. This means that all write access to the configuration is denied. The NSO CLI can be started with a flag `ncs_cli --noaaa` which will allow full unauthorized access to the configuration.

NSO failure after startup

NSO attempts to handle all runtime problems without terminating, e.g. by restarting specific components. However there are some cases where this is not possible, described below. When NSO is started the default way, i.e. as a daemon, the exit codes will of course not be available, but see the `--foreground` option in the `ncs(1)` manual page.

- Out of memory: If NSO is unable to allocate memory, it will exit by calling `abort(3)`. This will generate an exit code as for reception of the SIGABRT signal - e.g. if NSO is started from a shell script, it will see 134 as exit code (128 + the signal number).
- Out of file descriptors for `accept(2)`: If NSO fails to accept a TCP connection due to lack of file descriptors, it will log this and then exit with code 25. To avoid this problem, make sure that the process and system-wide file descriptor limits are set high enough, and if needed configure session limits in `ncs.conf`.



Note

The out-of-file descriptors issue may also manifest itself in that applications are no longer able to open new file descriptors.

In many Linux systems the default limit is 1024, but if we, for example, assume that there are 4 northbound interface ports, CLI, RESTCONF, SNMP, WebUI/JSON-RPC, or similar, plus a few hundreds of IPC ports, $x\ 1024 \approx 5120$. But one might as well use the next power of two, 8192, to be on the safe side.

Several application issues can contribute to consuming extra ports. In the scope of a NSO application that could, for example, be a script application that invokes CLI command or a callback daemon application that does not close the connection socket as they should.

A commonly used command for changing the maximum number of open file descriptors is `ulimit -n [limit]`. Commands such as `netstat` and `lsof` can be useful to debug file descriptor related issues.

Transaction commit failure

When the system is updated, NSO executes a two phase commit protocol towards the different participating databases including CDB. If a participant fails in the `commit()` phase although the participant succeeded in the prepare phase, the configuration is possibly in an inconsistent state.

When NSO considers the configuration to be in an inconsistent state, operations will continue. It is still possible to use NETCONF, the CLI and all other northbound management agents. The CLI has a different prompt which reflects that the system is considered to be in an inconsistent state and also the Web UI shows this:

```
-- WARNING -----
Running db may be inconsistent. Enter private configuration mode and
install a rollback configuration or load a saved configuration.
-----
```

The MAAPI API has two interface functions which can be used to set and retrieve the consistency status, those are `maapi_set_running_db_status()` and `maapi_get_running_db_status()` corresponding. This API can thus be used to manually reset the consistency state. The only alternative to reset the state to a consistent state is by reloading the entire configuration.

Troubleshooting

This section discusses problems that new users have seen when they started to use NSO. Please do not hesitate to contact our support team (see below) if you are having trouble, regardless of whether your problem is listed here or not. A very useful tool in that regard is the `ncs-collect-tech-report` tool, which is a Bash script that comes with the product. It collects all log files, CDB backup, and several debug dumps as a TAR file. Note that it only works with a system install.

```
root@linux:/# ncs-collect-tech-report --full
```

Installation Problems

Error messages during installation

The installation program gives a lot of error messages, the first few like the ones below. The resulting installation is obviously incomplete.

```
tar: Skipping to next header
gzip: stdin: invalid compressed data--format violated
```

Cause: This happens if the installation program has been damaged, most likely because it has been downloaded in 'ascii' mode.

Resolution: Remove the installation directory. Download a new copy of NSO from our servers. Make sure you use binary transfer mode every step of the way.

Problems Starting NSO

NSO terminating with GLIBC error

NSO terminates immediately with a message similar to the one below.

```
Internal error: Open failed: /lib/tls/libc.so.6: version
`GLIBC_2.3.4' not found (required by
.../lib/ncs/priv/util/syst_drv.so)
```

Cause: This happens if you are running on a very old Linux version. The GNU libc (GLIBC) version is older than 2.3.4, which was released 2004.

Resolution: Use a newer Linux system, or upgrade the GLIBC installation.

Problems Running Examples

The 'netconf-console' program fails

Sending NETCONF commands and queries with 'netconf-console' fails, while it works using 'netconf-console-tcp'. The error message is below.

You must install the Python SSH implementation Paramiko in order to use SSH.

Cause: The netconf-console command is implemented using the Python programming language. It depends on the Python SSHv2 implementation Paramiko. Since you are seeing this message, your operating system doesn't have the Python-module Paramiko installed.

Resolution: Install Paramiko using the instructions from

- <https://www.paramiko.org>

When properly installed, you are be able to import the Paramiko module without error messages.

```
$ python
...
>>> import paramiko
>>>
```

Exit the Python interpreter with Ctrl+D.

A workaround is to use 'netconf-console-tcp'. It uses TCP instead of SSH and doesn't require Paramiko. Note that TCP traffic is not encrypted.

Problems Using and Developing Services

If you encounter issues while loading service packages, creating service instances, or developing service models, template, and code, you can consult the Troubleshooting section in Chapter 7, *Implementing Services in Development Guide*.

General Troubleshooting Strategies

If you have trouble starting or running NSO, the examples or the clients you write, here are some troubleshooting tips.

Transcript

When contacting support, it often helps the support engineer to understand what you are trying to achieve if you copy-paste the commands, responses and shell scripts that you used to trigger the problem, together with any CLI outputs and logs produced by NSO.

Source ENV variables

If you have problems executing `ncs` commands, make sure you source the `ncsrc` script in your NSO directory (your path may be different than the one in the example if you are using a local install), which sets the required environmental variables.

```
$ source /etc/profile.d/ncs.sh
```

Log files

To find out what NSO is/was doing, browsing NSO log files is often helpful. In the examples, they are called 'devel.log', 'ncs.log', 'audit.log'. If you are working with your own system, make sure the log files are enabled in `ncs.conf`. They are already enabled in all the examples. You can read more about how to enable and inspect various logs in the [logging chapter](#)

Verify hardware resources	Both high CPU utilization and a lack of memory can negatively affect the performance of NSO. You can use commands such as <code>top</code> to examine resource utilization, and <code>free -mh</code> to see the amount of free and consumed memory. A common symptom of a lack of memory is NSO or Java-VM restarting. A sufficient amount of disk space is also required for CDB persistence and logs, so you can also check disk space with <code>df -h</code> command. In case there is enough space on disk and you still encounter ENOSPC errors, check the inode usage with <code>df -i</code> command.
Status	NSO will give you a comprehensive status of daemon status, YANG modules, loaded packages, MIBs, active user sessions, CDB locks and more, if you run <code>\$ ncs --status</code>
Check data provider	NSO status information is also available as operational data under <code>/ncs-state</code> . If you are implementing a data provider (for operational or configuration data), you can verify that it works for all possible data items using <code>\$ ncs --check-callbacks</code>
Debug dump	If you suspect you have experienced a bug in NSO, or NSO told you so, you can give Support a debug dump to help us diagnose the problem. It contains a lot of status information (including a full <code>ncs --status</code> report) and some internal state information. This information is only readable and comprehensible to the NSO development team, so send the dump to your support contact. A debug dump is created using <code>\$ ncs --debug-dump mydump1</code>
Error log	Just as in CSI on TV, it's important that the information is collected as soon as possible after the event. Many interesting traces will wash away with time, or stay undetected if there are lots of irrelevant facts in the dump. If NSO gets stuck while terminating, it can optionally create a debug dump after being stuck for 60 seconds. To enable this mechanism, set the environment variable <code>\$NCS_DEBUG_DUMP_NAME</code> to a filename of your choice.
System dump	Another thing you can do in case you suspect that you have experienced a bug in NSO, is to collect the error log. The logged information is only readable and comprehensible to the NSO development team, so send the log to your support contact. The log actually consists of a number of files called <code>ncserr.log.*</code> - make sure to provide them all. If NSO aborts due to failure to allocate memory (see the section called "Disaster management"), and you believe that this is due to a memory leak in NSO, creating one or more debug dumps as described above (before NSO aborts) will produce the most useful information for Support. If this is not possible, NSO will produce a system dump by default before aborting, unless <code>DISABLE_NCS_DUMP</code> is set. The default system dump file

System call trace

name is `ncs_crash.dump` and it could be changed by setting the environment variable `$NCS_DUMP` before starting NSO.

The dumped information is only comprehensible to the NSO development team, so send the dump to your support contact.

To catch certain types of problems, especially relating to system start and configuration, the operating system's system call trace can be invaluable. This tool is called `strace`/`ktrace`/`truss`. Please send the result to your support contact for a diagnosis. Running instructions below.

Linux:

```
# strace -f -o mylog1.strace -s 1024 ncs ...
```

BSD:

```
# ktrace -ad -f mylog1.ktrace ncs ...  
# kdump -f mylog1.ktrace > mylog1.kdump
```

Solaris:

```
# truss -f -o mylog1.truss ncs ...
```



CHAPTER 3

Cisco Smart Licensing

- [Introduction, page 35](#)
- [Smart Accounts and Virtual Accounts, page 35](#)
- [Validation and Troubleshooting, page 41](#)

Introduction

[Cisco Smart Licensing](#) is a cloud-based approach to licensing and it simplifies purchase, deployment and management of Cisco software assets. Entitlements are purchased through a Cisco account via Cisco Commerce Workspace (CCW) and are immediately deposited into a Smart Account for usage. This eliminates the need to install license files on every device. Products that are smart enabled communicate directly to Cisco to report consumption.

Cisco Smart Software Manager (CSSM) enables the management of software licenses and Smart Account from a single portal. The interface allows you to activate your product, manage entitlements, renew and upgrade software.

A functioning Smart Account is required to complete the registration process. For detailed information about CSSM, see [Cisco Smart Software Manager](#).

Smart Accounts and Virtual Accounts

A Virtual Account exists as a sub-account within the Smart Account. Virtual Accounts are a customer defined structure based on organizational layout, business function, geography or any defined hierarchy. They are created and maintained by the Smart Account administrator(s).

Visit [Cisco Cisco Software Central](#) to learn about how to create and manage Smart Accounts.

Request a Smart Account

The creation of a new Smart Account is a one-time event and subsequent management of users is a capability provided through the tool. To request a Smart Account, visit [Cisco Cisco Software Central](#) and take the following steps:

Step 1 After logging in select **Request a Smart Account** in the Administration section:



Administration

[Request a Smart Account](#)

Get a Smart Account for your organization.

[Request a Partner Holding Account](#)

Allows Cisco Partners to request a Holding Smart Account

[Manage Smart Account](#)

Modify the properties of your Smart Account and associate individual Cisco Smart Accounts with your Smart Account.

[Learn about Smart Accounts](#)

Access documentation and training.

Step 2

Select the type of Smart Account to create. There are two options: (a) Individual Smart Account requiring agreement to represent your company. By creating this Smart Account you agree to authorization to create and manage product and service entitlements, users and roles on behalf of your organization. (b) Create the account on behalf of someone else.

Create Account

Would you like to create the Smart Account now?

- ☐ Yes, I have authority to represent my company and want to create the Smart Account.
- ☒ No, the person specified below will create the account:

* Email Address:

Message to Creator:

Step 3

Provide the required domain identifier and the preferred account name:

Account Information

The Account Domain Identifier will be used to **uniquely identify the account**. It is based on the email address of the person creating the account by default and must belong to the company that will own this account. [Learn More](#)

* Account Domain Identifier: [Edit](#)

* Account Name:

Step 4

The account request will be pending an approval of the Account Domain Identifier. A subsequent email will be sent to the requester to complete the setup process:



When you press "Create Account", the account will be created and placed in a PENDING state until the person specified as Account Creator completes the account setup process. The Account Creator will receive an email containing instructions on how to do this.

[Back](#)[Create Account](#)

Adding users to a Smart Account

Smart Account user management is available in the Administration section of [Cisco Cisco Software Central](#). Take the following steps to add a new user to a Smart Account:

Step 1 After logging in Select "Manage Smart Account" in the Administration section:



Administration

[Request a Smart Account](#)

Get a Smart Account for your organization.

[Request a Partner Holding Account](#)

Allows Cisco Partners to request a Holding Smart Account

[Manage Smart Account](#)

Modify the properties of your Smart Account and associate individual Cisco Smart Accounts with your Smart Account.

[Learn about Smart Accounts](#)

Access documentation and training.

Step 2 Choose the **Users** tab:

My Smart Account

[Account Properties](#)[Virtual Accounts](#)[Users](#)[Account Agreements](#)[Event Log](#)

Step 3 Select **New User** and follow the instructions in the wizard to add a new user:

New User

STEP 1 Identify New User | STEP 2 Select Roles | STEP 3 Review and Confirm

In order to be granted access to your Smart Account, the user must have a Cisco.com ID. Begin by entering the user's Cisco.com ID or email address below to search for the user's account.

* Email or Cisco.com ID:

Cancel Back Next

Background text: Cisco Software Central, My Smart Account, Account Properties, Users, New User..., User Name, Adam Groudau, Benjamin Strickland, Burkhard Warning, James Ng, Jeffrey Smith, Joakim Grebeno, Marcus Bransell, mbransell@cisco.com, Cisco Systems, Inc., Virtual Account Administrator (1)

Create a License Registration Token

Step 1 To create a new token, log into CSSM and select the appropriate Virtual Account:

My Smart Account

[Account Properties](#) | [Virtual Accounts](#) | [Users](#) | [Account Agreements](#) | [Event Log](#)

Virtual Accounts

Virtual Account Name	Description
NSO	Tail-f

Step 2 Click on the "Smart Licenses" link to enter CSSM:

NSO

General | Users

* Name: NSO

Description: Tail-f

Current Default Virtual Account: DEFAULT

You can manage Traditional Licenses, Smart Licenses, or licenses that are part of an Enterprise License Agreement assigned to this Virtual Account.

Save Reset

Step 3 In CSSM click on "New Token...":

Smart Software Manager

[Alerts](#) | [Inventory](#) | [License Conversion](#) | [Reports](#) | [Email Notification](#) | [Satellites](#) | [Activity](#)

Virtual Account: [NSO](#)

General | Licenses | Product Instances | Event Log

Virtual Account

Description: Tail-f

Default Virtual Account: No

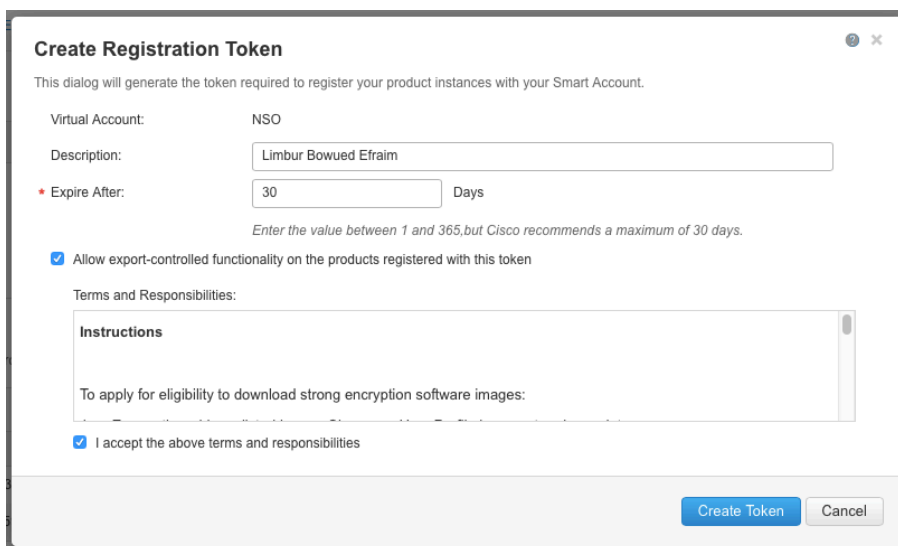
Product Instance Registration Tokens

The registration tokens below can be used to register new product instances to this virtual account.

New Token...

Token	Expiration Date	Description	Export-Controlled
YjQ2YzhINWMTYTMTMy00NzQ...	2017-Mar-29 13:30:59 (in 338 days)	testing	Allowed

Step 4 Follow the dialog to provide a description, expiration and export compliance applicability before accepting the terms and responsibilities. Click on "Create Token" to continue.



Create Registration Token

This dialog will generate the token required to register your product instances with your Smart Account.

Virtual Account: NSO

Description: Limbur Bowued Efrain

* Expire After: 30 Days

Enter the value between 1 and 365, but Cisco recommends a maximum of 30 days.

☒ Allow export-controlled functionality on the products registered with this token

Terms and Responsibilities:

Instructions

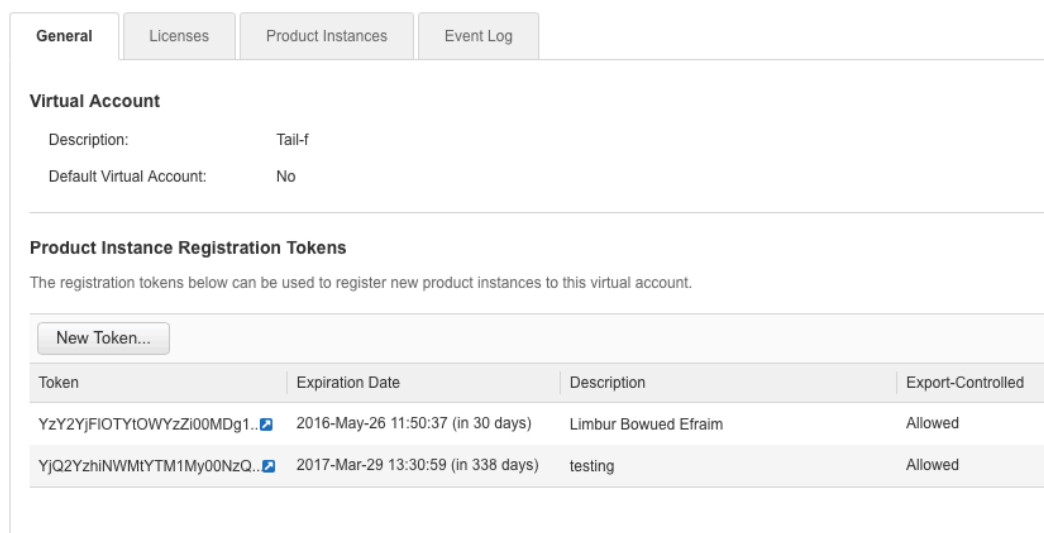
To apply for eligibility to download strong encryption software images:

☒ I accept the above terms and responsibilities

Create Token Cancel

Step 5 Click on the new token:

Virtual Account: [NSO](#)



General Licenses Product Instances Event Log

Virtual Account

Description: Tail-f

Default Virtual Account: No

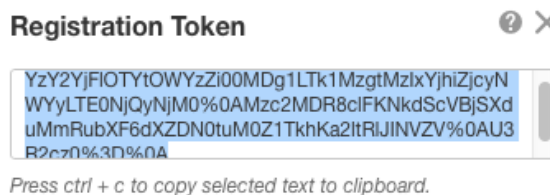
Product Instance Registration Tokens

The registration tokens below can be used to register new product instances to this virtual account.

New Token...

Token	Expiration Date	Description	Export-Controlled
YzY2YjFIOTYtOWYzZi00MDg1LTk1MzgtMzlxYjhiZjcyN...	2016-May-26 11:50:37 (in 30 days)	Limbur Bowued Efrain	Allowed
YjQ2YzhiNWMyYTM1My00NzQ..	2017-Mar-29 13:30:59 (in 338 days)	testing	Allowed

Step 6 Copy the token from the dialogue window into your clipboard:



Registration Token

YzY2YjFIOTYtOWYzZi00MDg1LTk1MzgtMzlxYjhiZjcyN
WYyLTE0NjQyNjM0%0AMzc2MDR8clFKNkdScVBjSXd
uMmRubXF6dXZDN0tuM0Z1TkhKa2ItRIJINVZV%0AU3
R2cz0%3D%0A

Press ctrl + c to copy selected text to clipboard.

Step 7 Go to the NSO CLI and provide the token to the **license smart register idtoken** command:

```
admin@ncs# license smart register idtoken YzY2YjFIOTYtOWYzZi00MDg1...
Registration process in progress.
Use the 'show license status' command to check the progress and result.
```

Notes on Configuring Smart Licensing



Note

If ncs.conf contains configuration for any of java-executable, java-options, override-url/url or proxy/url under the configure path /ncs-config/smart-license/smart-agent/ any corresponding configuration done via the CLI is ignored.



Note

The smart licensing component of NSO runs its own Java virtual machine. Usually the default Java options are sufficient:

```
leaf java-options {
  tailf:info "Smart licensing Java VM start options";
  type string;
  default "-Xmx64M -Xms16M
-Djava.security.egd=file:/dev/./urandom";
  description
    "Options which NCS will use when starting
the Java VM.";}

```

If you for some reason need to modify the Java options, remember to include the default values as found in the YANG model.

Validation and Troubleshooting

Available Show Commands

show license all	Displays all information
show license status	Displays status information
show license summary	Displays summary
show license tech	Displays license tech support information
show license usage	Displays usage information

Available Show Commands

debug smart_lic all	All available Smart Licensing debug flags
----------------------------	---



CHAPTER 4

NSO Alarms

- [Overview, page 43](#)
- [Alarm type structure, page 43](#)
- [Alarm type descriptions, page 44](#)

Overview

NSO generates alarms for serious problems that must be remedied. Alarms are available over all northbound interfaces and exist at the path **/alarms**. NSO alarms are managed as any other alarms by the general NSO Alarm Manager, see the specific section on the alarm manager in order to understand the general alarm mechanisms.

The NSO alarm manager also presents a northbound SNMP view, alarms can be retrieved as an alarm table, and alarm state changes are reported as SNMP Notifications. See the "NSO Northbound" documentation on how to configure the SNMP Agent.

This is also documented in the example `/examples.ncs/getting-started/using-ncs/5-snmpp-alarm-northbound`.

Alarm type structure

```
alarm-type
  ha-alarm
    certificate-expiration
    ha-node-down-alarm
    ha-primary-down
    ha-secondary-down
  ncs-cluster-alarm
    cluster-subscriber-failure
  ncs-dev-manager-alarm
    abort-error
    bad-user-input
    commit-through-queue-blocked
    commit-through-queue-failed
    commit-through-queue-failed-transiently
    commit-through-queue-rollback-failed
    configuration-error
    connection-failure
    final-commit-error
    missing-transaction-id
    ned-live-tree-connection-failure
```

```

    out-of-sync
    revision-error
  ncs-package-alarm
    package-load-failure
    package-operation-failure
  ncs-service-manager-alarm
    service-activation-failure
  ncs-snmp-notification-receiver-alarm
    receiver-configuration-error
  time-violation-alarm
    transaction-lock-time-violation

```

Alarm type descriptions

Table 2. Alarm type descriptions (alphabetically)

Alarm Identity	Initial Perceived Severity
abort-error	major
Description	Recommended Action
An error happened while aborting or reverting a transaction. Device's configuration is likely to be inconsistent with the NCS CDB.	Inspect the configuration difference with compare-config, resolve conflicts with sync-from or sync-to if any.
Alarm message(s)	
<ul style="list-style-type: none"> • Device {dev} is locked • Device {dev} is southbound locked • abort error 	
Clear condition(s)	
If NCS achieves sync with the device, or receives a transaction id for a netconf session towards the device, the alarm is cleared.	
Alarm Identity	
alarm-type	
Description	
Base identity for alarm types. A unique identification of the fault, not including the managed object. Alarm types are used to identify if alarms indicate the same problem or not, for lookup into external alarm documentation, etc. Different managed object types and instances can share alarm types. If the same managed object reports the same alarm type, it is to be considered to be the same alarm. The alarm type is a simplification of the different X.733 and 3GPP alarm IRP alarm correlation mechanisms and it allows for hierarchical extensions. A 'specific-problem' can be used in addition to the alarm type in order to have different alarm types based on information not known at design-time, such as values in textual SNMP Notification varbinds.	
Alarm Identity	Initial Perceived Severity
bad-user-input	critical
Description	Recommended Action
Invalid input from user. NCS cannot recognize parameters needed to connect to device.	Verify that the user supplied input are correct.
Alarm message(s)	
<ul style="list-style-type: none"> • Resource {resource} doesn't exist 	

Clear condition(s)

This alarm is not cleared.

Alarm Identity

certificate-expiration

Description	Recommended Action
The certificate is nearing its expiry or has already expired. The severity depends on the time left to expiry, it ranges from warning to critical.	Replace certificate.

Alarm message(s)

- Certificate expires in less than {days} day(s)/Certificate has expired.

Clear condition(s)

This alarm is cleared when the certificate is no longer loaded.

Alarm Identity

cluster-subscriber-failure

Initial Perceived Severity

critical

Description

Failure to establish a notification subscription towards a remote node.

Recommended Action

Verify IP connectivity between cluster nodes.

Alarm message(s)

- Failed to establish netconf notification subscription to node ~s, stream ~s
- Commit queue items with remote nodes will not receive required event notifications.

Clear condition(s)

This alarm is cleared if NCS succeeds to establish a subscription towards the remote node, or when the subscription is explicitly stopped.

Alarm Identity

commit-through-queue-blocked

Initial Perceived Severity

warning

Description

A commit was queued behind a queue item waiting to be able to connect to one of its devices. This is potentially dangerous since one unreachable device can potentially fill up the commit queue indefinitely.

Alarm message(s)

- Commit queue item ~p is blocked because item ~p cannot connect to ~s

Clear condition(s)

An alarm raised due to a transient error will be cleared when NCS is able to reconnect to the device.

Alarm Identity

commit-through-queue-failed

Initial Perceived Severity

critical

Description

A queued commit failed.

Recommended Action

Resolve with rollback if possible.

Alarm message(s)

- Failed to authenticate towards device {device}: {reason}
- Device {dev} is locked
- {Reason}
- Device {dev} is southbound locked
- Commit queue item {CqId} rollback invoked
- Commit queue item {CqId} has failed: Operation failed because: inconsistent database
- Remote commit queue item ~p cannot be unlocked: cluster node not configured correctly

Clear condition(s)

This alarm is not cleared.

Alarm Identity

commit-through-queue-failed-transiently

Initial Perceived Severity

critical

Description

A queued commit failed as it exhausted its retry attempts on transient errors.

Recommended Action

Resolve with rollback if possible.

Alarm message(s)

- Failed to connect to device {dev}: {reason}
- Connection to {dev} timed out
- Failed to authenticate towards device {device}: {reason}
- The configuration database is locked for device {dev}: {reason}
- the configuration database is locked by session {id} {identification}
- the configuration database is locked by session {id} {identification}
- {Dev}: Device is locked in a {Op} operation by session {session-id}
- resource denied
- Commit queue item {CqId} rollback invoked
- Commit queue item {CqId} has failed: Operation failed because: inconsistent database
- Remote commit queue item ~p cannot be unlocked: cluster node not configured correctly

Clear condition(s)

This alarm is not cleared.

Alarm Identity

commit-through-queue-rollback-failed

Initial Perceived Severity

critical

Description

Rollback of a commit-queue item failed.

Recommended Action

Investigate the status of the device and resolve the situation by issuing the appropriate action, i.e., service redeploy or a sync operation.

Alarm message(s)

- {Reason}

Clear condition(s)

This alarm is not cleared.

Alarm Identity configuration-error	Initial Perceived Severity critical
Description Invalid configuration of NCS managed device, NCS cannot recognize parameters needed to connect to device.	Recommended Action Verify that the configuration parameters defined in tailf-ncs-devices.yang submodule are consistent for this device.

Alarm message(s)

- Failed to resolve IP address for {dev}
- the configuration database is locked by session {id} {identification}
- {Reason}
- Resource {resource} doesn't exist

Clear condition(s)

The alarm is cleared when NCS reads the configuration parameters for the device, and is raised again if the parameters are invalid.

Alarm Identity connection-failure	Initial Perceived Severity major
Description NCS failed to connect to a managed device before the timeout expired.	Recommended Action Verify address, port, authentication, check that the device is up and running. If the error occurs intermittently, increase connect-timeout.

Alarm message(s)

- The connection to {dev} was closed
- Failed to connect to device {dev}: {reason}

Clear condition(s)

If NCS successfully reconnects to the device, the alarm is cleared.

Alarm Identity final-commit-error	Initial Perceived Severity critical
Description A managed device validated a configuration change, but failed to commit. When this happens, NCS and the device are out of sync.	Recommended Action Reconcile by comparing and sync-from or sync-to.

Alarm message(s)

- The connection to {dev} was closed
- External error in the NED implementation for device {dev}: {reason}
- Internal error in the NED NCS framework affecting device {dev}: {reason}

Clear condition(s)

If NCS achieves sync with a device, the alarm is cleared.

Alarm Identity

ha-alarm

Description

Base type for all alarms related to high availability. This is never reported, sub-identities for the specific high availability alarms are used in the alarms.

Alarm Identity

ha-node-down-alarm

Description

Base type for all alarms related to nodes going down in high availability. This is never reported, sub-identities for the specific node down alarms are used in the alarms.

Alarm Identity

ha-primary-down

Initial Perceived Severity

critical

Description

The node lost the connection to the primary node.

Recommended Action

Make sure the HA cluster is operational, investigate why the primary went down and bring it up again.

Alarm message(s)

- Lost connection to primary due to: Primary closed connection
- Lost connection to primary due to: Tick timeout
- Lost connection to primary due to: code {Code}

Clear condition(s)

This alarm is never automatically cleared and has to be cleared manually when the HA cluster has been restored.

Alarm Identity

ha-secondary-down

Initial Perceived Severity

critical

Description

The node lost the connection to a secondary node.

Recommended Action

Investigate why the secondary node went down, fix the connectivity issue and reconnect the secondary to the HA cluster.

Alarm message(s)

- Lost connection to secondary

Clear condition(s)

This alarm is cleared when the secondary node is reconnected to the HA cluster.

Alarm Identity

missing-transaction-id

Initial Perceived Severity

warning

Description

A device announced in its NETCONF hello message that it supports the transaction-id as defined in <http://tail-f.com/yang/netconf-monitoring>. However when NCS tries to read the

Recommended Action

Verify NACM rules on the concerned device.

transaction-id no data is returned. The NCS check-sync feature will not work. This is usually a case of misconfigured NACM rules on the managed device.

Alarm message(s)

- {Reason}
-

Clear condition(s)

If NCS successfully reads a transaction id for which it had previously failed to do so, the alarm is cleared.

Alarm Identity

ncs-cluster-alarm

Description

Base type for all alarms related to cluster. This is never reported, sub-identities for the specific cluster alarms are used in the alarms.

Alarm Identity

ncs-dev-manager-alarm

Description

Base type for all alarms related to the device manager This is never reported, sub-identities for the specific device alarms are used in the alarms.

Alarm Identity

ncs-package-alarm

Description

Base type for all alarms related to packages. This is never reported, sub-identities for the specific package alarms are used in the alarms.

Alarm Identity

ncs-service-manager-alarm

Description

Base type for all alarms related to the service manager This is never reported, sub-identities for the specific service alarms are used in the alarms.

Alarm Identity

ncs-snmp-notification-receiver-alarm

Description

Base type for SNMP notification receiver Alarms. This is never reported, sub-identities for specific SNMP notification receiver alarms are used in the alarms.

Alarm Identity

ned-live-tree-connection-failure

Initial Perceived Severity

major

Description

NCS failed to connect to a managed device using one of the optional live-status-protocol NEDs.

Recommended Action

Verify the configuration of the optional NEDs. If the error occurs intermittently, increase connect-timeout.

Alarm message(s)

- The connection to {dev} was closed
- Failed to connect to device {dev}: {reason}

Clear condition(s)

If NCS successfully reconnects to the managed device, the alarm is cleared.

Alarm Identity out-of-sync	Initial Perceived Severity major
Description A managed device is out of sync with NCS. Usually it means that the device has been configured out of band from NCS point of view.	Recommended Action Inspect the difference with compare-config, reconcile by invoking sync-from or sync-to.

Alarm message(s)

- Device {dev} is out of sync
- Out of sync due to no-networking or failed commit-queue commits.
- got: ~s expected: ~s.

Clear condition(s)

If NCS achieves sync with a device, the alarm is cleared.

Alarm Identity package-load-failure	Initial Perceived Severity critical
Description NCS failed to load a package.	Recommended Action Check the package for the reason.

Alarm message(s)

- failed to open file {file}: {str}
- Specific to the concerned package.

Clear condition(s)

If NCS successfully loads a package for which an alarm was previously raised, it will be cleared.

Alarm Identity package-operation-failure	Initial Perceived Severity critical
Description A package has some problem with its operation.	Recommended Action Check the package for the reason.

Clear condition(s)

This alarm is not cleared.

Alarm Identity receiver-configuration-error	Initial Perceived Severity major
Description The snmp-notification-receiver could not setup its configuration, either at startup or when reconfigured. SNMP notifications will now be missed.	Recommended Action Check the error-message and change the configuration.

Alarm message(s)

- Configuration has errors.

Clear condition(s)

This alarm will be cleared when the NCS is configured to successfully receive SNMP notifications

Alarm Identity	Initial Perceived Severity
revision-error	major
Description	Recommended Action
A managed device arrived with a known module, but too new revision.	Upgrade the Device NED using the new YANG revision in order to use the new features in the device.

Alarm message(s)

- The device has YANG module revisions not supported by NCS. Use the /devices/device/check-yang-modules action to check which modules that are not compatible.

Clear condition(s)

If all device yang modules are supported by NCS, the alarm is cleared.

Alarm Identity	Initial Perceived Severity
service-activation-failure	critical
Description	Recommended Action
A service failed during re-deploy.	Corrective action and another re-deploy is needed.

Alarm message(s)

- Multiple device errors: {str}

Clear condition(s)

If the service is successfully redeployed, the alarm is cleared.

Alarm Identity

time-violation-alarm

Description

Base type for all alarms related to time violations. This is never reported, sub-identities for the specific time violation alarms are used in the alarms.

Alarm Identity	Initial Perceived Severity
transaction-lock-time-violation	warning
Description	Recommended Action
The transaction lock time exceeded its threshold and might be stuck in the critical section. This threshold is configured in /ncs-config/transaction-lock-time-violation-alarm/timeout.	Investigate if the transaction is stuck and possibly interrupt it by closing the user session which it is attached to.

Alarm message(s)

- Transaction lock time exceeded threshold.

Clear condition(s)

This alarm is cleared when the transaction has finished.



CHAPTER 5

NSO Packages

- [Package Overview, page 53](#)
- [Loading Packages, page 54](#)
- [Redeploying Packages, page 55](#)
- [Adding NED Packages, page 56](#)
- [Managing Packages, page 56](#)

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, or a tar archive with the same directory layout. A package consists of code, YANG modules, etc., that are needed in order to add an application or function to NSO. Packages are a controlled way to manage loading and versions of custom applications.

Network Element Drivers (NEDs) are also packages. Each NED allows NSO to manage a network device of a specific type. Except for third party YANG NED package which do not contain a YANG device model by default (and must be downloaded and fixed before adding to package), a NED typically contains a device YANG model and the code, specifying how NSO should connect to the device. For NETCONF devices, NSO includes built-in tools to help you build a NED, as described in [Chapter 14, NED Administration](#), that you can use, if needed. Otherwise, a third party YANG NED, if available, should be used instead. Vendors, in some cases, provide the required YANG device models but not the entire NED. In practice, all NSO instances use at least one NED. The set of used NED packages depends of the number of different device types the NSO manages.

When NSO starts, it searches 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. On subsequent startups, NSO will by default only load and start the copied packages. The purpose of this procedure is to make it possible to reliably load new or updated packages while NSO is running, with fallback to the previously existing version of the packages if the reload should fail.

In a "system install" of NSO, packages are always installed (normally by means of symbolic links) in the `packages` subdirectory of the "run directory", i.e. by default `/var/opt/ncs/packages`, and the private directory tree is created in the `state` subdirectory, i.e. by default `/var/opt/ncs/state`.

Loading Packages

Loading of new or updated packages (as well as removal of packages that should no longer be used) can be requested via the `reload` action - from the NSO CLI:

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

This request makes NSO copy all packages found in the load path to a temporary version of its private directory, and load the packages from this directory. If the loading is successful, this temporary directory will be made permanent, otherwise the temporary directory is removed and NSO continues to use the previous version of the packages. Thus when updating packages, always update the version in the load path, and request that NSO does the reload via this action.

If the package changes include modified, added, or deleted `.fxs` files or `.ccl` files, NSO needs to run a data model upgrade procedure, also called a CDB upgrade. NSO provides a `dry-run` option to packages `reload` action to test the upgrade without committing the changes. Using `reload dry-run`, you can tell if a CDB upgrade is needed or not.

The `report all-schema-changes` option of the `reload` action instructs NSO to produce a report of how the current data model schema is being changed. Combined with a `dry-run`, the report allows you to verify the modifications introduced with the new versions of the packages before actually performing the upgrade.

For a data model upgrade, including a `dry-run`, all transactions must be closed. In particular, users having CLI sessions in configure mode must exit to operational mode. If there are ongoing commit queue items, and the `wait-commit-queue-empty` parameter is supplied, it will wait for the items to finish before proceeding the reload. During this time, it will not allow creating any new transactions. Hence, if one of the queue items fails with 'rollback-on-error' option set, the commit queue's rollback will also fail, and the queue item will be locked. In this case, the reload will be canceled. A manual investigation of the failure is needed in order to proceed with the reload.

While the data model upgrade is in progress, all transactions are closed and new transactions are not allowed. This means that starting a new management session, such as a CLI or SSH connection to the NSO, will also fail, producing an error that the node is in upgrade mode.

By default, the `reload` action will (when needed) wait up to 10 seconds for commit queue to empty (if the `wait-commit-queue-empty` parameter is entered) and reload to start.

If there are still open transactions at the end of this period, the upgrade will be canceled and the reload operation will fail. The `max-wait-time` and `timeout-action` parameters to the action can modify this behaviour. For example, to wait for up to 30 seconds, and forcibly terminate any transactions that still remain open after this period, we can invoke the action as:

```
admin@ncs# packages reload max-wait-time 30 timeout-action kill
```

Thus the default values for these parameters are 10 and `fail`, respectively. In case there are no changes to `.fxs` or `.ccl` files, the reload can be carried out without the data model upgrade procedure, and these parameters are ignored, since there is no need to close open transactions.

When reloading packages NSO will give a warning when the upgrade looks "suspicious", i.e. may break some functionality. Note that this is not a strict upgrade validation, but only intended as a hint to NSO administrator early in the upgrade process that something might be wrong. Currently the following scenarios will trigger the warnings:

- one or more namespaces are removed by the upgrade. The consequence of this is all data belonging to this namespace is permanently deleted from CDB upon upgrade. This may be intended in some scenarios, in which case it is advised to proceed overriding warnings as described below.
- there are source .java files found in the package, but no matching .class files in the jars loaded by NSO. This likely means that the package has not been compiled.
- there are matching .class files with modification time older than the source files, which hints that the source has been modified since the last time the package has been compiled. This likely means that the package was not re-compiled the last time source code has been changed.

If a warning has been triggered it is a strong recommendation to fix the root cause. If all of the warnings are intended, it is possible to proceed with "packages reload force" command.

In some specific situations upgrading a package with newly added custom validation points in the data model may produce an error similar to “no registration found for callpoint *NEW-VALIDATION/validate*” or simply “application communication failure”, resulting in an aborted upgrade. Please see the section called “New Validation Points” in *Development Guide* on how to proceed.

In some cases we may want NSO to do the same operation as the `reload` action at NSO startup, i.e. copy all packages from the load path before loading, even though the private directory copy already exists. This can be achieved in the following ways:

- Setting the shell environment variable `$NCS_RELOAD_PACKAGES` to `true`. This will make NSO do the copy from the load path on every startup, as long as the environment variable is set. In a "system install", NSO must be started via the init script, and this method must be used, but with a temporary setting of the environment variable:


```
# NCS_RELOAD_PACKAGES=true /etc/init.d/ncs start
```
- Giving the option `--with-package-reload` to the `ncs` command when starting NSO. This will make NSO do the copy from the load path on this particular startup, without affecting the behaviour on subsequent startups.
- If warnings are encountered when reloading packages at startup using one of the options above, the recommended way forward is to fix the root cause as indicated by the warnings as mentioned before. If the intention is to proceed with the upgrade without fixing the underlying cause for the warnings, it is possible to force the upgrade using `NCS_RELOAD_PACKAGES=force` environment variable or `--with-package-reload-force` option.

Always use one of these methods when upgrading to a new version of NSO in an existing directory structure, to make sure that new packages are loaded together with the other parts of the new system.

Redeploying Packages

If it is known in advance that there were no data model changes, i.e. none of the .fxs or .ccl files changed, and none of the shared JARs changed in a Java package, and the declaration of the components in the package-meta-data.xml is unchanged, then it is possible to do a lightweight package upgrade, called package redeploy. Package redeploy only loads the specified package, unlike packages reload which loads all of the packages found in the load-path.

```
admin@ncs# packages package mserv redeploy
result true
```

Redeploying a package allows to reload updated or load new templates, reload private JARs for a Java package or reload the python code which is a part of this package. Only the changed part of the package will be reloaded, e.g. if there were no changes to Python code, but only templates, then the Python VM will not be restarted, but only templates reloaded. The upgrade is not seamless however as the old templates will be unloaded for a short while before the new ones are loaded, so any user of the template during

this period of time will fail; same applies to changed Java or Python code. It is hence the responsibility of the user to make sure that the services or other code provided by package is unused while it is being redeployed.

The package `redeploy` will return `true` if the package's resulting status after the redeploy is up. Consequently, if the result of the action is `false`, then it is advised to check the operational status of the package in the package list.

```
admin@ncs# show packages package mserv oper-status
oper-status file-load-error
oper-status error-info "template3.xml:2 Unknown servicepoint: templ42-servicepoint"
```

Adding NED Packages

Unlike a full packages reload operation, new NED packages can be loaded into the system without disrupting existing transactions. This is only possible for new packages, since these packages don't yet have any instance data.

The operation is performed through the `/packages/add` action. No additional input is necessary. The operation scans all the load-paths for any new NED packages and also verifies the existing packages are still present. If packages are modified or deleted, the operation will fail.

Each NED package defines a `ned-id`, an identifier that is used in selecting the NED for each managed device. A new NED package is therefore a package with a `ned-id` value that is not already in use.

In addition, the system imposes some additional constraints, so it is not always possible to add just any arbitrary NED. In particular, NED packages can also contain one or more shared data models, such as NED settings or operational data for private use by the NED, that are not specific to each version of a NED package but rather shared between all versions. These are typically placed outside any mountpoint (device-specific data model), extending the NSO schema directly. So, if a NED defines schema nodes outside any mountpoint, there must be no changes to these nodes if they already exist.

Adding a NED package with modified shared data model is therefore not allowed and all shared data models are verified to be identical before a NED package can be added. If they are not, the `/packages/add` action will fail and you will have to use the `/packages/reload` command.

```
admin@ncs# packages add
add-result {
  package router-nc-1.1
  result true
}
```

The command returns `true` if the package's resulting status after deployment is up. Likewise, if the result for a package is `false`, then the package was added but its code has not started successfully and you should check the operational status of the package with the `show packages package PKG oper-status` command for additional information. You may then use the `/packages/package/redeploy` action to retry deploying the package's code, once you have corrected the error.



Note

In a High Availability setup, you can perform this same operation on all the nodes in the cluster with a single `packages ha sync and-add` command.

Managing Packages

In a "system install" of NSO, management of pre-built packages is supported through a number of actions. This support is not available in a "local install", since it is dependent on the directory structure created by

the "system install". Please refer to the YANG submodule `$NCS_DIR/src/ncs/yang/tailf-ncs-software.yang` for the full details of the functionality described in this section.

Actions

Actions are provided to list local packages, to fetch packages from the file system, and to install or deinstall packages:

- **software packages list [...]** List local packages, categorized into *loaded*, *installed*, and *installable*. The listing can be restricted to only one of the categories - otherwise each package listed will include the category for the package.
- **software packages fetch package-from-file *file*** Fetch a package by copying it from the file system, making it *installable*.
- **software packages install package *package-name* [...]** Install a package, making it available for loading via the **packages reload** action, or via a system restart with package reload. The action ensures that only one version of the package is installed - if any version of the package is installed already, the `replace-existing` option can be used to deinstall it before proceeding with the installation.
- **software packages deinstall package *package-name*** Deinstall a package, i.e. remove it from the set of packages available for loading.

There is also an **upload** action that can be used via NETCONF or REST to upload a package from the local host to the NSO host, making it *installable* there. It is not feasible to use in the CLI or Web UI, since the actual package file contents is a parameter for the action. It is also not suitable for very large (more than a few megabytes) packages, since the processing of action parameters is not designed to deal with very large values, and there is a significant memory overhead in the processing of such values.



CHAPTER 6

Advanced Topics

- [Locks, page 59](#)
- [Compaction, page 61](#)
- [IPC ports, page 62](#)
- [Restart strategies for service manager, page 63](#)
- [Security issues, page 63](#)
- [Running NSO as a non privileged user, page 65](#)
- [Using IPv6 on northbound interfaces, page 65](#)

Locks

This section will explain the different locks that exist in NSO and how they interact. It is important to understand the architecture of NSO with its management backplane, and the transaction state machine as described in Chapter 18, *Package Development in Development Guide* to be able to understand how the different locks fit into the picture.

Global locks

The NSO management backplane keeps a lock on the datastore: running. This lock is usually referred to as the global lock and it provides a mechanism to grant exclusive access to the datastore.

The global is the only lock that can explicitly be taken through a northbound agent, for example by the `NETCONF <lock>` operation, or by calling `Maapi.lock()`.

A global lock can be taken for the whole datastore, or it can be a partial lock (for a subset of the data model). Partial locks are exposed through `NETCONF` and `Maapi`.

An agent can request a global lock to ensure that it has exclusive write-access. When a global lock is held by an agent it is not possible for anyone else to write to the datastore the lock guards - this is enforced by the transaction engine. A global lock on running is granted to an agent if there are no other holders of it (including partial locks), and if all data providers approve the lock request. Each data provider (CDB and/or external data providers) will have its `lock()` callback invoked to get a chance to refuse or accept the lock. The output of `ncs --status` includes locking status. For each user session locks (if any) per datastore is listed.

Transaction locks

A northbound agent starts a user session towards NSO's management backplane. Each user session can then start multiple transactions. A transaction is either read/write or read-only.

The transaction engine has its internal locks towards the running datastore. These transaction locks exist to serialize configuration updates towards the datastore and are separate from the global locks.

As a northbound agent wants to update the running datastore with a new configuration it will implicitly grab and release the transactional lock. The transaction engine takes care of managing the locks, as it moves through the transaction state machine and there is no API that exposes the transactional locks to the northbound agents.

When the transaction engine wants to take a lock for a transaction (for example when entering the validate state) it first checks that no other transaction has the lock. Then it checks that no user session has a global lock on that datastore. Finally each data provider is invoked by its `transLock()` callback.

Northbound agents and global locks

In contrast to the implicit transactional locks, some northbound agents expose explicit access to the global locks. This is done a bit differently by each agent.

The management API exposes the global locks by providing `Maapi.lock()` and `Maapi.unlock()` methods (and the corresponding `Maapi.lockPartial()` `Maapi.unlockPartial()` for partial locking). Once a user session is established (or attached to) these functions can be called.

In the CLI the global locks are taken when entering different configure modes as follows:

config exclusive	The running datastore global lock will be taken.
config terminal	Does not grab any locks

The global lock is then kept by the CLI until the configure mode is exited.

The Web UI behaves in the same way as the CLI (it presents three edit tabs called "Edit private", "Edit exclusive", and which corresponds to the CLI modes described above).

The NETCONF agent translates the `<lock>` operation into a request for the global lock for the requested datastore. Partial locks are also exposed through the partial-lock rpc.

External data providers

Implementing the `lock()` and `unlock()` callbacks is not required of an external data provider.

NSO will never try to initiate the `transLock()` state transition (see the transaction state diagram in Chapter 18, *Package Development in Development Guide*) towards a data provider while a global lock is taken - so the reason for a data provider to implement the locking callbacks is if someone else can write (or lock for example to take a backup) to the data providers database.

CDB

CDB ignores the `lock()` and `unlock()` callbacks (since the data-provider interface is the only write interface towards it).

CDB has its own internal locks on the database. The running datastore has a single write and multiple read locks. It is not possible to grab the write-lock on a datastore while there are active read-locks on it. The locks in CDB exist to make sure that a reader always gets a consistent view of the data (in particular it becomes very confusing if another user is able to delete configuration nodes in between calls to `getNext()` on YANG list entries).

During a transaction `transLock()` takes a CDB read-lock towards the transactions datastore and `writeStart()` tries to release the read-lock and grab the write-lock instead.

A CDB external reader client implicitly takes a CDB read-lock between `Cdb.startSession()` and `Cdb.endSession()`. This means that while a CDB client is reading, a transaction can not pass through `writeStart()` (and conversely a CDB reader can not start while a transaction is in between `writeStart()` and `commit()` or `abort()`).

The Operational store in CDB does not have any locks. NSO's transaction engine can only read from it, and the CDB client writes are atomic per write operation.

Lock impact on user sessions

When a session tries to modify a data store that is locked in some way, it will fail. For example, the CLI might print:

```
admin@ncs(config)# commit
Aborted: the configuration database is locked
```

Since some of the locks are short lived (such as a CDB read lock), NSO is by default configured to retry the failing operation for a short period of time. If the data store still is locked after this time, the operation fails.

To configure this, set `/ncs-config/commit-retry-timeout` in `ncs.conf`.

Compaction

CDB implements write-ahead logging to provide durability in the datastores, appending a new log for each CDB transaction to the target datastore (A.cdb for configuration, O.cdb for operational, and S.cdb for snapshot datastore). Depending on the size and number of transactions towards the system, these files will grow in size leading to increased disk utilization, longer boot times, and longer initial data synchronization time when setting up a high-availability cluster. Compaction is a mechanism used to reduce the size of the write-ahead logs to a minimum. It works by replacing an existing write-ahead log, which is composed by a number of consecutive transactions logs created in run-time, with a single transaction log representing the full current state of the datastore. From this perspective, it can be seen that a compaction acts similar to a write transaction towards a datastore. To ensure data integrity, write transactions towards the datastore are not permitted during the time compaction takes place.

Automatic Compaction

By default, compaction is handled automatically by the CDB. After each transaction, CDB evaluates whether compaction is required for the affected datastore.

This is done by examining the number of added nodes as well as the file size changes since the last performed compaction. The thresholds used can be modified in the `ncs.conf` file by configuring the `/ncs-config/compaction/file-size-relative`, `/ncs-config/compaction/file-size-absolute`, and `/ncs-config/compaction/num-node-relative` settings. It is also possible to automatically trigger compaction after a set number of transactions by setting the `/ncs-config/compaction/num-transaction` property.

Manual Compaction

Compaction may require a significant amount of 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. If doing so, it is *highly recommended* to have another automated system in place.

CDB CAPI provides a set of functions which may be used to create an external mechanism for compaction. See `cdb_initiate_journal_compaction()`, `cdb_initiate_journal_dbfile_compaction()`, and `cdb_get_compaction_info()` in `confd_lib_cdb(3)` in *Manual Pages*.

Automation of compaction can be done by using a scheduling mechanism such as CRON, or by using the NCS scheduler. See Chapter 29, *Scheduler* in *Development Guide*, for more information.

By default, CDB may perform compaction during its boot process. This may be disabled if required, by starting NSO with the flag **--disable-compaction-on-start**.

Delayed Compaction

In the configuration datastore, compaction is by default delayed by 5 seconds when the threshold is reached in order to prevent any upcoming write transaction from being blocked. If the system is idle during these 5 seconds, meaning that there is no new transaction, the compaction will initiate. Otherwise, compaction is delayed by another 5 seconds. The delay time can be configured in `ncs.conf` by setting the `/ncs-config/compaction/delayed-compaction-timeout` property.

IPC ports

Client libraries connect to NSO using TCP. We tell NSO which address to use for these connections through the `/ncs-config/ncs-ipc-address/ip` (default value 127.0.0.1) and `/ncs-config/ncs-ipc-address/port` (default value 4569) elements in `ncs.conf`. It is possible to change these values, but it requires a number of steps to also configure the clients. Also there are security implications, see [the section called “Security issues”](#) below.

Some clients read the environment variables `NCS_IPC_ADDR` and `NCS_IPC_PORT` to determine if something other than the default is to be used, others might need to be recompiled. This is a list of clients which communicate with NSO, and what needs to be done when `ncs-ipc-address` is changed.

Client	Changes required
Remote commands via the <code>ncs</code> command	Remote commands, such as <code>ncs --reload</code> , check the environment variables <code>NCS_IPC_ADDR</code> and <code>NCS_IPC_PORT</code> .
CDB and MAAPI clients	The address supplied to <code>Cdb.connect()</code> and <code>Maapi.connect()</code> must be changed.
Data provider API clients	The address supplied to <code>Dp</code> constructor socket must be changed.
<code>ncs_cli</code>	The Command Line Interface (CLI) client, <code>ncs_cli</code> , checks the environment variables <code>NCS_IPC_ADDR</code> and <code>NCS_IPC_PORT</code> . Alternatively the port can be provided on the command line (using the <code>-P</code> option).
Notification API clients	The new address must be supplied to the socket for the <code>Notif</code> constructor.

To run more than one instance of NSO on the same host (which can be useful in development scenarios) each instance needs its own IPC port. For each instance set `/ncs-config/ncs-ipc-address/port` in `ncs.conf` to something different.

There are two more instances of ports that will have to be modified, NETCONF and CLI over SSH. The netconf (SSH and TCP) ports that NSO listens to by default are 2022 and 2023 respectively. Modify `/ncs-config/netconf/transport/ssh` and `/ncs-config/netconf/transport/tcp`, either by disabling them or changing the ports they listen to. The CLI over SSH by default listens to 2024; modify `/ncs-config/cli/ssh` either by disabling or changing the default port.

Restricting access to the IPC port

By default, the clients connecting to the IPC port are considered trusted, i.e. there is no authentication required, and we rely on the use of 127.0.0.1 for `/ncs-config/ncs-ipc-address/ip` to prevent remote access. In case this is not sufficient, it is possible to restrict the access to the IPC port by configuring an access check.

The access check is enabled by setting the `ncs.conf` element `/ncs-config/ncs-ipc-access-check/enabled` to "true", and specifying a filename for `/ncs-config/ncs-ipc-access-check/filename`. The file should contain a shared secret, i.e. a random character string. Clients connecting to the IPC port will then be required to prove that they have knowledge of the secret through a challenge handshake, before they are allowed access to the NSO functions provided via the IPC port.



Note

Obviously the access permissions on this file must be restricted via OS file permissions, such that it can only be read by the NSO daemon and client processes that are allowed to connect to the IPC port. E.g. if both the daemon and the clients run as root, the file can be owned by root and have only "read by owner" permission (i.e. mode 0400). Another possibility is to have a group that only the daemon and the clients belong to, set the group ID of the file to that group, and have only "read by group" permission (i.e. mode 040).

To provide the secret to the client libraries, and inform them that they need to use the access check handshake, we have to set the environment variable `NCS_IPC_ACCESS_FILE` to the full pathname of the file containing the secret. This is sufficient for all the clients mentioned above, i.e. there is no need to change application code to support or enable this check.



Note

The access check must be either enabled or disabled for both the daemon and the clients. E.g. if `/ncs-config/ncs-ipc-access-check/enabled` in `ncs.conf` is *not* set to "true", but clients are started with the environment variable `NCS_IPC_ACCESS_FILE` pointing to a file with a secret, the client connections will fail.

Restart strategies for service manager

The service manager executes in a Java VM outside of NSO. The NcsMux initializes a number of sockets to NSO at startup. These are Maapi sockets and data provider sockets. NSO can choose to close any of these sockets whenever NSO requests the service manager to perform a task, and that task is not finished within the stipulated timeout. If that happens, the service manager must be restarted. The timeout(s) are controlled by a several `ncs.conf` parameters found under `/ncs-config/japi`.

Security issues

NSO requires some privileges to perform certain tasks. The following tasks may, depending on the target system, require root privileges.

- Binding to privileged ports. The `ncs.conf` configuration file specifies which port numbers NSO should *bind(2)* to. If any of these port numbers are lower than 1024, NSO usually requires root privileges unless the target operating system allows NSO to bind to these ports as a non-root user.
- If PAM is to be used for authentication, the program installed as `$NCS_DIR/lib/ncs/priv/pam/epam` acts as a PAM client. Depending on the local PAM configuration, this program may require root privileges. If PAM is configured to read the local `passwd` file, the program must either run as root, or be `setuid` root. If the local PAM configuration instructs NSO to run for example `pam_radius_auth`, root privileges are possibly not required depending on the local PAM installation.
- If the CLI is used and we want to create CLI commands that run executables, we may want to modify the permissions of the `$NCS_DIR/lib/ncs/lib/core/confd/priv/cmdptywrapper` program.

To be able to run an executable as root or a specific user, we need to make `cmdptywrapper` `setuid` root, i.e.:

```
1 # chown root cmdptywrapper
2 # chmod u+s cmdptywrapper
```

Failing that, all programs will be executed as the user running the `ncs` daemon. Consequently, if that user is root we do not have to perform the `chmod` operations above.

The same applies for executables run via actions, but then we may want to modify the permissions of the `$NCS_DIR/lib/ncs/lib/core/confd/priv/cmdwrapper` program instead:

```
1 # chown root cmdwrapper
2 # chmod u+s cmdwrapper
```

NSO can be instructed to terminate NETCONF over clear text TCP. This is useful for debugging since the NETCONF traffic can then be easily captured and analyzed. It is also useful if we want to provide some local proprietary transport mechanism which is not SSH. Clear text TCP termination is not authenticated, the clear text client simply tells NSO which user the session should run as. The idea is that authentication is already done by some external entity, such as an SSH server. If clear text TCP is enabled, it is very important that NSO binds to localhost (127.0.0.1) for these connections.

Client libraries connect to NSO. For example the CDB API is TCP based and a CDB client connects to NSO. We instruct NSO which address to use for these connections through the `ncs.conf` parameters `/ncs-config/ncs-ipc-address/ip` (default address 127.0.0.1) and `/ncs-config/ncs-ipc-address/port` (default port 4565).

NSO multiplexes different kinds of connections on the same socket (IP and port combination). The following programs connect on the socket:

- Remote commands, such as e.g. `ncs --reload`
- CDB clients.
- External database API clients.
- MAAPI, The Management Agent API clients.
- The `ncs_cli` program

By default, all of the above are considered trusted. MAAPI clients and `ncs_cli` should supposedly authenticate the user before connecting to NSO whereas CDB clients and external database API clients are considered trusted and do not have to authenticate.

Thus, since the `ncs-ipc-address` socket allows full unauthenticated access to the system, it is important to ensure that the socket is not accessible from untrusted networks. However it is also possible to restrict

access to this socket by means of an access check, see [the section called “Restricting access to the IPC port”](#).

Running NSO as a non privileged user

A common misfeature found on UN*X operating systems is the restriction that only root can bind to ports below 1024. Many a dollar has been wasted on workarounds and often the results are security holes.

Both FreeBSD and Solaris have elegant configuration options to turn this feature off. On FreeBSD:

```
# sysctl net.inet.ip.portrange.reservedhigh=0
```

The above is best added to your `/etc/sysctl.conf`

Similarly on Solaris we can just configure this. Assuming we want to run NSO under a non-root user "ncs". On Solaris we can do that easily by granting the specific right to bind privileged ports below 1024 (and only that) to the "ncs" user using:

```
# /usr/sbin/usermod -K defaultpriv=basic,net_privaddr ncs
```

And check the we get what we want through:

```
# grep ncs /etc/user_attr
ncs:::type=normal;defaultpriv=basic,net_privaddr
```

Linux doesn't have anything like the above. There are a couple of options on Linux. The best is to use an auxiliary program like `authbind` <http://packages.debian.org/stable/authbind> or `privbind` <http://sourceforge.net/projects/privbind/>

These programs are run by root. To start ncs under e.g `privbind` we can do:

```
# privbind -u ncs /opt/ncs/current/bin/ncs -c /etc/ncs.conf
```

The above command starts NSO as user `ncs` and binds to ports below 1024

Using IPv6 on northbound interfaces

NSO supports access to all northbound interfaces via IPv6, and in the most simple case, i.e. IPv6-only access, this is just a matter of configuring an IPv6 address (typically the wildcard address "::") instead of IPv4 for the respective agents and transports in `ncs.conf`, e.g. `/ncs-config/cli/ssh/ip` for SSH connections to the CLI, or `/ncs-config/netconf-north-bound/transport/ssh/ip` for SSH to the NETCONF agent. The SNMP agent configuration is configured via one of the other northbound interfaces rather than via `ncs.conf`, see Chapter 4, *The NSO SNMP Agent in Northbound APIs*. For example via the CLI, we would set 'snmp agent ip' to the desired address. All these addresses default to the IPv4 wildcard address "0.0.0.0".

In most IPv6 deployments, it will however be necessary to support IPv6 and IPv4 access simultaneously. This requires that both IPv4 and IPv6 addresses are configured, typically "0.0.0.0" plus "::". To support this, there is in addition to the `ip` and `port` leafs also a list `extra-listen` for each agent and transport, where additional IP address and port pairs can be configured. Thus to configure the CLI to accept SSH connections to port 2024 on any local IPv6 address, in addition to the default (port 2024 on any local IPv4 address), we can add an `<extra-listen>` section under `/ncs-config/cli/ssh` in `ncs.conf`:

```
<cli>
  <enabled>true</enabled>

  <!-- Use the built-in SSH server -->
  <ssh>
```

```
<enabled>true</enabled>
<ip>0.0.0.0</ip>
<port>2024</port>

<extra-listen>
  <ip>::</ip>
  <port>2024</port>
</extra-listen>

</ssh>

...
</cli>
```

To configure the SNMP agent to accept requests to port 161 on any local IPv6 address, we could similarly use the CLI and give the command:

```
admin@nics(config)# snmp agent extra-listen :: 161
```

The `extra-listen` list can take any number of address/port pairs, thus this method can also be used when we want to accept connections/requests on several specified (IPv4 and/or IPv6) addresses instead of the wildcard address, or we want to use multiple ports.



High Availability

- [Introduction to NSO High Availability, page 67](#)
- [NSO HA Raft, page 68](#)
- [NSO Rule-based HA, page 79](#)
- [Tail-f HCC Package, page 84](#)
- [Setup with an External Load Balancer, page 99](#)
- [NB listen addresses on HA primary for Load Balancers, page 102](#)
- [HA framework requirements, page 102](#)
- [Mode of operation, page 103](#)
- [Security aspects, page 105](#)
- [API, page 105](#)
- [Ticks, page 105](#)
- [Relay secondaries, page 106](#)
- [CDB replication, page 107](#)

Introduction to NSO High Availability

As a single NSO node can fail or lose network connectivity, you can configure multiple nodes in a highly available (HA) setup, which replicates the CDB configuration and operational data across participating nodes. It allows the system to continue functioning even when some nodes are inoperable.

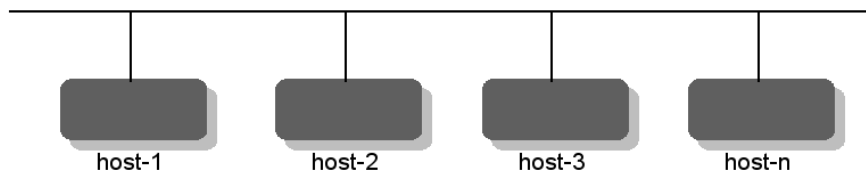
The replication architecture is that of one active primary and a number of secondaries. This means all configuration write operations must occur on the primary, which distributes the updates to the secondaries.

Operational data in the CDB may be replicated or not based on the `tailf:persistent` statement in the data model. If replicated, operational data writes can only be performed on the primary, whereas non-replicated operational data can also be written on the secondaries.

Replication is supported in several different architectural setups. For example, two-node active/standby designs as well as multi-node clusters with runtime software upgrade.



Primary - secondary configuration



One primary - several secondaries

This feature is independent from but compatible with the Layered Service Architecture (LSA) in *Layered Service Architecture* feature, which also configures multiple NSO nodes to provide additional scalability. When the following text simply refers to a cluster, it identifies the set of NSO nodes participating in the same HA group, not an LSA cluster, which is a separate concept.

NSO supports the following options for implementing an HA setup to cater to the widest possible range of use-cases (only one can be used at a time):

- **HA Raft:** Using a modern, consensus-based algorithm, it offers a robust, hands-off solution that works best in the majority of cases.
- **Rule-based HA:** A less sophisticated solution that allows you to influence the primary selection but may require occasional manual operator action.
- **External HA:** NSO only provides data replication; all other functions, such as primary selection and group membership management, are performed by an external application, using the HA framework (HAFW).

In addition to data replication, having a fixed address to connect to the current primary in an HA group greatly simplifies access for operators, users, and other systems alike. Use the [the section called “Tail-f HCC Package”](#) or [an external load balancer](#) to manage it.

NSO HA Raft

[Raft](#) is a consensus algorithm that reliably distributes a set of changes to a group of nodes and robustly handles network and node failure. It can operate in face of multiple, subsequent failures, while also

allowing a previously failed or disconnected node to automatically rejoin the cluster without risk of data conflicts.

Compared to traditional fail-over HA solutions, Raft relies on the consensus of the participating nodes, which addresses the so-called “split-brain” problem, where multiple nodes assume a primary role. This problem is especially characteristic of two-node systems, where it is impossible for a single node on its own to distinguish between losing network connectivity itself versus the other node malfunctioning. For this reason Raft requires at least three nodes in the cluster.

Raft achieves robustness by requiring at least three nodes in the HA cluster. Three is the recommended cluster size, allowing the cluster to operate in face of a single node failure. In case you need to tolerate two nodes failing simultaneously, you can add two additional nodes, for a 5-node cluster. But permanently having more than five nodes in a single cluster is currently not recommended since Raft requires the majority of the currently configured nodes in the cluster to reach consensus. Without the consensus, the cluster *cannot* function.

You can start a sample HA Raft cluster using the `examples.ncs/high-availability/raft-cluster` example to test it out. The scripts in the example show various aspects of cluster setup and operation, which are further described in the rest of this chapter.

Optionally, examples using separate containers for each HA Raft cluster member with NSO system installations are available and referenced in the `examples.ncs/development-guide/high-availability/hcc` example in the NSO example set.

Overview of Raft Operation

The Raft algorithm works with the concept of (election) terms. In each term, nodes in the cluster vote for a leader. The leader is elected when it receives the majority of the votes. Since each node only votes for a single leader in a given term, there can only be one leader in the cluster for this term.

Once elected, the leader becomes responsible for distributing the changes and ensuring consensus in the cluster for that term. Consensus means that the majority of the participating nodes must confirm a change before it is accepted. This is required for the system to ensure no changes ever get overwritten and provide the reliability guarantees. On the other hand, it also means more than half of the nodes must be available for normal operation.

Changes can only be performed on the leader, which will accept the change after majority of the cluster nodes confirm it. This is the reason a typical Raft cluster has an odd number of nodes; exactly half of the nodes agreeing on a change is not sufficient. It also makes a two-node cluster (or any even number of nodes in a cluster) impractical; the system as a whole is no more available than it is with one fewer node.

If the connection to the leader is broken, such as during a network partition, the nodes start a new term and a new election. Another node can become a leader if it gets the majority of the votes of all nodes initially in the cluster. While gathering votes, the node has a status of a candidate. In case multiple nodes assume candidate status, a split-vote scenario may occur, which is resolved by starting a fresh election until a candidate secures the majority vote.

If it happens that there aren't enough reachable nodes to obtain a majority, a candidate can stay in the candidate state for an indefinite time. Otherwise, when a node votes for a candidate, it becomes a follower and stays a follower in this term, regardless if the candidate is elected or not.

Additionally, NSO node can also be in the stalled state, if HA Raft is enabled but the node has not joined a cluster.

Node Names and Certificates

Each node in an HA Raft cluster needs a unique name. Names are usually in the *ADDRESS* format, where *ADDRESS* identifies a network host where the NSO process is running, such as a fully-qualified domain name (FQDN) or an IPv4 address.

Other nodes in the cluster must be able to resolve and reach the *ADDRESS*, which creates a dependency on the DNS if you use domain names instead of IP addresses.

Limitations of the underlying platform place a constraint on the format of *ADDRESS*, which can't be a simple short name (without a dot), even if the system is able to resolve such a name using `hosts` file or a similar mechanism.

You specify the node address in the `ncs.conf` file as the value for `node-address`, under the `listen` container. You can also use the full node name (with the “@” character), however, that is usually unnecessary as the system prepends `ncsd@` as-needed.

Another aspect in which *ADDRESS* plays a role is authentication. The HA system uses mutual TLS to secure communication between cluster nodes. This requires you to configure a trusted Certificate Authority (CA) and a key/certificate pair for each node. When nodes connect, they check that certificate of the peer validates against the CA and matches the *ADDRESS* of the peer.



Note

Consider that TLS not only verifies that the certificate/key pair comes from a trusted source (certificate is signed by a trusted CA), it also checks that the certificate matches the host you are connecting to. Host A may have a valid certificate and key, signed by a trusted CA, however, if the certificate is for another host, say host B, the authentication will fail.

In most cases this means the *ADDRESS* must appear in the node certificate's Subject Alternative Name (SAN) extension, as `dNSName` (see [RFC2459](#)).

Create and use a self-signed CA to secure the NSO HA Raft cluster. A self-signed CA is the only secure option. The CA should only be used to sign the certificates of the member nodes in one NSO HA Raft cluster. It is critical for security that the CA is not used to sign any other certificates. Any certificate signed by the CA can be used to gain complete control of the NSO HA Raft cluster.

See the `examples.ncs/high-availability/raft-cluster` example for one way to set up a self-signed CA and provision individual node certificates. The example uses a shell script `gen_tls_certs.sh` that invokes the `openssl` command. Consult [the section called “Recipe for a Self-signed CA”](#) for using it independently of the example.

Examples using separate containers for each HA Raft cluster member with NSO system installations that use a variant of the `gen_tls_certs.sh` script are available and referenced in the `examples.ncs/development-guide/high-availability/hcc` example in the NSO example set.



Note

When using an IP address instead of a DNS name for node's *ADDRESS*, you must add the IP address to the certificate's `dNSName` SAN field (adding it to `iPAddress` field only is insufficient). This is a known limitation in the current version.

The following is a HA Raft configuration snippet for `ncs.conf` that includes certificate settings and a sample *ADDRESS*:


```

<ha-raft>
  <!-- ... -->
  <listen>
    <node-address>198.51.100.10</node-address>
  </listen>
  <ssl>
    <ca-cert-file>${NCS_CONFIG_DIR}/dist/ssl/cert/myca.crt</ca-cert-file>
    <cert-file>${NCS_CONFIG_DIR}/dist/ssl/cert/node-100-10.crt</cert-file>
    <key-file>${NCS_CONFIG_DIR}/dist/ssl/cert/node-100-10.key</key-file>
  </ssl>
</ha-raft>

```

Recipe for a Self-signed CA

HA Raft uses standard TLS protocol with public key cryptography for securing cross-node communication, where each node requires a separate public/private key pair and a corresponding certificate. Key and certificate management is a broad topic and is critical to the overall security of the system.

The following text provides a recipe for generating certificates using a self-signed CA. It uses strong cryptography and algorithms that are deemed suitable for production use. However, it makes a few assumptions which may not be appropriate for all environments. Always consider how they affect your own deployment and consult a security professional if in doubt.

The recipe makes the following assumptions:

- You use a secured workstation or server to run these commands and handle the generated keys with care. In particular, you must copy the generated keys to NSO nodes in a secure fashion, such as using **scp**.
- The CA is used solely for a single NSO HA Raft cluster, with certificates valid for 10 years, and provides no CRL. If a single key or host is compromised, a new CA and **all** key/certificate pairs must be recreated and reprovisioned in the cluster.
- Keys and signatures based on **ecdsa-with-sha384/P-384** are sufficiently secure for the vast majority of environments. However, if your organization has specific requirements, be sure to follow those.

To use this recipe, first prepare a working environment on a secure host by creating a new directory and copy the **gen_tls_certs.sh** script from **\$NCS_DIR/examples.ncs/high-availability/raft-cluster** into it. Additionally, ensure the **openssl** command, version 1.1 or later, is available and the system time is set correctly. Supposing you have a cluster named *lower-west*, you might run:

```

$ mkdir raft-ca-lower-west
$ cd raft-ca-lower-west
$ cp $NCS_DIR/examples.ncs/high-availability/raft-cluster/gen_tls_certs.sh .
$ openssl version
$ date

```



Note

Including cluster name in the directory name helps distinguish certificates of one HA cluster from another, such as when using an LSA deployment in an HA configuration.

The recipe relies on the **gen_tls_certs.sh** script to generate individual certificates. For clusters using FQDN node addresses, invoke the script with full hostnames of all the participating nodes. For example:

```
$ ./gen_tls_certs.sh node1.example.org node2.example.org node3.example.org
```

**Note**

Using only hostnames, e.g. `node1`, will not work.

If your HA cluster is using IP addresses instead, add the `-a` option to the command and list the IPs:

```
$ ./gen_tls_certs.sh -a 192.0.2.1 192.0.2.2 192.0.2.3
```

The script outputs the location of the relevant files and you should securely transfer each set of files to the corresponding NSO node. For each node, transfer only the three files: `ca.crt`, `host.crt`, and `host.key`.

Once certificates are deployed, you can check their validity with the **openssl verify** command:

```
$ openssl verify -CAfile ssl/certs/ca.crt ssl/certs/node1.example.org.crt
```

This command takes into account the current time and can be used during troubleshooting. It can also display information contained in the certificate if you use the **openssl x509 -text -in ssl/certs/node1.example.org.crt -noout** variant. The latter form allows you to inspect the incorporated hostname/IP address and certificate validity dates.

Network and `ncs.conf` Prerequisites

In addition to network connectivity required for normal operation of a standalone NSO node, nodes in the HA Raft cluster must be able to initiate TCP connections from a random ephemeral client port to the following ports on other nodes:

- Port 4369
- Ports in the range 4370-4399 (configurable)

You can change the ports in the second listed range from the default of 4370-4399. Use the `min-port` and `max-port` settings of the `ha-raft/listen` container.

The Raft implementation does not impose any other hard limits on the network but you should keep in mind that consensus requires communication with other nodes in the cluster. A high round-trip latency between cluster nodes is likely to negatively impact transaction throughput of the system.

The HA Raft cluster also requires compatible `ncs.conf` files among the member nodes. In particular, `/ncs-config/cdb/operational/enabled` and `/ncs-config/rollback/enabled` values affect replication behavior and must match. Likewise, each member must have the same set of encryption keys and the keys cannot be changed while cluster is in operation.

To update the `ncs.conf` configuration, you must manually update the copy on each member node, making sure the new versions contain compatible values. Then perform the reload on the leader and the follower members will automatically reload their copies of the configuration file as well.

If a node is a cluster member but has been configured with a new, incompatible `ncs.conf` file, it gets automatically disabled. See the `/ha-raft/status/disabled-reason` for reason. You can re-enable the node with the **ha-raft reset** command, once you have reconciled the incompatibilities.

Connected Nodes and Node Discovery

Raft has a notion of cluster configuration, in particular, how many and which members the cluster has. You define member nodes when you first initialize the cluster with the **create-cluster** command or use the **adjust-membership** command. The member nodes allow cluster to know how many nodes are needed for consensus and similar.

However, not all cluster members may be reachable or alive all the time. Raft implementation in NSO uses TCP connections between nodes to transport data. The TCP connections are authenticated and encrypted using TLS by default (see [the section called “Security Considerations”](#)). A working connection between nodes is essential for the cluster to function but a number of factors, such as firewall rules or expired/invalid certificates, can prevent the connection from establishing.

Therefore, NSO distinguishes between configured member nodes and nodes to which it has established a working transport connection. The latter are called connected nodes. In a normal, fully working and properly configured cluster, the connected nodes will be the same as member nodes (except for the current node).

To help troubleshoot connectivity issues without affecting cluster operation, connected nodes will show even nodes that are not actively participating in the cluster but have established a transport connection to nodes in the cluster. The optional discovery mechanism, described next, relies on this functionality.

NSO includes a mechanism that simplifies the initial cluster setup by enumerating known nodes. This mechanism uses a set of *seed nodes* to discover all connectable nodes, which can then be used with the **create-cluster** command to form a Raft cluster.

When you specify one or more nodes with the `/ha-raft/seed-nodes/seed-node` setting in the `ncs.conf` file, the current node tries to establish connection to these seed nodes, in order to discover the list of all nodes potentially participating in the cluster. For the discovery to work properly, all other nodes must also use seed nodes and the set of seed nodes must overlap. The recommended practice is to use the same set of seed nodes on every participating node.

Along with providing an autocompletion list for the **create-cluster** command, this feature streamlines discovery of node names when using NSO in containerized or other dynamic environments, where node addresses are not known in advance.

Initial Cluster Setup

Creating a new HA cluster consists of two parts: configuring the individual nodes and running the **create-cluster** action.

First, you must update the `ncs.conf` configuration file for each node. All HA Raft configuration comes under the `/ncs-config/ha-raft` element.

As part of the configuration, you must:

- Enable HA Raft functionality through the `enabled` leaf.
- Set `node-address` and the corresponding TLS parameters (see [the section called “Node Names and Certificates”](#)).
- Identify the cluster this node belongs to with `cluster-name`.
- Reload or restart the NSO process (if already running).
- Repeat preceding steps for every participating node.
- Invoke the **create-cluster** action.

The cluster name is simply a character string that uniquely identifies this HA cluster. The nodes in the cluster must use the same cluster name or they will refuse to establish connection. This setting helps prevent mistakenly adding a node to the wrong cluster when multiple clusters are in operation, such as in an LSA setup.

Example 3. Sample HA Raft config for a cluster node

```
<ha-raft>
```

```

<enabled>true</enabled>
<cluster-name>sherwood</cluster-name>
<listen>
  <node-address>ash.example.org</node-address>
</listen>
<ssl>
  <ca-cert-file>${NCS_CONFIG_DIR}/dist/ssl/cert/myca.crt</ca-cert-file>
  <cert-file>${NCS_CONFIG_DIR}/dist/ssl/cert/ash.crt</cert-file>
  <key-file>${NCS_CONFIG_DIR}/dist/ssl/cert/ash.key</key-file>
</ssl>
<seed-nodes>
  <seed-node>birch.example.org</seed-node>
</seed-nodes>
</ha-raft>

```

With all the nodes configured and running, connect to the node that you would like to become the initial leader and invoke the **ha-raft create-cluster** action. The action takes a list of nodes identified by their names. If you have configured seed-nodes, you will get auto-completion support, otherwise you have to type in names of the nodes yourself.

This action makes the current node a cluster leader and joins the other specified nodes to the newly created cluster. For example:

```

admin@ncs# ha-raft create-cluster members [ birch.example.org cedar.example.org ]
admin@ncs# show ha-raft
ha-raft status role leader
ha-raft status leader ash.example.org
ha-raft status member [ ash.example.org birch.example.org cedar.example.org ]
ha-raft status connected-node [ birch.example.org cedar.example.org ]
ha-raft status local-node ash.example.org
...

```

You can use the **show ha-raft** command on any node to inspect the status of the HA Raft cluster. The output includes the current cluster leader and members according to this node, as well as information about the local node, such as node name (`local-node`) and role. The `status/connected-node` list contains the names of the nodes with which this node has active network connections.

In case you get an error, such as the `Error: NSO can't reach member node 'ncsd@ADDRESS' .`, please verify all of the following:

- Node at `ADDRESS` is reachable. You can use the **ping ADDRESS** command, for example.
- The problematic node has correct `ncs.conf` configuration, especially `cluster-name` and `node-address`. The latter should match the `ADDRESS` and should contain at least one dot.
- Nodes use compatible configuration. For example, make sure the `ncs.crypto_keys` file (if used) or the `encrypted-strings` configuration in `ncs.conf` is identical across nodes.
- HA Raft is enabled, using the **show ha-raft** command on the unreachable node.
- The firewall configuration on OS and on network level permits traffic on the required ports (see [the section called “Network and ncs.conf Prerequisites”](#)).
- The node uses a certificate that the CA can validate. For example, copy the certificates to the same location and run **openssl verify -CAfile CA_CERT NODE_CERT** to verify this.
- Verify the **epmd -names** command on each node shows the `ncsd` process. If not, stop NSO, run **epmd -kill**, and then start NSO again.

In addition to the above, you may also examine the `logs/raft.log` file for detailed information on the error message and overall operation of the Raft algorithm. The amount of information in the file is controlled by the `/ncs-config/logs/raft-log` configuration in the `ncs.conf`.

Cluster Management

After the initial cluster setup, you can add new nodes or remove existing nodes from the cluster with the help of the **ha-raft adjust-membership** action. For example:

```
admin@ncs# show ha-raft status member
ha-raft status member [ ash.example.org birch.example.org cedar.example.org ]
admin@ncs# ha-raft adjust-membership remove-node birch.example.org
admin@ncs# show ha-raft status member
ha-raft status member [ ash.example.org cedar.example.org ]
admin@ncs# ha-raft adjust-membership add-node dollartree.example.org
admin@ncs# show ha-raft status member
ha-raft status member [ ash.example.org cedar.example.org dollartree.example.org ]
```

When removing nodes using the **ha-raft adjust-membership remove-node** command, the removed node is not made aware that it is removed from the cluster and continues signaling the other nodes. This is a limitation in the algorithm, as it must also handle situations, where the removed node is down or unreachable. To prevent further communication with the cluster, it is important you ensure the removed node is shut down. You should shut down the to-be-removed node prior to removal from the cluster, or immediately after it. The former is recommended but the latter is required if there are only two nodes left in the cluster and shutting down prior to removal would prevent the cluster from reaching consensus.

Additionally, you can force an existing follower node to perform a full re-sync from the leader by invoking the **ha-raft reset** action with the `force` option. Using this action on the leader will make the node give up the leader role and perform a sync with the newly elected leader.

As leader selection during Raft election is not deterministic, NSO provides the **ha-raft handover** action, which allows you to either trigger a new election if called with no arguments, or transfer leadership to a specific node. The latter is especially useful when, for example, one of the nodes resides in a different location and more traffic between locations may incur extra costs or additional latency, so you prefer this node is not the leader under normal conditions.

Migrating From Existing Rule-based HA

If you have an existing HA cluster using the rule-based built-in HA, you can migrate it to use HA Raft instead. This procedure is performed in four distinct high-level steps:

- Ensuring the existing cluster meets migration prerequisites.
- Preparing the required HA Raft configuration files.
- Switching to HA Raft.
- Adding additional nodes to the cluster.

The procedure does not perform an NSO version upgrade, so the cluster remains on the same version. It also does not perform any schema upgrades, it only changes the type of the HA cluster.

The migration procedure is in-place, that is, the existing nodes are disconnected from the old cluster and connected to the new one. This results in a temporary disruption of the service, so it should be performed during a service window.

First, you should ensure the cluster meets migration prerequisites. The cluster must use:

- NSO 6.1.2 or later
- `tailf-hcc` 6.0 or later (if used)

In case these prerequisites are not met, follow the standard upgrade procedures to upgrade the existing cluster to supported versions first.

Additionally, ensure that all used packages are compatible with HA Raft, as NSO uses some new or updated notifications about HA state changes. Also verify the network supports the new cluster communications ([the section called “Network and ncs.conf Prerequisites”](#)).

Secondly, prepare all the `ncs.conf` and related files for each node, such as certificates and keys. Create a copy of all the `ncs.conf` files and disable or remove the existing `>ha<` section in the copies. Then add the required configuration items to the copies, as described in [the section called “Initial Cluster Setup”](#) and [the section called “Node Names and Certificates”](#). Do not update the `ncs.conf` files used by the nodes yet.

It is recommended but not necessary that you set the seed nodes in `ncs.conf` to the designated primary and fail-over primary. Do this for all `ncs.conf` files for all nodes.

Procedure 7.1. Migration to HA Raft

- Step 1** With the new configurations at hand and verified, start the switch to HA Raft. The cluster nodes should be in their nominal, designated roles. If not, perform a fail over first.
- Step 2** On the designated (actual) primary, called `node1`, enable read-only mode.
- ```
admin@node1# high-availability read-only mode true
```
- Step 3** Then take a backup of all nodes.
- Step 4** Once the backup successfully completes, stop the designated fail-over primary (actual secondary) NSO process, update its `ncs.conf` and the related (certificate) files for HA Raft, then start it again. Connect to this node's CLI, here called `node2`, and verify HA Raft is enabled with the **show ha-raft** command.

```
admin@node2# show ha-raft
ha-raft status role stalled
ha-raft status local-node node2.example.org
> ... output omitted ... <
```

**Step 5** Now repeat the same for the designated primary (`node1`). If you have set the seed nodes, you should see the fail-over primary show under connected-node.

```
admin@node1# show ha-raft
ha-raft status role stalled
ha-raft status connected-node [node2.example.org]
ha-raft status local-node node1.example.org
> ... output omitted ... <
```

**Step 6** On the old designated primary (`node1`) invoke the **ha-raft create-cluster** action and create a two node Raft cluster with the old fail-over primary (`node2`, actual secondary). The action takes a list of nodes identified by their names. If you have configured seed-nodes, you will get auto-completion support, otherwise you have to type in the name of the node yourself.

```
admin@node1# ha-raft create-cluster members [node2.example.org]
admin@node1# show ha-raft
ha-raft status role leader
ha-raft status leader node1.example.org
ha-raft status member [node1.example.org node2.example.org]
ha-raft status connected-node [node2.example.org]
ha-raft status local-node node1.example.org
> ... output omitted ... <
```

In case of errors running the action, refer to [the section called “Initial Cluster Setup”](#) for possible causes and troubleshooting steps.

**Step 7** Raft requires at least three nodes to operate effectively (as described in [the section called “NSO HA Raft”](#)) and currently there are only two in the cluster.

If the initial cluster had only two nodes, you must provision an additional node and set it up for HA Raft.

If the cluster initially had three nodes, there is the remaining secondary node, node3, which you must stop, update its configuration as you did with the other two nodes, and start it up again.

### Step 8

Finally, on the old designated primary and current HA Raft leader, use the **ha-raft adjust-membership add-node** action to add this third node to the cluster.

```
admin@node1# ha-raft adjust-membership add-node node3.example.org
admin@node1# show ha-raft status member
ha-raft status member [node1.example.org node2.example.org node3.example.org]
```

## Security Considerations

Communication between the NSO nodes in an HA Raft cluster takes place over Distributed Erlang, an RPC protocol transported over TLS (unless explicitly disabled by setting `/ncs-config/ha-raft/ssl/enabled` to 'false').

TLS (Transport Layer Security) provides Authentication and Privacy by only allowing NSO nodes to connect using certificates and keys issued from the same Certificate Authority (CA). Distributed Erlang is transported over TLS 1.3. Access to a host can be revoked by the CA through the means of a CRL (Certificate Revocation List). To enforce certificate revocation within an HA Raft cluster, invoke the action `/ha-raft/disconnect` to terminate the pre-existing connection. A connection to the node can re-establish once the node's certificate is valid.

Please ensure the CA key is kept in a safe place since it can be used to generate new certificate and key pairs for peers.

Distributed Erlang supports for multiple NSO nodes to run on the same host and the node addresses are resolved by the `epmd` ([Erlang Port Mapper Daemon](#)) service. Once resolved, the NSO nodes communicate directly.

The ports `epmd` and the NSO nodes listen to can be found here: [the section called “Network and ncs.conf Prerequisites”](#). `epmd` binds the wildcard IPv4 address `0.0.0.0` and the IPv6 address `:::`.

In case `epmd` is exposed to a DoS attack, the HA Raft members may be unable to resolve addresses and communication could be disrupted. Please ensure traffic on these ports are only accepted between the HA Raft members by using firewall rules or other means.

Two NSO nodes can only establish a connection if a shared secret "cookie" matches. The cookie is optionally configured from `/ncs-config/ha-raft/cluster-name`. Please note the cookie is not a security feature but a way to isolate HA Raft clusters and to avoid accidental misuse.

## Packages Upgrades in Raft Cluster

NSO contains a mechanism for distributing packages to nodes in a Raft cluster, greatly simplifying package management in a highly-available setup.

You perform all package management operations on the current leader node. To identify the leader node, you can use the **show ha-raft status leader** command on a running cluster.

Invoking the **packages reload** command makes the leader node update its currently loaded packages, identical to a non-HA, single-node setup. At the same time, the leader also distributes these packages to the followers to load. However, the load paths on the follower nodes, such as `/var/opt/ncs/packages/`, are **not** updated. This means, if leader election took place, a different leader was elected, and you performed another **packages reload**, the system would try to load the versions of the packages on this other leader, which may be out of date or not even present.

The recommended approach is therefore to use the **packages ha sync and-reload** command instead, unless a load path is shared between NSO nodes, such as the same network drive. This command distributes and updates packages in the load paths on the follower nodes, as well as loading them.

For the full procedure, first ensure all cluster nodes are up and operational, then follow these steps on the leader node:

- Perform a full backup of the NSO instance, such as running **ncs-backup**.
- Add, replace, or remove packages on the filesystem. The exact location depends on the type of NSO deployment, for example `/var/opt/ncs/packages/`.
- Invoke the **packages ha sync and-reload** or **packages ha sync and-add** command to start the upgrade process.

Note that while the upgrade is in progress, writes to the CDB are not allowed and will be rejected.

For a **packages ha sync and-reload** example see the `raft-upgrade-12` NSO system installation-based example referenced by the `examples.ncs/development-guide/high-availability/hcc` example in the NSO example set.

For more details, troubleshooting, and general upgrade recommendations, see [Chapter 5, NSO Packages](#) and [Chapter 10, Upgrade](#).

## Version Upgrade of Cluster Nodes

Currently the only supported and safe way of upgrading the Raft HA cluster NSO version requires the cluster be taken offline, since the nodes must at all times run the same software version.

Do not attempt upgrade unless all cluster member nodes are up and actively participating in the cluster. Verify the current cluster state with the **show ha-raft status** command. All member nodes must also be present in the connected-node list.

The procedure differentiates between the current leader node versus followers. To identify the leader, you can use the **show ha-raft status leader** command on a running cluster.

---

### Procedure 7.2. Cluster version upgrade

- |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |               |                                                                                                                                                                                                                                                                                                             |               |                                |
|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|--------------------------------|
| <b>Step 1</b> | On the leader, first enable read-only mode using the <b>ha-raft read-only mode true</b> command and then verify all cluster nodes are in-sync with the <b>show ha-raft status log replications state</b> command.                                                                                                                                                                                                                                                                                                                                                                                                                                                               |               |                                                                                                                                                                                                                                                                                                             |               |                                |
| <b>Step 2</b> | Stop the <b>ncs</b> process on all the follower nodes, for example invoking the <code>/etc/init.d/ncs stop</code> command on each node.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |               |                                                                                                                                                                                                                                                                                                             |               |                                |
| <b>Step 3</b> | Stop the <b>ncs</b> process on the leader node only after you have stopped all the follower nodes in the previous step.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |               |                                                                                                                                                                                                                                                                                                             |               |                                |
| <b>Step 4</b> | Remember to take backup of each node before attempting the upgrade procedure.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |               |                                                                                                                                                                                                                                                                                                             |               |                                |
| <b>Step 5</b> | Perform the full single-node upgrade procedure on the original leader node. Upon completion, the node will come back up but will not have quorum to become a leader because all other nodes are stopped.                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |               |                                                                                                                                                                                                                                                                                                             |               |                                |
| <b>Step 6</b> | Upgrade each original follower node one by one by performing the following actions: <table border="0" style="margin-left: 20px;"> <tr> <td style="vertical-align: top; padding-right: 10px;"><b>Step a</b></td> <td>Delete the <code>\$NCS_RUN_DIR/state/raft/</code> directory with a command such as <b>rm -rf /var/opt/ncs/state/raft/</b> for a typical system install or <b>rm -rf /nso/run/state/raft/</b> in an NSO container. Note that this action is done only on the follower nodes, <i>not</i> the original leader.</td> </tr> <tr> <td style="vertical-align: top; padding-right: 10px;"><b>Step b</b></td> <td>Perform a single-node upgrade.</td> </tr> </table> | <b>Step a</b> | Delete the <code>\$NCS_RUN_DIR/state/raft/</code> directory with a command such as <b>rm -rf /var/opt/ncs/state/raft/</b> for a typical system install or <b>rm -rf /nso/run/state/raft/</b> in an NSO container. Note that this action is done only on the follower nodes, <i>not</i> the original leader. | <b>Step b</b> | Perform a single-node upgrade. |
| <b>Step a</b> | Delete the <code>\$NCS_RUN_DIR/state/raft/</code> directory with a command such as <b>rm -rf /var/opt/ncs/state/raft/</b> for a typical system install or <b>rm -rf /nso/run/state/raft/</b> in an NSO container. Note that this action is done only on the follower nodes, <i>not</i> the original leader.                                                                                                                                                                                                                                                                                                                                                                     |               |                                                                                                                                                                                                                                                                                                             |               |                                |
| <b>Step b</b> | Perform a single-node upgrade.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |               |                                                                                                                                                                                                                                                                                                             |               |                                |
| <b>Step 7</b> | Once follower nodes are back online, the original leader node will attain quorum and be re-elected a leader. Verify the state of the cluster through the <b>show ha-raft status</b> command.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |               |                                                                                                                                                                                                                                                                                                             |               |                                |
| <b>Step 8</b> | Finally, verify that all data has been correctly synchronized across all cluster nodes and that the leader is no longer read-only. The latter happens automatically on being re-elected.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |               |                                                                                                                                                                                                                                                                                                             |               |                                |
-



For a standard system install, the single-node procedure is described in [the section called “Single Instance Upgrade”](#), but in general depends on the NSO deployment type. For example, it will be different for containerized environments. For specifics, please refer to the documentation for the deployment type.

For an example see the `raft-upgrade-12` NSO system installation-based example referenced by the `examples.ncs/development-guide/high-availability/hcc` example in the NSO example set.

If the upgrade fails before or during the upgrade of the original leader, start up the original followers to restore service and then restore the original leader, using backup as necessary.

However, if upgrade fails after the original leader was successfully upgraded, you should still be able to complete the cluster upgrade. If you are unable to upgrade a follower node, you may provision a (fresh) replacement and the data and packages in use will be copied from the leader.

## NSO Rule-based HA

NSO has capability to manage HA groups based on a set of predefined rules. This functionality was added in NSO 5.4 and is sometimes referred to simply as the built-in HA. However, since NSO 6.1, HA Raft, which is also built-in is available as well and is likely a better choice in most situations.

Rule-based HA allows administrators to:

- Configure HA group members with IP addresses and default roles
- Configure failover behavior
- Configure start-up behavior
- Configure HA group members with IP addresses and default roles
- Assign roles, join HA group, enable/disable rule-based HA through actions
- View the state of current HA setup

NSO rule-based HA is defined in `tailf-ncs-high-availability.yang`, with data residing under the `/high-availability/` container.

NSO rule-based HA does not manage any virtual IP addresses, advertise any BGP routes or similar. This must be handled by an external package. Tail-f HCC 5.x and greater has this functionality compatible with NSO rule-based HA. You can read more about the HCC package in the [following chapter](#).

## Prerequisites

In order to use NSO rule-based HA, HA must first be enabled in `ncs.conf` - See [the section called “Mode of operation”](#)

Note: if the package `tailf-hcc` with a version less than 5.0 is loaded, NSO rule-based HA will not function. These HCC versions may still be used but NSO built-in HA will not function in parallel.

## HA Member configuration

All HA group members are defined under `/high-availability/ha-node`. Each configured node must have a unique IP address configured, and a unique HA Id. Additionally, nominal roles and fail-over settings may be configured on a per node-basis.

The HA Node Id is a unique identifier used to identify NSO instances in a HA group. The HA Id of the local node - relevant amongst others when an action is called - is determined by matching configured HA

node IP addresses against IP addresses assigned to the host machine of the NSO instance. As the HA Id is crucial to NSO HA, NSO rule-based HA will not function if the local node cannot be identified.

In order to join a HA group, a shared secret must be configured on the active primary and any prospective secondary. This used for a CHAP-2 like authentication and is specified under `/high-availability/token/`.



#### Note

In an NSO system install setup, not only the shared token needs to match between the HA group nodes, the configuration for encrypted-strings, default stored in `/etc/ncs/ncs.crypto_keys`, need to match between the nodes in the HA group too.

The token configured on the secondary node is overwritten with the encrypted token of type `aes-256-cfb-128-encrypted-string` from the primary node when the secondary node connects to the primary. If there is a mismatch between the encrypted-string configuration on the nodes, NSO will not decrypt the HA token to match the token presented. As a result, the primary node denies the secondary node access the next time the HA connection needs to reestablish with a "Token mismatch, secondary is not allowed" error.

See the `upgrade-12` example, referenced from `examples.ncs/development-guide/high-availability/hcc`, for an example setup and the [Chapter 11, Deployment Example](#) for a description of the example.

Also, note that the `ncs.crypto_keys` file is highly sensitive. The file contains the encryption keys for all CDB data that is encrypted on disk. Besides the HA token, this often includes passwords for various entities, such as login credentials to managed devices.

## HA Roles

NSO can assume HA roles *primary*, *secondary* and *none*. Roles can be assigned directly through actions, or at startup or failover. See [the section called “HA framework requirements”](#) for the definition of these roles.

Note: NSO rule-based HA does not support relay-secondaries.

NSO rule-based HA distinguishes between the concepts of *nominal role* and *assigned role*. Nominal-role is configuration data that applies when a NSO instance starts up and at failover. Assigned role is the role the NSO instance has been ordered to assume either by an action, or as result of startup or failover.

## Failover

Failover may occur when a secondary node loses the connection to the primary node. A secondary may then take over the primary role. Failover behaviour is configurable and controlled by the parameters:

- `/high-availability/ha-node{id}/failover-primary`
- `/high-availability/settings/enable-failover`

For automatic failover to function, `/high-availability/settings/enable-failover` must be set to *true*. It is then possible to enable at most one node with nominal role secondary as *failover-primary*, by setting the parameter `/high-availability/ha-node{id}/failover-primary`. A node with nominal role primary is also implicitly a failover-primary - it will act as failover-primary if its currently assigned role is a secondary.

Before failover happens, a failover-primary enabled secondary node may attempt to reconnect to the previous primary before assuming the primary role. This behaviour is configured by the parameters

- `/high-availability/settings/reconnect-attempts`
- `/high-availability/settings/reconnect-interval`

denoting how many reconnect attempts will be made, and with which interval, respectively.

HA Members that are assigned as secondaries, but are neither failover-primaries nor set with nominal-role primary, may attempt to rejoin the HA group after losing connection to primary.

This is controlled by `/high-availability/settings/reconnect-secondaries`. If this is true, secondary nodes will query the nodes configured under `/high-availability/ha-node` for a NSO instance that currently has the primary role. Any configured nominal-roles will not be considered. If no primary node is found, subsequent attempts to rejoin the HA setup will be issued with an interval defined by `/high-availability/settings/reconnect-interval`.

In case a net-split provokes a failover it is possible to end up in a situation with two primaries, both nodes accepting writes. The primaries are then not synchronized and will end up in split-brain. Once one of the primaries join the other as a secondary, the HA cluster is once again consistent but any out of sync changes will be overwritten.

To prevent split-brain to occur, NSO 5.7 or later comes with a rule-based algorithm. The algorithm is enabled by default, it can be disabled or changed from the parameters:

- `/high-availability/settings/consensus/enabled [true]`
- `/high-availability/settings/consensus/algorithm [ncs:rule-based]`

The rule-based algorithm can be used in either of the two HA constellations:

- Two nodes: one nominal primary and one nominal secondary configured as failover-primary.
- Three nodes: one nominal primary, one nominal secondary configured as failover-primary and one perpetual secondary.

On failover:

- Failover-primary: become primary but enable read-only mode. Once the secondary joins, disable read-only.
- Nominal primary: on loss of all secondaries, change role to none. If one secondary node is connected, stay primary.

Note: In certain cases the rule-based consensus algorithm results in nodes being disconnected and will not automatically re-join the HA cluster, such as in the example above when the nominal primary becomes none on loss of all secondaries.

To restore the HA cluster one may need to manually invoke the `/high-availability/be-secondary-to` action.

Note #2: In the case where the failover-primary takes over as primary, it will enable read-only mode, if no secondary connects it will remain read-only. This is done to guarantee consistency.

Read-write mode can manually be enabled from the `/high-availability/read-only` action with the parameter `mode` passed with value `false`.

When any node loses connection, this can also be observed in high-availability alarms as either a `ha-primary-down` or a `ha-secondary-down` alarm.

```
alarms alarm-list alarm ncs ha-primary-down /high-availability/ha-node[id='paris']
```

```

is-cleared false
last-status-change 2022-05-30T10:02:45.706947+00:00
last-perceived-severity critical
last-alarm-text "Lost connection to primary due to: Primary closed connection"
status-change 2022-05-30T10:02:45.706947+00:00
received-time 2022-05-30T10:02:45.706947+00:00
perceived-severity critical
alarm-text "Lost connection to primary due to: Primary closed connection"

```

```

alarms alarm-list alarm ncs ha-secondary-down /high-availability/ha-node[id='london'] ""
is-cleared false
last-status-change 2022-05-30T10:04:33.231808+00:00
last-perceived-severity critical
last-alarm-text "Lost connection to secondary"
status-change 2022-05-30T10:04:33.231808+00:00
received-time 2022-05-30T10:04:33.231808+00:00
perceived-severity critical
alarm-text "Lost connection to secondary"

```

## Startup

Startup behaviour is defined by a combination of the parameters `/high-availability/settings/start-up/assume-nominal-role` and `/high-availability/settings/start-up/join-ha` as well as the nodes nominal role:

| assume-nominal-role | join-ha | nominal-role | behaviour                                                                                                                                                                                            |
|---------------------|---------|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| true                | false   | primary      | Assume primary role.                                                                                                                                                                                 |
| true                | false   | secondary    | Attempt to connect as secondary to the node (if any) which has nominal-role primary. If this fails, make no retry attempts and assume none role.                                                     |
| true                | false   | none         | Assume none role                                                                                                                                                                                     |
| false               | true    | primary      | Attempt to join HA setup as secondary by querying for current primary. Retries will be attempted. Retry attempt interval is defined by <code>/high-availability/settings/reconnect-interval</code> . |
| false               | true    | secondary    | Attempt to join HA setup as secondary by querying for current primary. Retries will be attempted. Retry attempt interval is defined by <code>/high-availability/settings/</code>                     |

|       |       |           |                                                                                                                                                                                                                                      |
|-------|-------|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|       |       |           | reconnect-interval. If all retry attempts fail, assume none role.                                                                                                                                                                    |
| false | true  | none      | Assume none role.                                                                                                                                                                                                                    |
| true  | true  | primary   | Query HA setup once for a node with primary role. If found, attempt to connect as secondary to that node. If no current primary is found, assume primary role.                                                                       |
| true  | true  | secondary | Attempt to join HA setup as secondary by querying for current primary. Retries will be attempted. Retry attempt interval is defined by /high-availability/settings/reconnect-interval. If all retry attempts fail, assume none role. |
| true  | true  | none      | Assume none role.                                                                                                                                                                                                                    |
| false | false | -         | Assume none role.                                                                                                                                                                                                                    |

## Actions

NSO rule-based HA can be controlled through a number of actions. All actions are found under /high-availability/. The available actions are listed below:

| Action          | Description                                                                                                                                                                      |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| be-primary      | Order the local node to assume ha role primary                                                                                                                                   |
| be-none         | Order the local node to assume ha role none                                                                                                                                      |
| be-secondary-to | Order the local node to connect as secondary to the provided HA node. This is an asynchronous operation, result can be found under /high-availability/status/be-secondary-result |
| local-node-id   | Identify the which of the nodes in /high-availability/ha-node (if any) corresponds to the local NSO instance                                                                     |
| enable          | Enable NSO rule-based HA and optionally assume a ha role according to /high-availability/settings/start-up/ parameters                                                           |
| disable         | Disable NSO rule-based HA and assume a ha role none                                                                                                                              |

## Status Check

The current state of NSO rule-based HA can be monitored by observing `/high-availability/status/`. Information can be found about current active HA mode and current assigned role. For nodes with active mode primary a list of connected nodes and their source IP addresses is shown. For nodes with assigned role secondary the latest result of the be-secondary operation is listed. All NSO rule-based HA status information is non-replicated operational data - the result here will differ between nodes connected in a HA setup.

## Tail-f HCC Package

### Overview

The Tail-f HCC package extends the built-in HA functionality by providing virtual IP addresses (VIPs) that can be used to connect to the NSO HA group primary node. HCC ensures that the VIP addresses are always bound by the HA group primary and never bound by a secondary. Each time a node transitions between primary and secondary states HCC reacts by binding (primary) or unbinding (secondary) the VIP addresses.

HCC manages IP addresses at link-layer (OSI layer 2) for Ethernet interfaces, and optionally, also at network-layer (OSI layer 3) using BGP router advertisements. The layer-2 and layer-3 functions are mostly independent and this document describes the details of each one separately. However, the layer-3 function builds on top of the layer-2 function. The layer-2 function is always necessary, otherwise, the Linux kernel on the primary node would not recognize the VIP address or accept traffic directed to it.



#### Note

Tail-f HCC version 5.x is non-backwards compatible with previous versions of Tail-f HCC and requires functionality provided by NSO version 5.4 and greater. For more details see [the following chapter](#).

## Dependencies

Both the HCC layer-2 VIP and layer-3 BGP functionality depend on **iproute2** utilities and **awk**. An optional dependency is **arping** (either from `iputils` or Thomas Habets `arping` implementation) which allows HCC to announce the VIP to MAC mapping to all nodes in the network by sending gratuitous ARP requests.

The HCC layer-3 BGP functionality depends on the `GoBGP` daemon version 2.x being installed on each NSO host that is configured to run HCC in BGP mode.

`GoBGP` is open source software originally developed by NTT Communications and released under the Apache License 2.0. `GoBGP` can be obtained directly from <https://osrg.github.io/gobgp/> and is also packaged for mainstream Linux distributions.

The HCC layer-3 DNS Update functionality depends on the command line utility **nsupdate**.

**Table 4. Tools Dependencies**

| Tool | Package  | Required | Description                                                 |
|------|----------|----------|-------------------------------------------------------------|
| ip   | iproute2 | yes      | Adds and deletes the virtual IP from the network interface. |

| Tool             | Package                     | Required | Description                                                                                                                                                |
|------------------|-----------------------------|----------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| awk              | mawk or gawk                | yes      | Installed with most Linux distributions.                                                                                                                   |
| sed              | sed                         | yes      | Installed with most Linux distributions.                                                                                                                   |
| arping           | iputils or arping           | optional | Installation recommended. Will reduce the propagation of changes to the virtual IP for layer-2 configurations.                                             |
| gobgpd and gobgp | GoBGP 2.x                   | optional | Required for layer-3 configurations. gobgpd is started by the HCC package and advertises the virtual IP using BGP. gobgp is used to get advertised routes. |
| nsupdate         | bind-tools or knot-dnsutils | optional | Required for layer-3 DNS update functionality and is used to submit Dynamic DNS Update requests to a name server.                                          |

Same as with built-in HA functionality, all NSO instances must be configured to run in HA mode. See [the following instructions](#) on how to enable HA on NSO instances.

## Running the HCC Package with NSO as a Non-Root User

GoBGP uses TCP port 179 for its communications and binds to it at startup. As port 179 is considered a privileged port it is normally required to run gobgpd as root.

When NSO is running as a non-root user the gobgpd command will be executed as the same user as NSO and will prevent gobgpd from binding to port 179.

There are multiple ways to handle this and two are listed here.

- 1 Set capability CAP\_NET\_BIND\_SERVICE on the gobgpd file. May not be supported by all Linux distributions.

```
$ sudo setcap 'cap_net_bind_service=+ep' /usr/bin/gobgpd
```

- 2 Set owner to root and the setuid bit of the gobgpd file. Works on all Linux distributions.

```
$ sudo chown root /usr/bin/gobgpd
$ sudo chmod u+s /usr/bin/gobgpd
```

- 3 The vipctl script, included in the HCC package, uses **sudo** to run the **ip** and **arping** commands when NSO is not running as root. If **sudo** is used, you must ensure it does not require password input. For example, if NSO runs as admin user, the sudoers file can be edited similarly to the following:

```
$ sudo echo "admin ALL = (root) NOPASSWD: /bin/ip" << /etc/sudoers
$ sudo echo "admin ALL = (root) NOPASSWD: /path/to/arping" << /etc/sudoers
```

## Tail-f HCC Compared with HCC Version 4.x and Older

### HA Group Management Decisions

Tail-f HCC 5.x or later does not participate in decisions on which NSO node is primary or secondary. These decisions are taken by NSO's built-in HA and then pushed as notifications to HCC. The NSO built-in HA functionality is available in NSO starting with version 5.4, where older NSO versions are not compatible with the HCC 5.x or later.

## Embedded BGP Daemon

HCC 5.x or later operates a GoBGP daemon as a subprocess completely managed by NSO. The old HCC function pack interacted with an external Quagga BGP daemon using a NED interface.

## Automatic Interface Assignment

HCC 5.x or later automatically associates VIP addresses with Linux network interfaces using the `ip` utility from the `iproute2` package. VIP addresses are also treated as `/32` without defining a new subnet. The old HCC function pack used explicit configuration to associate VIPs with existing addresses on each NSO host and define IP subnets for VIP addresses.

## Upgrading

Since version 5.0, HCC relies on the NSO built-in HA for cluster management and only performs address or route management in reaction to cluster changes. Therefore, no special measures are necessary if using HCC when performing an NSO version upgrade or a package upgrade. Instead, you should follow the standard best practice HA upgrade procedure from [the section called “NSO HA Version Upgrade”](#).

A reference to upgrade examples can be found in the NSO example set under `examples.ncs/development-guide/high-availability/hcc/README`.

## Layer-2

### Overview

The purpose of the HCC layer-2 functionality is to ensure that the configured VIP addresses are bound in the Linux kernel of the NSO primary node only. This ensures that the primary node (and only the primary node) will accept traffic directed toward the VIP addresses.

HCC also notifies the local layer-2 network when VIP addresses are bound by sending Gratuitous ARP (GARP) packets. Upon receiving the Gratuitous ARP, all the nodes in the network update their ARP tables with the new mapping so they can continue to send traffic to the non-failed, now primary node.

### Operational Details

HCC binds the VIP addresses as additional (alias) addresses on existing Linux network interfaces (e.g. `eth0`). The network interface for each VIP is chosen automatically by performing a kernel routing lookup on the VIP address. That is, the VIP will automatically be associated with the same network interface that the Linux kernel chooses to send traffic to the VIP.

This means that you can map each VIP onto a particular interface by defining a route for a subnet that includes the VIP. If no such specific route exists the VIP will automatically be mapped onto the interface of the default gateway.



#### Note

To check which interface HCC will choose for a particular VIP address simply run for example

```
admin@paris:~$ ip route get 192.168.123.22
```

and look at the device `dev` in the output, for example `eth0`.

## Configuration

The layer-2 functionality is configured by providing a list of IPv4 and/or IPv6 VIP addresses and enabling HCC. The VIP configuration parameters are found under `/hcc:hcc`.



Table 5. Global Layer-2 Configuration

| Parameters  | Type                    | Description                                                                                                                                                                                                                                |
|-------------|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| enabled     | boolean                 | If set to 'true', the primary node in an HA group automatically binds the set of Virtual IPv[46] addresses.                                                                                                                                |
| vip-address | list of inet:ip-address | The list of virtual IPv[46] addresses to bind on the primary node. The addresses are automatically unbound when a node becomes secondary. The addresses can therefore be used externally to reliably connect to the HA group primary node. |

## Example Configuration

```
admin@ncs(config)# hcc enabled
admin@ncs(config)# hcc vip 192.168.123.22
admin@ncs(config)# hcc vip 2001:db8::10
admin@ncs(config)# commit
```

## Layer-3 BGP

### Overview

The purpose of the HCC layer-3 BGP functionality is to operate a BGP daemon on each NSO node and to ensure that routes for the VIP addresses are advertised by the BGP daemon on the primary node only.

The layer-3 functionality is an optional add-on to the layer-2 functionality. When enabled, the set of BGP neighbors must be configured separately for each NSO node. Each NSO node operates an embedded BGP daemon and maintains connections to peers but only the primary node announces the VIP addresses.

The layer-3 functionality relies on the layer-2 functionality to assign the virtual IP addresses to one of the host's interfaces. One notable difference in assigning virtual IP addresses when operating in Layer-3 mode is that the virtual IP addresses are assigned to the loopback interface `lo` rather than to a specific physical interface.

### Operational Details

HCC operates a GoBGP subprocess as an embedded BGP daemon. The BGP daemon is started, configured, and monitored by HCC. The HCC YANG model includes basic BGP configuration data and state data.

Operational data in the YANG model includes the state of the BGP daemon subprocess and the state of each BGP neighbor connection. The BGP daemon writes log messages directly to NSO where the HCC module extracts updated operational data and then repeats the BGP daemon log messages into the HCC log verbatim. You can find these log messages in the developer log (`developer.log`).

```
admin@ncs# show hcc
NODE BGPD BGPD
ID PID STATUS ADDRESS STATE CONNECTED

london - - 192.168.30.2 - -
paris 827 running 192.168.31.2 ESTABLISHED true
```

**Note**

GoBGP must be installed separately. The gobgp and gobgpd binaries must be found in paths specified by the \$PATH environment variable. For system install NSO reads \$PATH in the systemd init script /etc/init.d/ncs. Since tailf-hcc 6.0.2, the path to gobgp/gobgpd is no longer possible to specify from the configuration data leaf `/hcc/bgp/node/gobgp-bin-dir`. The leaf has been removed from the tailf-hcc/src/yang/tailf-hcc.yang module.

Upgrades: If BGP is enabled and the gobgp or gobgpd binaries are not found, the tailf-hcc package will fail to load. The user must then install GoBGP and invoke the **packages reload** action or restart NSO with `NCS_RELOAD_PACKAGES=true /etc/init.d/ncs restart`.

## Configuration

The layer-3 BGP functionality is configured as a list of BGP configurations with one list entry per node. Configurations are separate because each NSO node usually has different BGP neighbors with their own IP addresses, authentication parameters, etc.

The BGP configuration parameters are found under `/hcc:hcc/bgp/node{id}`.

**Table 6. Per-Node Layer-3 Configuration**

| Parameters | Type            | Description                                                                                            |
|------------|-----------------|--------------------------------------------------------------------------------------------------------|
| node-id    | string          | Unique node ID. A reference to <code>/ncs:high-availability/ha-node/id</code> .                        |
| enabled    | boolean         | If set to <code>true</code> this node uses BGP to announce VIP addresses when in the HA primary state. |
| as         | inet:as-number  | The BGP Autonomous System Number for the local BGP daemon.                                             |
| router-id  | inet:ip-address | The router-id for the local BGP daemon.                                                                |

Each NSO node can connect to a different set of BGP neighbors. For each node, the BGP neighbor list configuration parameters are found under `/hcc:hcc/bgp/node{id}/neighbor{address}`.

**Table 7. Per-Neighbor BGP Configuration**

| Parameters | Type            | Description                                                                                                        |
|------------|-----------------|--------------------------------------------------------------------------------------------------------------------|
| address    | inet:ip-address | BGP neighbor IP address.                                                                                           |
| as         | inet:as-number  | BGP neighbor Autonomous System Number.                                                                             |
| ttl-min    | uint8           | Optional minimum TTL value for BGP packets. When configured enables BGP Generalized TTL Security Mechanism (GTSM). |
| password   | string          | Optional password to use for BGP authentication with this neighbor.                                                |
| enabled    | boolean         | If set to <code>true</code> then an outgoing BGP connection to this neighbor                                       |

| Parameters | Type | Description                                  |
|------------|------|----------------------------------------------|
|            |      | is established by the HA group primary node. |

## Example

```
admin@ncs(config)# hcc bgp node paris enabled
admin@ncs(config)# hcc bgp node paris as 64512
admin@ncs(config)# hcc bgp node paris router-id 192.168.31.99
admin@ncs(config)# hcc bgp node paris gobgp-bindir /usr/bin
admin@ncs(config)# hcc bgp node paris neighbor 192.168.31.2 as 64514
admin@ncs(config)# ... repeated for each neighbor if more than one ...
... repeated for each node ...
admin@ncs(config)# commit
```

## Layer-3 DNS update

### Overview

The purpose of the HCC layer-3 DNS Update functionality is to notify a DNS server of the IP address change of the active primary NSO server, allowing the DNS server to update the DNS record for the given domain name.

Geographically redundant NSO setup typically relies on DNS support. To enable this use case, tailf-hcc can dynamically update DNS with the **nsupdate** utility on HA status change notification.

The DNS server used should support updates through nsupdate command (RFC 2136).

### Operational Details

HCC listens on the underlying NSO HA notifications stream. When HCC receives a notification about an NSO node being Primary, it updates the DNS Server with the IP address of the Primary NSO for the given hostname. The HCC YANG model includes basic DNS configuration data and operational status data.

Operational data in the YANG model includes the result of the latest DNS update operation.

```
admin@ncs# show hcc dns
hcc dns status time 2023-10-20T23:16:33.472522+00:00
hcc dns status exit-code 0
```

If the DNS Update is unsuccessful, an error message will be populated in operational data, for example:

```
admin@ncs# show hcc dns
hcc dns status time 2023-10-20T23:36:33.372631+00:00
hcc dns status exit-code 2
hcc dns status error-message "; Communication with 10.0.0.10#53 failed: timed out"
```



#### Note

The DNS Server must be installed and configured separately, and details are provided to HCC as configuration data. The DNS Server must be configured to update the reverse DNS record.

## Configuration

The layer-3 DNS Update functionality needs DNS-related information like DNS server IP address, port, zone, etc, and information about NSO nodes involved in HA - node, ip, and location.

The DNS configuration parameters are found under `/hcc:hcc/dns`.

**Table 8. Layer-3 DNS Configuration**

| Parameters | Type             | Description                                                |
|------------|------------------|------------------------------------------------------------|
| enabled    | boolean          | If set to <code>true</code> DNS updates will be enabled.   |
| fqdn       | inet:domain-name | DNS domain-name for the HA primary.                        |
| ttl        | uint32           | Time to live for DNS record, default 86400.                |
| key-file   | string           | Specifies the file path for <code>nsupdate</code> keyfile. |
| server     | inet:ip-address  | DNS Server IP Address.                                     |
| port       | uint32           | DNS Server port, default 53.                               |
| zone       | inet:host        | DNS Zone to update on the server.                          |
| timeout    | uint32           | Timeout for <code>nsupdate</code> command, default 300.    |

Each NSO node can be placed in a separate Location/Site/Availability-Zone. This is configured as a list member configuration, with one list entry per node-id. The member list configuration parameters are found under `/hcc:hcc/dns/member{node-id}`.

**Table 9. Layer-3 DNS member Configuration**

| Parameters | Type            | Description                                                                                                                                                   |
|------------|-----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| node-id    | string          | Unique NSO HA node ID. Valid values are: <code>/high-availability/ha-node</code> when built-in HA is used or <code>/ha-raft/status/member</code> for HA Raft. |
| ip-address | inet:ip-address | IP where NSO listens for incoming requests to any northbound interfaces.                                                                                      |
| location   | string          | Name of the Location/Site/Availability-Zone where node is placed.                                                                                             |

## Example

Here is an example configuration for a setup of two dual-stack NSO nodes, node-1 and node-2, that have an IPv4 and an IPv6 address configured. The configuration also sets up update signing with the specified key.

```
admin@ncs(config)# hcc dns enabled
admin@ncs(config)# hcc dns fqdn example.com
admin@ncs(config)# hcc dns ttl 120
admin@ncs(config)# hcc dns key-file /home/cisco/DNS-testing/good.key
admin@ncs(config)# hcc dns server 10.0.0.10
admin@ncs(config)# hcc dns port 53
admin@ncs(config)# hcc dns zone zone1.nso
admin@ncs(config)# hcc dns member node-1 ip-address [10.0.0.20 ::10]
admin@ncs(config)# hcc dns member node-1 location SanJose
admin@ncs(config)# hcc dns member node-2 ip-address [10.0.0.30 ::20]
admin@ncs(config)# hcc dns member node-2 location NewYork
admin@ncs(config)# commit
```

## Usage

This chapter describes basic deployment scenarios for HCC. Layer-2 mode is demonstrated first and then the layer-3 BGP functionality is configured in addition. A reference to container-based examples for the layer-2 and layer-3 deployment scenarios described here can be found in the NSO example set under `examples.ncs/development-guide/high-availability/hcc`.

Both scenarios consist of two test nodes: `london` and `paris` with a single IPv4 VIP address. For the layer-2 scenario, the nodes are on the same network. The layer-3 scenario also involves a BGP-enabled router node as the `london` and `paris` nodes are on two different networks.

## Layer-2 Deployment

The layer-2 operation is configured by simply defining the VIP addresses and enabling HCC. The HCC configuration on both nodes should match, otherwise, the primary node's configuration will overwrite the secondary node configuration when the secondary connects to the primary node.

**Table 10. Addresses**

| Hostname | Address        | Role                               |
|----------|----------------|------------------------------------|
| paris    | 192.168.23.99  | Paris service node.                |
| london   | 192.168.23.98  | London service node.               |
| vip4     | 192.168.23.122 | NSO primary node IPv4 VIP address. |

## Configuring VIPs

```
admin@ncs(config)# hcc enabled
admin@ncs(config)# hcc vip 192.168.23.122
admin@ncs(config)# commit
```

## Verifying VIP Availability

Once enabled, HCC on the HA group primary node will automatically assign the VIP addresses to corresponding Linux network interfaces.

```
root@paris:/var/log/ncs# ip address list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
 link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
 inet 127.0.0.1/8 scope host lo
 valid_lft forever preferred_lft forever
 inet6 ::1/128 scope host
 valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default
 link/ether 52:54:00:fa:61:99 brd ff:ff:ff:ff:ff:ff
 inet 192.168.23.99/24 brd 192.168.23.255 scope global enp0s3
 valid_lft forever preferred_lft forever
 inet 192.168.23.122/32 scope global enp0s3
 valid_lft forever preferred_lft forever
 inet6 fe80::5054:ff:fefa:6199/64 scope link
 valid_lft forever preferred_lft forever
```

On the secondary node HCC will not configure these addresses.

```
root@london:~# ip address list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 ...
 link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
 inet 127.0.0.1/8 scope host lo
 valid_lft forever preferred_lft forever
```

```

 inet6 ::1/128 scope host
 valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 ...
 link/ether 52:54:00:fa:61:98 brd ff:ff:ff:ff:ff:ff
 inet 192.168.23.98/24 brd 192.168.23.255 scope global enp0s3
 valid_lft forever preferred_lft forever
 inet6 fe80::5054:ff:fe6a:6198/64 scope link
 valid_lft forever preferred_lft forever

```

## Layer-2 Example Implementation

A reference to a container-based example of the layer-2 scenario can be found in the NSO example set under `examples.ncs/development-guide/high-availability/hcc/README`.

## Enabling Layer-3 BGP

Layer-3 operation is configured for each NSO HA group node separately. The HCC configuration on both nodes should match, otherwise, the primary node's configuration will overwrite the configuration on the secondary node.

**Table 11. Addresses**

| Hostname | Address                      | AS    | Role                          |
|----------|------------------------------|-------|-------------------------------|
| paris    | 192.168.31.99                | 64512 | Paris node                    |
| london   | 192.168.30.98                | 64513 | London node                   |
| router   | 192.168.30.2<br>192.168.31.2 | 64514 | BGP-enabled router            |
| vip4     | 192.168.23.122               |       | Primary node IPv4 VIP address |

## Configuring BGP for Paris Node

```

admin@ncs(config)# hcc bgp node paris enabled
admin@ncs(config)# hcc bgp node paris as 64512
admin@ncs(config)# hcc bgp node paris router-id 192.168.31.99
admin@ncs(config)# hcc bgp node paris gobgp-bindir /usr/bin
admin@ncs(config)# hcc bgp node paris neighbor 192.168.31.2 as 64514
admin@ncs(config)# commit

```

## Configuring BGP for London Node

```

admin@ncs(config)# hcc bgp node london enabled
admin@ncs(config)# hcc bgp node london as 64513
admin@ncs(config)# hcc bgp node london router-id 192.168.30.98
admin@ncs(config)# hcc bgp node london gobgp-bindir /usr/bin
admin@ncs(config)# hcc bgp node london neighbor 192.168.30.2 as 64514
admin@ncs(config)# commit

```

## Check BGP Neighbor Connectivity

Check neighbor connectivity on the `paris` primary node. Note that its connection to neighbor 192.168.31.2 (router) is ESTABLISHED.

```

admin@ncs# show hcc
 BGPD BGPD
NODE ID PID STATUS ADDRESS STATE CONNECTED

london - - 192.168.30.2 - -
paris 2486 running 192.168.31.2 ESTABLISHED true

```

Check neighbor connectivity on the london secondary node. Note that the primary node also has an ESTABLISHED connection to its neighbor 192.168.30.2 (router). The primary and secondary nodes both maintain their BGP neighbor connections at all times when BGP is enabled, but only the primary node announces routes for the VIPs.

```
admin@ncs# show hcc
 BGPD BGPD
NODE ID PID STATUS ADDRESS STATE CONNECTED

london 494 running 192.168.30.2 ESTABLISHED true
paris - - 192.168.31.2 - -
```

## Check Advertised BGP Routes Neighbors

Check the BGP routes received by the router.

```
admin@ncs# show ip bgp
...
Network Next Hop Metric LocPrf Weight Path
*> 192.168.23.122/32
 192.168.31.99 0 64513 ?
```

The VIP subnet is routed to the paris host, which is the primary node.

## Layer-3 BGP Example Implementation

A reference to a container-based example of the combined layer-2 and layer-3 BGP scenario can be found in the NSO example set under `examples.ncs/development-guide/high-availability/hcc/README`.

## Enabling Layer-3 DNS

If enabled prior to the HA being established, HCC will update the DNS server with IP address of the Primary node once a primary is selected.

If an HA is already operational, and Layer-3 DNS is enabled and configured afterward, HCC will not update the DNS server automatically. An automatic DNS server update will only happen if a HA switchover happens. HCC exposes an update action to manually trigger an update to the DNS server with the IP address of the primary node.

## DNS Update Action

User can explicitly update DNS from the specific NSO node by running the update action.

```
admin@ncs# hcc dns update
```

Check the result of invoking the DNS update utility using the operational data in `/hcc/dns`:

```
admin@ncs# show hcc dns
hcc dns status time 2023-10-10T20:47:31.733661+00:00
hcc dns status exit-code 0
hcc dns status error-message ""
```

One way to verify DNS server updates is through the **nslookup** program. However, be mindful of the DNS caching mechanism, which may cache the old value for the amount of time controlled by the TTL setting.

```
cisco@node-2:~$ nslookup example.com
Server: 10.0.0.10
Address: 10.0.0.10#53

Name: example.com
Address: 10.0.0.20
Name: example.com
```

```
Address: ::10
```

## DNS get-node-location Action

/hcc/dns/member holds the information about all members involved in HA. The **get-node-location** action provides information on the location of an NSO node.

```
admin@ncs(config)# hcc dns get-node-location
location SanJose
```

## Data Model

### Tail-f HCC Model

```
module tailf-hcc {
 yang-version 1.1;
 namespace "http://cisco.com/pkg/tailf-hcc";
 prefix hcc;

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

 organization "Cisco Systems";
 description
 "This module defines Layer-2 and Layer-3 virtual IPv4 and IPv6 address
 (VIP) management for clustered operation.";

 revision 2024-03-18 {
 description
 "Removed leaf /hcc/bgp/node/gobgp-bin-dir,
 Added validation callpoint hcc-validate-gobgp.

 Released as part of tailf-hcc 6.0.2.";
 }

 revision 2023-08-31 {
 description
 "Added /hcc/dns for DNS update and Site location support.

 Released as part of tailf-hcc 6.0.1.";
 }

 revision 2023-06-26 {
 description
 "Added support for HA Raft, changed /hcc/bgp/node leafref to
 completion point and validation callback.

 Released as part of tailf-hcc 6.0.0.";
 }

 revision 2022-09-28 {
```



```

 description
 "Use bias-free language.

 Released as part of tailf-hcc 5.0.4.";
}

revision 2020-06-29 {
 description "Released as part of tailf-hcc 5.0.";
}

container hcc {
 description "Tail-f HCC package configuration.";
 leaf enabled {
 type boolean;
 default "false";
 description
 "If set to 'true', the primary node in a cluster automatically
 binds the set of Virtual IPv4 and IPv6 addresses.";
 }

 leaf-list vip-address {
 type inet:ip-address;
 tailf:info "IPv4/IPv6 VIP address list";
 description
 "The list of virtual IPv4 and IPv6 addresses to bind on the primary
 node. The addresses are automatically unbound when a node
 becomes secondary. The addresses can therefore be used externally
 to reliably connect to the primary node in the cluster.";
 }

 action update {
 tailf:actionpoint hcc-action;
 tailf:info "Update VIP routes";
 description
 "Update VIP address configuration in the Linux kernel.
 Generally this is not necessary but can be useful if the VIP
 addresses have been disturbed in some way e.g. if network
 configuration on the host has been completely reset.";
 output {
 leaf status {
 type string;
 }
 }
 }
}

container bgp {
 tailf:info "VIP announcement over BGP";
 description
 "Run a local BGP daemon and advertise VIP routes to neighbors.";

 list node {
 tailf:validate hcc-validate-gobgp {
 tailf:dependency '.';
 tailf:dependency '../enabled';
 }
 key "node-id";
 description
 "Unique NCS HA node ID. Valid values are:
 - /high-availability/ha-node when built-in HA is used or
 - /ha-raft/status/member for HA Raft.";
 leaf node-id {
 tailf:cli-completion-actionpoint "hcc-complete-members";
 }
 }
}

```

```

 tailf:validate hcc-validate-members {
 tailf:dependency '.';
 }
 type string;
}

leaf enabled {
 type boolean;
 default true;
 description
 "If set to 'true' this node uses BGP to announce VIP
 addresses in the primary state.";
}

leaf as {
 type inet:as-number;
 mandatory true;
 tailf:info "BGP Autonomous System Number";
 description
 "The BGP Autonomous System Number for the local BGP daemon.";
}

leaf router-id {
 type inet:ip-address;
 mandatory true;
 tailf:info "Local BGP router ID";
 description
 "The router-id for the local BGP daemon.";
}

leaf bgpd-pid {
 type uint32;
 config false;
 tailf:callpoint hcc-data;
 tailf:info "PID of BGP daemon process";
 description
 "Unix PID of the local BGP daemon process (when running).";
}

leaf bgpd-status {
 type string;
 config false;
 tailf:callpoint hcc-data;
 tailf:info "Status of BGP daemon process";
 description
 "String describing the current status of the local BGP
 daemon process.";
}

list neighbor {
 key "address";
 description "BGP neighbor list";
 leaf address {
 type inet:ip-address;
 mandatory true;
 description "BGP neighbor IP address";
 }
 leaf as {
 type inet:as-number;
 mandatory true;
 description "BGP neighbor Autonomous System number";
 }
}

```

```

 leaf ttl-min {
 type uint8;
 description
 "Optional minimum TTL value for BGP packets. When configured
 enables BGP Generalized TTL Security Mechanism (GTSM).";
 }
 leaf multihop-ttl{
 type uint8;
 description "eBGP multihop TTL";
 }
 leaf password {
 type string;
 tailf:info "Optional BGP MD5 auth password.";
 description
 "Optional password to use for BGP authentication with this
 neighbor.";
 }
 leaf enabled {
 type boolean;
 default "true";
 description
 "If set to 'true' then an outgoing BGP connection to this
 neighbor is established by the cluster primary.";
 }
 leaf state {
 type string;
 config false;
 tailf:callpoint hcc-data;
 tailf:info "State of BGP neighbor connection";
 description
 "String describing the current state of the BGP connection
 from the local BGP daemon to this neighbor.";
 }
 leaf connected {
 type boolean;
 config false;
 tailf:callpoint hcc-data;
 tailf:info "BGP session establishment status";
 description
 "Flag indicating whether the BGP session to this neighbor
 is currently established.";
 }
 }
}

container dns {
 tailf:info "DNS Config for dynamic DNS updates";
 description "DNS Config for dynamic DNS updates";
 presence true;
 leaf enabled {
 type boolean;
 default false;
 description
 "If set to 'true' dns updates will be enabled";
 }
 leaf fqdn {
 tailf:info "DNS domain-name of the HA primary";
 type inet:domain-name;
 mandatory true;
 }
 leaf ttl {

```

```

 tailf:info "Time to live for DNS record";
 type uint32;
 default 86400;
}
leaf key-file {
 tailf:info "Specifies the file path for nsupdate keyfile";
 type string;
}
leaf server {
 tailf:info "DNS server";
 type inet:ip-address;
}
leaf port {
 tailf:info "DNS server port";
 when "../server";
 type uint32;
 default 53;
}
leaf zone {
 tailf:info "DNS zone";
 type inet:host;
}
leaf timeout {
 tailf:info "Timeout for nsupdate command";
 type uint32;
 default 300;
}

container status {
 config false;
 leaf time {
 type yang:date-and-time;
 }
 leaf exit-code {
 tailf:info "Exit code returned by os upon executing nsupdate";
 type uint32;
 }
 leaf error-message {
 tailf:info "Status message returned by os upon executing nsupdate when exit-code is no";
 type string;
 }
}

list member {
 key "node-id";
 description "Details about all members involved in HA - nso node, ip and location.";
 min-elements 1;
 leaf node-id {
 description
 "Unique NCS HA node ID. Valid values are:
 - /high-availability/ha-node when built-in HA is used or
 - /ha-raft/status/member for HA Raft.";
 tailf:cli-completion-actionpoint "hcc-complete-members";
 tailf:validate hcc-validate-dns-member {
 tailf:dependency '.';
 tailf:dependency '../..//member';
 }
 type string;
 }
 leaf-list ip-address {
 type inet:ip-address;
 description "IP where NSO listens for incoming request to any northbound interfaces";
 }
}

```

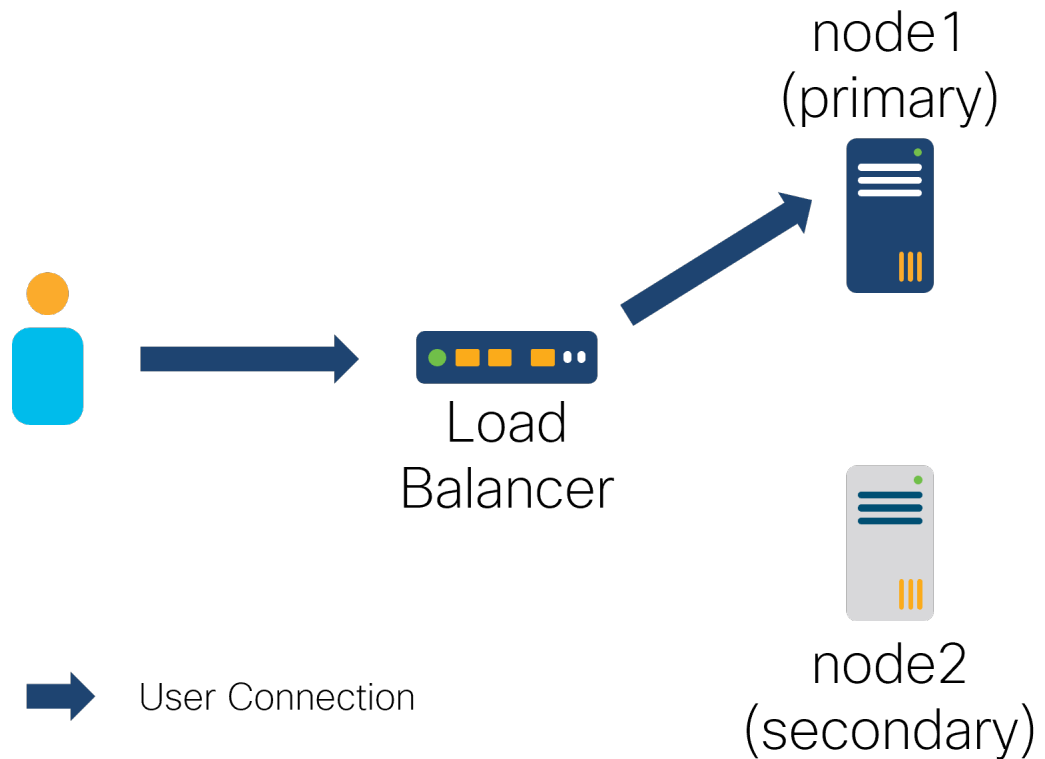
```
tailf:info "IP where NSO listens for incoming request to any northbound interfaces"
min-elements 1;
}
leaf location {
 type string;
 description "Name of the Location/Site/Availability-Zone where node is placed";
 tailf:info "Name of the Location/Site/Availability-Zone where node is placed";
}
}

action update {
 tailf:actionpoint hcc-dns-update-action;
 description "Update DNS RR entry";
 tailf:info "Manually retry update of the DNS RR configuration to the DNS server.
 This can help if the automatic DNS update failed for some reason.
 Reason of failure is specified in the /hcc/dns/status/error-message";
 output {
 leaf error-status {
 type string;
 }
 }
}

action get-node-location {
 tailf:actionpoint hcc-get-node-location-action;
 tailf:info "Returns the location of node";
 description "Returns the location of node";
 output {
 leaf location {
 type string;
 }
 leaf error-status {
 type string;
 }
 }
}
}
```

## Setup with an External Load Balancer

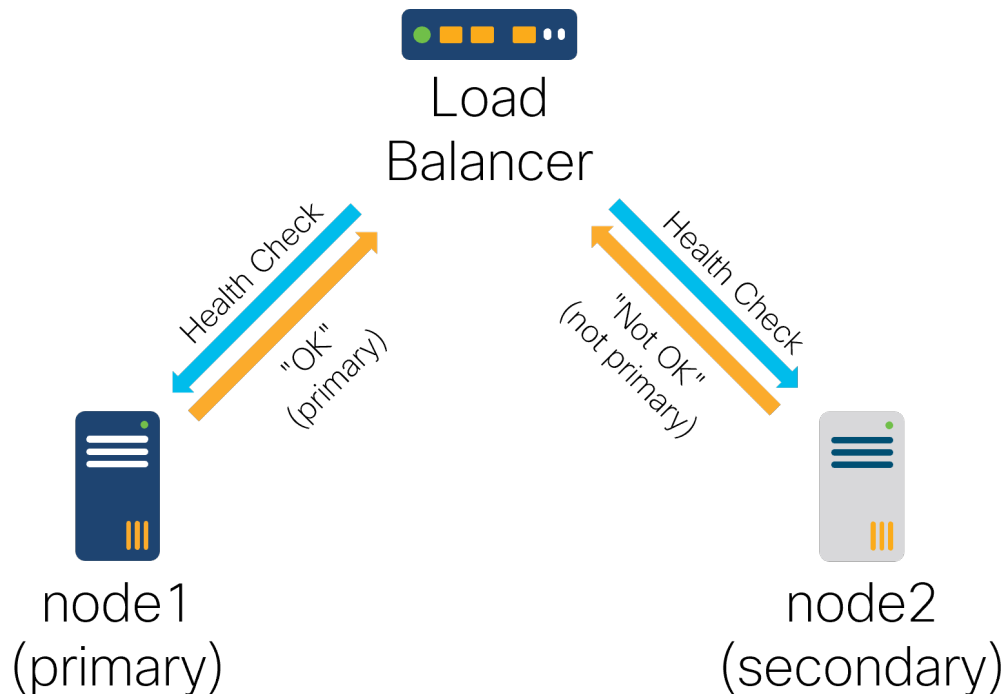
As an alternative to the HCC package, NSO built-in HA, either rule-based or HA Raft, can also be used in conjunction with a load balancer device in a reverse proxy configuration. Instead of managing the virtual IP address directly as HCC does, this setup relies on an external load balancer to route traffic to the currently active primary node.

**Figure 12. Load balancer routes connections to the appropriate NSO node**

The load balancer uses HTTP health checks to determine which node is currently the active primary. The example, found in the `examples.ncs/development-guide/high-availability/load-balancer` directory, uses HTTP status codes on the health check endpoint to easily distinguish whether the node is currently primary or not.

In the example, freely available HAProxy software is used as a load balancer to demonstrate the functionality. It is configured to steer connections on localhost to either the TCP port 2024 (SSH CLI) and TCP port 8080 (web UI and RESTCONF) to the active node in a 2-node HA cluster. The HAProxy software is required if you wish to run this example yourself.

Figure 13. Load balancer uses health checks to determine the currently active primary node



You can start all the components in the example by running the **make build start** command. At the beginning, the first node *n1* is the active primary. Connecting to the localhost port 2024 will establish connection to this node:

```

$ make build start
Setting up run directory for nso-nodel
... make output omitted ...
Waiting for n2 to connect: .
$ ssh -p 2024 admin@localhost
admin@localhost's password: admin

admin connected from 127.0.0.1 using ssh on localhost
admin@n1> switch cli
admin@n1# show high-availability
high-availability enabled
high-availability status mode primary
high-availability status current-id n1
high-availability status assigned-role primary
high-availability status read-only-mode false
ID ADDRESS

n2 127.0.0.1

```

Then, you can disable the high availability subsystem on *n1* to simulate a node failure.

```

admin@n1# high-availability disable
result NSO Built-in HA disabled
admin@n1# exit
Connection to localhost closed.

```

Disconnect and wait a few seconds for the build-in HA to perform the fail over to node *n2*. The time depends on the `high-availability/settings/reconnect-interval` and is set quite

aggressively in this example to make the fail over in about 6 seconds. Reconnect with the SSH client and observe the connection is now made to the fail-over node which has become the active primary:

```
$ ssh -p 2024 admin@localhost
admin@localhost's password: admin

admin connected from 127.0.0.1 using ssh on localhost
admin@n2> switch cli
admin@n2# show high-availability
high-availability enabled
high-availability status mode primary
high-availability status current-id n2
high-availability status assigned-role primary
high-availability status read-only-mode false
```

Finally, shut down the example with the **make stop clean** command.

## NB listen addresses on HA primary for Load Balancers

NSO can be configured for the HA primary to listen on additional ports for the northbound interfaces NETCONF, RESTCONF, the web server (including JSON-RPC) and the CLI over SSH. Once a different node transitions to role primary the configured listen addresses are brought up on that node instead.

when the following configuration is added to `ncs.conf`, then the primary HA node will listen(2) and bind(2) port 1830 on the wildcard IPv4 and IPv6 addresses.

```
<netconf-north-bound>
 <transport>
 <ssh>
 <enabled>true</enabled>
 <ip>0.0.0.0</ip>
 <port>830</port>
 <ha-primary-listen>
 <ip>0.0.0.0</ip>
 <port>1830</port>
 </ha-primary-listen>
 <ha-primary-listen>
 <ip>:::</ip>
 <port>1830</port>
 </ha-primary-listen>
 </ssh>
 </transport>
</netconf-north-bound>
```

similar configuration can be added for other NB interfaces, see the `ha-primary-listen` list under `/ncs-config/{restconf,webui,cli}`.

## HA framework requirements

If an external HAFW is used, NSO only replicates the CDB data. NSO must be told by the HAFW which node should be primary and which nodes should be secondaries.

The HA framework must also detect when nodes fail and instruct NSO accordingly. If the primary node fails, the HAFW must elect one of the remaining secondaries and appoint it the new primary. The remaining secondaries must also be informed by the HAFW about the new primary situation.



# Mode of operation

NSO must be instructed through the `ncs.conf` configuration file that it should run in HA mode. The following configuration snippet enables HA mode:

```
<ha>
 <enabled>true</enabled>
 <ip>0.0.0.0</ip>
 <port>4570</port>
 <extra-listen>
 <ip>:::</ip>
 <port>4569</port>
 </extra-listen>
 <tick-timeout>PT20S</tick-timeout>
</ha>
```



## Note

Make sure to restart the **ncs** process in order for the changes to take effect.

The IP address and the port above indicates which IP and which port should be used for the communication between the HA nodes. `extra-listen` is an optional list of `ip:port` pairs which a HA primary also listens to for secondary connections. For IPv6 addresses, the syntax `[ip]:port` may be used. If the `":port"` is omitted, `port` is used. The `tick-timeout` is a duration indicating how often each secondary must send a tick message to the primary indicating liveness. If the primary has not received a tick from a secondary within 3 times the configured tick time, the secondary is considered to be dead. Similarly, the primary sends tick messages to all the secondaries. If a secondary has not received any tick messages from the primary within the 3 times the timeout, the secondary will consider the primary dead and report accordingly.

A HA node can be in one of three states: `NONE`, `SECONDARY` or `PRIMARY`. Initially a node is in the `NONE` state. This implies that the node will read its configuration from CDB, stored locally on file. Once the HA framework has decided whether the node should be a secondary or a primary the HAFW must invoke either the methods `Ha.beSecondary(primary)` or `Ha.bePrimary()`

When a NSO HA node starts, it always starts up in mode `NONE`. At this point there are no other nodes connected. Each NSO node reads its configuration data from the locally stored CDB and applications on or off the node may connect to NSO and read the data they need. Although write operations are allowed in the `NONE` state it is highly discouraged to initiate southbound communication unless necessary. A node in `NONE` state should only be used to configure NSO itself or to do maintenance such as upgrades. When in `NONE` state, some features are disabled, including but not limited to:

- commit queue
- NSO scheduler
- nano-service side effect queue

This is in order to avoid situations where multiple NSO nodes are trying to perform the same southbound operation simultaneously.

At some point, the HAFW will command some nodes to become secondary nodes of a named primary node. When this happens, each secondary node tracks changes and (logically or physically) copies all the data from the primary. Previous data at the secondary node is overwritten.

Note that the HAFW, by using NSO's start phases, can make sure that NSO does not start its northbound interfaces (NETCONF, CLI, ...) until the HAFW has decided what type of node it is. Furthermore once a

node has been set to the `SECONDARY` state, it is not possible to initiate new write transactions towards the node. It is thus never possible for an agent to write directly into a secondary node. Once a node is returned either to the `NONE` state or to the `PRIMARY` state, write transactions can once again be initiated towards the node.

The HAFW may command a secondary node to become primary at any time. The secondary node already has up-to-date data, so it simply stops receiving updates from the previous primary. Presumably, the HAFW also commands the primary node to become a secondary node, or takes it down or handles the situation somehow. If it has crashed, the HAFW tells the secondary to become primary, restarts the necessary services on the previous primary node and gives it an appropriate role, such as secondary. This is outside the scope of NSO.

Each of the primary and secondary nodes have the same set of all callpoints and validation points locally on each node. The start sequence has to make sure the corresponding daemons are started before the HAFW starts directing secondary nodes to the primary, and before replication starts. The associated callbacks will however only be executed at the primary. If e.g. the validation executing at the primary needs to read data which is not stored in the configuration and only available on another node, the validation code must perform any needed RPC calls.

If the order from the HAFW is to become primary, the node will start to listen for incoming secondaries at the `ip:port` configured under `/ncs-config/ha`. The secondaries TCP connect to the primary and this socket is used by NSO to distribute the replicated data.

If the order is to be a secondary, the node will contact the primary and possibly copy the entire configuration from the primary. This copy is not performed if the primary and secondary decide that they have the same version of the CDB database loaded, in which case nothing needs to be copied. This mechanism is implemented by use of a unique token, the "transaction id" - it contains the node id of the node that generated it and a time stamp, but is effectively "opaque".

This transaction id is generated by the cluster primary each time a configuration change is committed, and all nodes write the same transaction id into their copy of the committed configuration. If the primary dies, and one of the remaining secondaries is appointed new primary, the other secondaries must be told to connect to the new primary. They will compare their last transaction id to the one from the newly appointed primary. If they are the same, no CDB copy occurs. This will be the case unless a configuration change has sneaked in, since both the new primary and the remaining secondaries will still have the last transaction id generated by the old primary - the new primary will not generate a new transaction id until a new configuration change is committed. The same mechanism works if a secondary node is simply restarted. In fact no cluster reconfiguration will lead to a CDB copy unless the configuration has been changed in between.

Northbound agents should run on the primary, it is not possible for an agent to commit write operations at a secondary node.

When an agent commits its CDB data, CDB will stream the committed data out to all registered secondaries. If a secondary dies during the commit, nothing will happen, the commit will succeed anyway. When and if the secondary reconnects to the cluster, the secondary will have to copy the entire configuration. All data on the HA sockets between NSO nodes only go in the direction from the primary to the secondaries. A secondary which isn't reading its data will eventually lead to a situation with full TCP buffers at the primary. In principle it is the responsibility of HAFW to discover this situation and notify the primary NSO about the hanging secondary. However if 3 times the tick timeout is exceeded, NSO will itself consider the node dead and notify the HAFW. The default value for tick timeout is 20 seconds.

The primary node holds the active copy of the entire configuration data in CDB. All configuration data has to be stored in CDB for replication to work. At a secondary node, any request to read will be serviced while write requests will be refused. Thus, CDB subscription code works the same regardless of whether

the CDB client is running at the primary or at any of the secondaries. Once a secondary has received the updates associated to a commit at the primary, all CDB subscribers at the secondary will be duly notified about any changes using the normal CDB subscription mechanism.

If the system has been setup to subscribe for NETCONF notifications, the secondaries will have all subscriptions as configured in the system, but the subscription will be idle. All NETCONF notifications are handled by the primary, and once the notifications get written into stable storage (CDB) at the primary, the list of received notifications will be replicated to all secondaries.

## Security aspects

We specify in `ncs.conf` which IP address the primary should bind for incoming secondaries. If we choose the default value `0.0.0.0` it is the responsibility of the application to ensure that connection requests only arrive from acceptable trusted sources through some means of firewalling.

A cluster is also protected by a token, a secret string only known to the application. The `Ha.connect()` method must be given the token. A secondary node that connects to a primary node negotiates with the primary using a CHAP-2 like protocol, thus both the primary and the secondary are ensured that the other end has the same token without ever revealing their own token. The token is never sent in clear text over the network. This mechanism ensures that a connection from a NSO secondary to a primary can only succeed if they both have the same token.

It is indeed possible to store the token itself in CDB, thus an application can initially read the token from the local CDB data, and then use that token in the constructor for the `Ha` class. In this case it may very well be a good idea to have the token stored in CDB be of type `tailf:aes-256-cfb-128-encrypted-string`.

If the actual CDB data that is sent on the wire between cluster nodes is sensitive, and the network is untrusted, the recommendation is to use IPsec between the nodes. An alternative option is to decide exactly which configuration data is sensitive and then use the `tailf:aes-256-cfb-128-encrypted-string` type for that data. If the configuration data is of type `tailf:aes-256-cfb-128-encrypted-string` the encrypted data will be sent on the wire in update messages from the primary to the secondaries.

## API

There are two APIs used by the HA framework to control the replication aspects of NSO. First there exists a synchronous API used to tell NSO what to do, secondly the application may create a notifications socket and subscribe to HA related events where NSO notifies the application on certain HA related events such as the loss of the primary etc. The HA related notifications sent by NSO are crucial to how to program the HA framework.

The HA related classes reside in the `com.tailf.ha` package. See Javadocs for reference. The HA notifications related classes reside in the `com.tailf.notif` package, See Javadocs for reference.

## Ticks

The configuration parameter `/ncs-cfg/ha/tick-timeout` is by default set to 20 seconds. This means that every 20 seconds each secondary will send a tick message on the socket leading to the primary. Similarly, the primary will send a tick message every 20 seconds on every secondary socket.

This aliveness detection mechanism is necessary for NSO. If a socket gets closed all is well, NSO will cleanup and notify the application accordingly using the notifications API. However, if a remote node freezes, the socket will not get properly closed at the other end. NSO will distribute update data from the primary to the secondaries. If a remote node is not reading the data, TCP buffer will get full and NSO will have to start to buffer the data. NSO will buffer data for at most `tickTime` times 3 time units. If a

`tick` has not been received from a remote node within that time, the node will be considered dead. NSO will report accordingly over the notifications socket and either remove the hanging secondary or, if it is a secondary that loose contact with the primary, go into the initial `NONE` state.

If the HAFW can be really trusted, it is possible to set this timeout to `PT0S`, i.e zero, in which case the entire dead-node-detection mechanism in NSO is disabled.

## Relay secondaries

The normal setup of a NSO HA cluster is to have all secondaries connected directly to the primary. This is a configuration that is both conceptually simple and reasonably straightforward to manage for the HAFW. In some scenarios, in particular a cluster with multiple secondaries at a location that is network-wise distant from the primary, it can however be sub-optimal, since the replicated data will be sent to each remote secondary individually over a potentially low-bandwidth network connection.

To make this case more efficient, we can instruct a secondary to be a relay for other secondaries, by invoking the `Ha.beRelay()` method. This will make the secondary start listening on the IP address and port configured for HA in `ncs.conf`, and handle connections from other secondaries in the same manner as the cluster primary does. The initial CDB copy (if needed) to a new secondary will be done from the relay secondary, and when the relay secondary receives CDB data for replication from its primary, it will distribute the data to all its connected secondaries in addition to updating its own CDB copy.

To instruct a node to become a secondary connected to a relay secondary, we use the `Ha.beSecondary()` method as usual, but pass the node information for the relay secondary instead of the node information for the primary. I.e. the "sub-secondary" will in effect consider the relay secondary as its primary. To instruct a relay secondary to stop being a relay, we can invoke the `Ha.beSecondary()` method with the same parameters as in the original call. This is a no-op for a "normal" secondary, but it will cause a relay secondary to stop listening for secondary connections, and disconnect any already connected "sub-secondaries".

This setup requires special consideration by the HAFW. Instead of just telling each secondary to connect to the primary independently, it must setup the secondaries that are intended to be relays, and tell them to become relays, before telling the "sub-secondaries" to connect to the relay secondaries. Consider the case of a primary `M` and a secondary `S0` in one location, and two secondaries `S1` and `S2` in a remote location, where we want `S1` to act as relay for `S2`. The setup of the cluster then needs to follow this procedure:

- 1 Tell `M` to be primary.
- 2 Tell `S0` and `S1` to be secondary with `M` as primary.
- 3 Tell `S1` to be relay.
- 4 Tell `S2` to be secondary with `S1` as primary.

Conversely, the handling of network outages and node failures must also take the relay secondary setup into account. For example, if a relay secondary loses contact with its primary, it will transition to the `NONE` state just like any other secondary, and it will then disconnect its "sub-secondaries" which will cause those to transition to `NONE` too, since they lost contact with "their" primary. Or if a relay secondary dies in a way that is detected by its "sub-secondaries", they will also transition to `NONE`. Thus in the example above, `S1` and `S2` needs to be handled differently. E.g. if `S2` dies, the HAFW probably won't take any action, but if `S1` dies, it makes sense to instruct `S2` to be a secondary of `M` instead (and when `S1` comes back, perhaps tell `S2` to be a relay and `S1` to be a secondary of `S2`).

Besides the use of `Ha.beRelay()`, the API is mostly unchanged when using relay secondaries. The HA event notifications reporting the arrival or the death of a secondary are still generated only by the "real" cluster primary. If the `Ha.HaStatus()` method is used towards a relay secondary, it will report the

node state as `SECONDARY_RELAY` rather than just `SECONDARY`, and the array of nodes will have its primary as the first element (same as for a "normal" secondary), followed by its "sub-secondaries" (if any).

## CDB replication

When HA is enabled in `ncs.conf`, CDB automatically replicates data written on the primary to the connected secondary nodes. Replication is done on a per-transaction basis to all the secondaries in parallel and is synchronous. When NSO is in secondary mode the northbound APIs are in read-only mode, that is the configuration can not be changed on a secondary other than through replication updates from the primary. It is still possible to read from for example `NETCONF` or `CLI` (if they are enabled) on a secondary. CDB subscriptions works as usual. When NSO is in the `NONE` state CDB is unlocked and it behaves as when NSO is not in HA mode at all.

Unlike configuration data, operational data is replicated only if it is defined as persistent in the data model (using the `tailf:persistent` extension).





## CHAPTER 8

# Rollbacks

---

- [Introduction, page 109](#)
- [Configuration, page 109](#)

## Introduction

NSO support creating rollback files during the commit of a transaction that allows for rolling back the introduced changes. Rollbacks does not come without a cost and should be disabled if the functionality is not going to be used. Enabling rollbacks impact both the time it takes to commit a change and requires sufficient storage on disk.

Rollback files contain a set of headers and the data required to restore the changes that were made when the rollback was created. One of the header fields includes a unique rollback id that can be used to address the rollback file independent of the rollback numbering format.

Use of rollbacks from the supported APIs and the CLI is documented in the documentation for the given API.

## Configuration

NSO is configured through a configuration file - `ncs.conf`. In that file we have the following items related to rollbacks:

<code>/ncs-config/rollback/ enabled</code>	If 'true', then a rollback file will be created whenever the running configuration is modified.
<code>/ncs-config/rollback/ directory</code>	Location where rollback files will be created.
<code>/ncs-config/rollback/ history-size</code>	Number of old rollback files to save.







## CHAPTER 9

# The AAA infrastructure

---

- [The problem, page 111](#)
- [Structure - data models, page 111](#)
- [AAA related items in ncs.conf, page 112](#)
- [Authentication, page 114](#)
- [Restricting the IPC port, page 127](#)
- [Group Membership, page 127](#)
- [Authorization, page 128](#)
- [The AAA cache, page 143](#)
- [Populating AAA using CDB, page 143](#)
- [Hiding the AAA tree, page 144](#)

## The problem

This chapter describes how to use NSO's built-in authentication and authorization mechanisms. Users log into NSO through the CLI, NETCONF, RESTCONF, SNMP, or via the Web UI. In either case, users need to be *authenticated*. That is, a user needs to present credentials, such as a password or a public key in order to gain access. As an alternative for RESTCONF, users can be authenticated via token validation.

Once a user is authenticated, all operations performed by that user need to be *authorized*. That is, certain users may be allowed to perform certain tasks, whereas others are not. This is called *authorization*. We differentiate between authorization of commands and authorization of data access.

## Structure - data models

The NSO daemon manages device configuration including AAA information. In fact, NSO both manages AAA information and uses it. The AAA information describes which users may login, what passwords they have and what they are allowed to do.

This is solved in NSO by requiring a data model to be both loaded and populated with data. NSO uses the YANG module `tailf-aaa.yang` for authentication, while `ietf-netconf-acm.yang` (NACM, [RFC 8341](#)) as augmented by `tailf-acm.yang` is used for group assignment and authorization.

## Data model contents

The NACM data model is targeted specifically towards access control for NETCONF operations, and thus lacks some functionality that is needed in NSO, in particular support for authorization of CLI commands and the possibility to specify the "context" (NETCONF/CLI/etc) that a given authorization rule should apply to. This functionality is modeled by augmentation of the NACM model, as defined in the `tailf-acm.yang` YANG module.

The `ietf-netconf-acm.yang` and `tailf-acm.yang` modules can be found in `$NCS_DIR/src/ncs/yang` directory in the release, while `tailf-aaa.yang` can be found in the `$NCS_DIR/src/ncs/aaa` directory.

NACM options related to services are modeled by augmentation of the NACM model, as defined in the `tailf-ncs-acm.yang` YANG module. The `tailf-ncs-acm.yang` can be found in `$NCS_DIR/src/ncs/yang` directory in the release.

The complete AAA data model defines a set of users, a set of groups and a set of rules. The data model must be populated with data that is subsequently used by NSO itself when it authenticates users and authorizes user data access. These YANG modules work exactly like all other fxs files loaded into the system with the exception that NSO itself uses them. The data belongs to the application, but NSO itself is the user of the data.

Since NSO requires a data model for the AAA information for its operation, it will report an error and fail to start if these data models can not be found.

## AAA related items in ncs.conf

NSO itself is configured through a configuration file - `ncs.conf`. In that file we have the following items related to authentication and authorization:

`/ncs-config/aaa/ssh-server-key-dir`

If SSH termination is enabled for NETCONF or the CLI, the NSO built-in SSH server needs to have server keys. These keys are generated by the NSO install script and by default end up in `$NCS_DIR/etc/ncs/ssh`.

It is also possible to use OpenSSH to terminate NETCONF or the CLI. If OpenSSH is used to terminate SSH traffic, this setting has no effect.

`/ncs-config/aaa/ssh-pubkey-authentication`

If SSH termination is enabled for NETCONF or the CLI, this item controls how the NSO SSH daemon locates the user keys for public key authentication. See [the section called "Public Key Login"](#) for the details.

`/ncs-config/aaa/local-authentication/enabled`

The term "local user" refers to a user stored under `/aaa/authentication/users`. The alternative is a user unknown to NSO, typically authenticated by PAM.

By default, NSO first checks local users before trying PAM or external authentication.

Local authentication is practical in test environments. It is also useful when we want to have one set of users that are allowed to login to the host with normal shell access and another set of users that are only allowed to access the system using the normal encrypted, fully authenticated, northbound interfaces of NSO.

/ncs-config/aaa/pam	<p>If we always authenticate users through PAM, it may make sense to set this configurable to <code>false</code>. If we disable local authentication, it implicitly means that we must use either PAM authentication or "external authentication". It also means that we can leave the entire data trees under <code>/aaa/authentication/users</code> and, in the case of "external auth" also <code>/nacm/groups</code> (for NACM) or <code>/aaa/authentication/groups</code> (for legacy tailf-aaa) empty.</p> <p>NSO can authenticate users using PAM (Pluggable Authentication Modules). PAM is an integral part of most Unix-like systems.</p> <p>PAM is a complicated - albeit powerful - subsystem. It may be easier to have all users stored locally on the host. However, if we want to store users in a central location, PAM can be used to access the remote information. PAM can be configured to perform most login scenarios including RADIUS and LDAP. One major drawback with PAM authentication is that there is no easy way to extract the group information from PAM. PAM authenticates users; it does not assign a user to a set of groups.</p>
/ncs-config/aaa/default-group	<p>PAM authentication is thoroughly described later in this chapter.</p> <p>If this configuration parameter is defined and if the group of a user cannot be determined, a logged in user ends up in the given default group.</p>
/ncs-config/aaa/external-authentication	<p>NSO can authenticate users using an external executable. This is further described in the <a href="#">the section called "External authentication"</a> section.</p>
/ncs-config/aaa/external-validation	<p>As an alternative, you may consider using package authentication.</p> <p>NSO can authenticate users by validation of tokens using an external executable. This is further described in the <a href="#">the section called "External token validation"</a> section. Where external authentication uses a username and password to authenticate a user, external validation uses a token. The validation script should use the token to authenticate a user and can, optionally, also return a new token to be returned with the result of the request. It is currently only supported for RESTCONF.</p>
/ncs-config/aaa/external-challenge	<p>NSO has support for multifactor authentication by sending challenges to a user. Challenges may be sent from any of the external authentication mechanisms but is currently only supported by JSONRPC and CLI over SSH. This is further described in the <a href="#">the section called "External multi factor authentication"</a> section.</p>
/ncs-config/aaa/package-authentication	<p>NSO can authenticate users using package authentication. It extends the concept of external authentication by allowing multiple packages to be used for authentication instead of a single executable. This is further described in the <a href="#">the section called "Package Authentication"</a> section.</p>
/ncs-config/aaa/single-sign-on	<p>With this setting enabled, NSO invokes Package Authentication on all requests to HTTP endpoints with the <code>/sso</code> prefix. This way, Package Authentication packages that require custom endpoints can expose them under the <code>/sso</code> base route.</p>

```
/ncs-config/aaa/
single-sign-on/enable-
automatic-redirect
```

For example, a SAMLv2 Single Sign-On (SSO) package needs to process requests to an AssertionConsumerService endpoint, such as `/sso/saml/acs`, and therefore requires enabling this setting.

This is a valid authentication method for webui and JSON-RPC interfaces and needs Package Authentication to be enabled as well.

If only one Single Sign-On package is configured (a package with `single-sign-on-url` set in `package-meta-data.xml`) and also this setting is enabled, NSO automatically redirects all unauthenticated access attempts to the configured `single-sign-on-url`.

## Authentication

Depending on northbound management protocol, when a user session is created in NSO, it may or may not be authenticated. If the session is not yet authenticated, NSO's AAA subsystem is used to perform authentication and authorization, as described below. If the session already has been authenticated, NSO's AAA assigns groups to the user as described in [the section called “Group Membership”](#), and performs authorization, as described in [the section called “Authorization”](#).

The authentication part of the data model can be found in `tailf-aaa.yang`:

```
container authentication {
 tailf:info "User management";
 container users {
 tailf:info "List of local users";
 list user {
 key name;
 leaf name {
 type string;
 tailf:info "Login name of the user";
 }
 leaf uid {
 type int32;
 mandatory true;
 tailf:info "User Identifier";
 }
 leaf gid {
 type int32;
 mandatory true;
 tailf:info "Group Identifier";
 }
 leaf password {
 type passwdStr;
 mandatory true;
 }
 leaf ssh_keydir {
 type string;
 mandatory true;
 tailf:info "Absolute path to directory where user's ssh keys
 may be found";
 }
 leaf homedir {
 type string;
 mandatory true;
 tailf:info "Absolute path to user's home directory";
 }
 }
 }
}
```

AAA authentication is used in the following cases:

- When the built-in SSH server is used for NETCONF and CLI sessions.
- For Web UI sessions and REST access.
- When the method `Maapi.Authenticate()` is used.

NSO's AAA authentication is *not* used in the following cases:

- When NETCONF uses an external SSH daemon, such as OpenSSH.  
In this case, the NETCONF session is initiated using the program **netconf-subsys**, as described in the section called “NETCONF Transport Protocols” in *Northbound APIs*.
- When NETCONF uses TCP, as described in the section called “NETCONF Transport Protocols” in *Northbound APIs*, e.g. through the command **netconf-console**.
- When the CLI uses an external SSH daemon, such as OpenSSH, or a telnet daemon.  
In this case, the CLI session is initiated through the command **ncs\_cli**. An important special case here is when a user has logged in to the host and invokes the command **ncs\_cli** from the shell. In NSO deployments, it is crucial to consider this case. If non trusted users have shell access to the host, the `NCS_IPC_ACCESS_FILE` feature as described in [the section called “Restricting access to the IPC port”](#) must be used.
- When SNMP is used. SNMP has its own authentication mechanisms. See Chapter 4, *The NSO SNMP Agent* in *Northbound APIs*.
- When the method `Maapi.startUserSession()` is used without a preceding call of `Maapi.authenticate()`.

## Public Key Login

When a user logs in over NETCONF or the CLI using the built-in SSH server, with public key login, the procedure is as follows.

The user presents a username in accordance with the SSH protocol. The SSH server consults the settings for `/ncs-config/aaa/ssh-pubkey-authentication` and `/ncs-config/aaa/local-authentication/enabled`.

- 1 If `ssh-pubkey-authentication` is set to `local`, and the SSH keys in `/aaa/authentication/users/user{$USER}/ssh_keydir` match the keys presented by the user, authentication succeeds.
- 2 Otherwise, if `ssh-pubkey-authentication` is set to `system`, `local-authentication` is enabled, and the SSH keys in `/aaa/authentication/users/user{$USER}/ssh_keydir` match the keys presented by the user, authentication succeeds.
- 3 Otherwise, if `ssh-pubkey-authentication` is set to `system` and the user `/aaa/authentication/users/user{$USER}` does not exist, but the user does exist in the OS password database, the keys in the user's `$HOME/.ssh` directory are checked. If these keys match the keys presented by the user, authentication succeeds.
- 4 Otherwise, authentication fails.

In all cases the keys are expected to be stored in a file called `authorized_keys` (or `authorized_keys2` if `authorized_keys` does not exist), and in the native OpenSSH format (i.e. as generated by the OpenSSH **ssh-keygen** command). If authentication succeeds, the user's group membership is established as described in [the section called “Group Membership”](#).

This is exactly the same procedure that is used by the OpenSSH server with the exception that the built-in SSH server also may locate the directory containing the public keys for a specific user by consulting the `/aaa/authentication/users` tree.

## Setting up Public Key Login

We need to provide a directory where SSH keys are kept for a specific user, and give the absolute path to this directory for the `/aaa/authentication/users/user/ssh_keydir` leaf. If public key login is not desired at all for a user, the value of the `ssh_keydir` leaf should be set to "", i.e. the empty string. Similarly, if the directory does not contain any SSH keys, public key logins for that user will be disabled.

The built-in SSH daemon supports DSA, RSA and ED25519 keys. To generate and enable RSA keys of size 4096 bits for, say, user "bob", the following steps are required.

On the client machine, as user "bob", generate a private/public key pair as:

```
ssh-keygen -b 4096 -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/bob/.ssh/id_rsa):
Created directory '/home/bob/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/bob/.ssh/id_rsa.
Your public key has been saved in /home/bob/.ssh/id_rsa.pub.
The key fingerprint is:
ce:1b:63:0a:f9:d4:1d:04:7a:1d:98:0c:99:66:57:65 bob@buzz
ls -lt ~/.ssh
total 8
-rw----- 1 bob users 3247 Apr 4 12:28 id_rsa
-rw-r--r-- 1 bob users 738 Apr 4 12:28 id_rsa.pub
```

Now we need to copy the public key to the target machine where the NETCONF or CLI SSH client runs.

Assume we have the following user entry:

```
<user>
 <name>bob</name>
 <uid>100</uid>
 <gid>10</gid>
 <password>1feedbabe$nGlMYlZpQ0bzenyFOQI3Ll</password>
 <ssh_keydir>/var/system/users/bob/.ssh</ssh_keydir>
 <homedir>/var/system/users/bob</homedir>
</user>
```

We need to copy the newly generated file `id_rsa.pub`, which is the public key, to a file on the target machine called `/var/system/users/bob/.ssh/authorized_keys`



### Note

Since the release of [OpenSSH 7.0](#) support of `ssh-dss` host and user keys is disabled by default. If you want to continue using these, you may re-enable it using the following options for OpenSSH client:

```
HostKeyAlgorithms+=ssh-dss
PubkeyAcceptedKeyTypes+=ssh-dss
```

You may find full instructions at [OpenSSH Legacy Options](#) webpage.

## Password Login

Password login is triggered in the following cases:

- When a user logs in over NETCONF or the CLI using the built in SSH server, with a password. The user presents a username and a password in accordance with the SSH protocol.

- When a user logs in using the Web UI. The Web UI asks for a username and password.
- When the method `Maapi.authenticate()` is used.

In this case, NSO will by default try local authentication, PAM, external authentication and package authentication in that order, as described below. It is possible to change the order in which these are tried, by modifying the `ncs.conf` parameter `/ncs-config/aaa/auth-order`. See `ncs.conf(5)` in *Manual Pages* for details.

- 1 If `/aaa/authentication/users/user{$USER}` exists and the presented password matches the encrypted password in `/aaa/authentication/users/user{$USER}/password` the user is authenticated.
- 2 If the password does not match or if the user does not exist in `/aaa/authentication/users`, PAM login is attempted, if enabled. See [the section called “PAM”](#) for details.
- 3 If all of the above fails and external authentication is enabled, the configured executable is invoked. See [the section called “External authentication”](#) for details.

If authentication succeeds, the user's group membership is established as described in [the section called “Group Membership”](#).

## PAM

On operating systems supporting PAM, NSO also supports PAM authentication. Using PAM authentication with NSO can be very convenient since it allows us to have the same set of users and groups having access to NSO as those that have access to the UNIX/Linux host itself.

If we use PAM, we do not have to have any users or any groups configured in the NSO `aaa` namespace at all. To configure PAM we typically need to do the following:

- 1 Remove all users and groups from the `aaa` initialization XML file.
- 2 Enable PAM in `ncs.conf` by adding:

```
<pam>
 <enabled>true</enabled>
 <service>common-auth</service>
</pam>
```

to the `aaa` section in `ncs.conf`. The `service` name specifies the PAM service, typically a file in the directory `/etc/pam.d`, but may alternatively be an entry in a file `/etc/pam.conf`, depending on OS and version. Thus it is possible to have a different login procedure to NSO than to the host itself.

- 3 If `pam` is enabled and we want to use `pam` for login the system may have to run as root. This depends on how `pam` is configured locally. However the default "system-auth" will typically require root since the `pam` libraries then read `/etc/shadow`. If we don't want to run NSO as root, the solution here is to change owner of a helper program called `$NCS_DIR/lib/ncs/lib/core/pam/priv/epam` and also set the `setuid` bit.

```
cd $NCS_DIR/lib/ncs/lib/core/pam/priv/
chown root:root epam
chmod u+s epam
```

PAM is the recommended way to authenticate NSO users.

As an example, say that we have user `test` in `/etc/passwd`, and furthermore:

```
grep test /etc/group
operator:x:37:test
admin:x:1001:test
```

thus, the test user is part of the admin and the operator groups and logging in to NSO as the test user, through CLI ssh, Web UI, or netconf renders the following in the audit log.

```
<INFO> 28-Jan-2009::16:05:55.663 buzz ncs[14658]: audit user: test/0 logged
 in over ssh from 127.0.0.1 with authmeth:password
<INFO> 28-Jan-2009::16:05:55.670 buzz ncs[14658]: audit user: test/5 assigned
 to groups: operator,admin
<INFO> 28-Jan-2009::16:05:57.655 buzz ncs[14658]: audit user: test/5 CLI 'exit'
```

Thus, the test user was found and authenticated from `/etc/passwd`, and the crucial group assignment of the test user was done from `/etc/group`.

If we wish to be able to also manipulate the users, their passwords etc on the device we can write a private YANG model for that data, store that data in CDB, setup a normal CDB subscriber for that data, and finally when our private user data is manipulated, our CDB subscriber picks up the changes and changes the contents of the relevant `/etc` files.

## External authentication

A common situation is when we wish to have all authentication data stored remotely, not locally, for example on a remote RADIUS or LDAP server. This remote authentication server typically not only stores the users and their passwords, but also the group information.

If we wish to have not only the users, but also the group information stored on a remote server, the best option for NSO authentication is to use "external authentication".

If this feature is configured, NSO will invoke the executable configured in `/ncs-config/aaa/external-authentication/executable` in `ncs.conf`, and pass the username and the clear text password on `stdin` using the string notation: `"[user;password;]\n"`.

For example if user "bob" attempts to login over SSH using the password "secret", and external authentication is enabled, NSO will invoke the configured executable and write `"[bob;secret;]\n"` on the `stdin` stream for the executable.

The task of the executable is then to authenticate the user and also establish the username-to-groups mapping.

For example the executable could be a RADIUS client which utilizes some proprietary vendor attributes to retrieve the groups of the user from the RADIUS server. If authentication is successful, the program should write "accept " followed by a space-separated list of groups the user is member of, and additional information as described below. Again, assuming that Bob's password indeed was "secret", and that Bob is member of the "admin" and the "lamers" groups, the program should write "accept admin lamers \$uid \$gid \$supplementary\_gids \$HOME\n" on its standard output and then exit.



### Note

There is a general limit of 16000 bytes of output from the "externalauth" program

Thus the format of the output from an "externalauth" program when authentication is successful should be:

```
"accept $groups $uid $gid $supplementary_gids $HOME\n"
```

Where

- `$groups` is a space separated list of the group names the user is a member of.
- `$uid` is the UNIX integer user id NSO should use as default when executing commands for this user.
- `$gid` is the UNIX integer group id NSO should use as default when executing commands for this user.



- `$supplementary_gids` is a (possibly empty) space separated list of additional UNIX group ids the user is also a member of.
- `$HOME` is the directory which should be used as HOME for this user when NSO executes commands on behalf of this user.

It is further possible for the program to return a token on successful authentication, by using "accept\_token" instead of "accept":

```
"accept_token $groups $uid $gid $supplementary_gids $HOME $token\n"
```

Where `$token` is an arbitrary string. NSO will then, for some northbound interfaces, include this token in responses.

It is also possible for the program to return additional information on successful authentication, by using "accept\_info" instead of "accept":

```
"accept_info $groups $uid $gid $supplementary_gids $HOME $info\n"
```

Where `$info` is some arbitrary text. NSO will then just append this text to the generated audit log message (CONFD\_EXT\_LOGIN).

Yet another possibility is for the program to return a warning that the user's password is about to expire, by using "accept\_warning" instead of "accept":

```
"accept_warning $groups $uid $gid $supplementary_gids $HOME $warning\n"
```

Where `$warning` is an appropriate warning message. The message will be processed by NSO according to the setting of `/ncs-config/aaa/expiration-warning` in `ncs.conf`.

There is also support for token variations of "accept\_info" and "accept\_warning" namely "accept\_token\_info" and "accept\_token\_warning". Both "accept\_token\_info" and "accept\_token\_warning" expects the external program to output exactly the same as described above with the addition of a token after `$HOME`:

```
"accept_token_info $groups $uid $gid $supplementary_gids $HOME $token $info\n"
```

```
"accept_token_warning $groups $uid $gid $supplementary_gids $HOME $token $warning\n"
```

If authentication failed, the program should write "reject" or "abort", possibly followed by a reason for the rejection, and a trailing newline. For example "reject Bad password\n" or just "abort\n". The difference between "reject" and "abort" is that with "reject", NSO will try subsequent mechanisms configured for `/ncs-config/aaa/auth-order` in `ncs.conf` (if any), while with "abort", the authentication fails immediately. Thus "abort" can prevent subsequent mechanisms from being tried, but when external authentication is the last mechanism (as in the default order), it has the same effect as "reject".

Supported by some northbound APIs, such as JSONRPC and CLI over SSH, the external authentication may also choose to issue a challenge:

```
"challenge $challenge-id $challenge-prompt\n"
```



#### Note

The challenge-prompt may be multi line, why it must be base64 encoded

For more information on multi factor authentication, see the

the section called “External multi factor authentication” section.

When external authentication is used, the group list returned by the external program is prepended by any possible group information stored locally under the `/aaa` tree. Hence when we use external authentication it is indeed possible to have the entire `/aaa/authentication` tree empty. The group assignment performed by the external program will still be valid and the relevant groups will be used by NSO when the authorization rules are checked.

## External token validation

When username, password authentication is not feasible, authentication by token validation is possible. Currently only RESTCONF supports this mode of authentication. It shares all properties of external authentication, but instead of a username and password, it takes a token as input. The output is also almost the same, the only difference is that it is also expected to output a username.

If this feature is configured, NSO will invoke the executable configured in `/ncs-config/aaa/external-validation/executable` in `ncs.conf`, and pass the token on `stdin` using the string notation: `"[token;]\n"`.

For example if user "bob" attempts to login over RESTCONF using the token "topsecret", and external validation is enabled, NSO will invoke the configured executable and write `"[topsecret;]\n"` on the `stdin` stream for the executable.

The task of the executable is then to validate the token, thereby authenticating the user and also establish the username and username-to-groups mapping.

For example the executable could be a FUSION client which utilizes some proprietary vendor attributes to retrieve the username and groups of the user from the FUSION server. If token validation is successful, the program should write `"accept "` followed by a space-separated list of groups the user is member of, and additional information as described below. Again, assuming that Bob's token indeed was "topsecret", and that Bob is member of the "admin" and the "lamers" groups, the program should write `"accept admin lamers $uid $gid $supplementary_gids $HOME $USER\n"` on its standard output and then exit.



### Note

There is a general limit of 16000 bytes of output from the "externalvalidation" program

Thus the format of the output from an "externalvalidation" program when token validation authentication is successful should be:

```
"accept $groups $uid $gid $supplementary_gids $HOME $USER\n"
```

Where

- `$groups` is a space separated list of the group names the user is a member of.
- `$uid` is the UNIX integer user id NSO should use as default when executing commands for this user.
- `$gid` is the UNIX integer group id NSO should use as default when executing commands for this user.
- `$supplementary_gids` is a (possibly empty) space separated list of additional UNIX group ids the user is also a member of.
- `$HOME` is the directory which should be used as HOME for this user when NSO executes commands on behalf of this user.
- `$USER` is the user derived from mapping the token.

It is further possible for the program to return a new token on successful token validation authentication, by using "accept\_token" instead of "accept":

```
"accept_token $groups $uid $gid $supplementary_gids $HOME $USER $token
\n"
```

Where \$token is an arbitrary string. NSO will then, for some northbound interfaces, include this token in responses.

It is also possible for the program to return additional information on successful token validation authentication, by using "accept\_info" instead of "accept":

```
"accept_info $groups $uid $gid $supplementary_gids $HOME $USER $info\n"
```

Where \$info is some arbitrary text. NSO will then just append this text to the generated audit log message (CONFD\_EXT\_LOGIN).

Yet another possibility is for the program to return a warning that the user's password is about to expire, by using "accept\_warning" instead of "accept":

```
"accept_warning $groups $uid $gid $supplementary_gids $HOME $USER
$warning\n"
```

Where \$warning is an appropriate warning message. The message will be processed by NSO according to the setting of /ncs-config/aaa/expiration-warning in ncs.conf.

There is also support for token variations of "accept\_info" and "accept\_warning" namely "accept\_token\_info" and "accept\_token\_warning". Both "accept\_token\_info" and "accept\_token\_warning" expects the external program to output exactly the same as described above with the addition of a token after \$USER:

```
"accept_token_info $groups $uid $gid $supplementary_gids $HOME $USER
$token $info\n"
```

```
"accept_token_warning $groups $uid $gid $supplementary_gids $HOME $USER
$token $warning\n"
```

If token validation authentication failed, the program should write "reject" or "abort", possibly followed by a reason for the rejection, and a trailing newline. For example "reject Bad password\n" or just "abort\n". The difference between "reject" and "abort" is that with "reject", NSO will try subsequent mechanisms configured for /ncs-config/aaa/validation-order in ncs.conf (if any), while with "abort", the token validation authentication fails immediately. Thus "abort" can prevent subsequent mechanisms from being tried. Currently the only available token validation authentication mechanism is the external one.

Supported by some northbound APIs, such as JSONRPC and CLI over SSH, the external validation may also choose to issue a challenge:

```
"challenge $challenge-id $challenge-prompt\n"
```



#### Note

The challenge-prompt may be multi line, why it must be base64 encoded

For more information on multi factor authentication, see the [the section called “External multi factor authentication”](#) section.

## External multi factor authentication

When username, password or token authentication is not enough, a challenge may be sent from any of the external authentication mechanisms to the user. A challenge consists of a challenge id and a base64 encoded challenge prompt, and a user is supposed to send a response to the challenge. Currently only JSONRPC and CLI over SSH supports multi factor authentication. Responses to challenges of multi factor authentication has the same output as the token authentication mechanism.

If this feature is configured, NSO will invoke the executable configured in `/ncs-config/aaa/external-challenge/executable` in `ncs.conf`, and pass the challenge id and response on `stdin` using the string notation: `"[challenge-id;response;]\n"`.

For example a user "bob" has received a challenge from external authentication, external validation or external challenge and then attempts to login over JSONRPC with a response to the challenge using challenge id:"22efa",response:"ae457b". The external challenge mechanism is enabled, NSO will invoke the configured executable and write `"[22efa;ae457b;]\n"` on the `stdin` stream for the executable.

The task of the executable is then to validate the challenge id, response combination, thereby authenticating the user and also establish the username and username-to-groups mapping.

For example the executable could be a RADIUS client which utilizes some proprietary vendor attributes to retrieve the username and groups of the user from the RADIUS server. If challenge id, response validation is successful, the program should write `"accept "` followed by a space-separated list of groups the user is member of, and additional information as described below. Again, assuming that Bob's challenge id, response combination indeed was "22efa", "ae457b" and that Bob is member of the "admin" and the "lamers" groups, the program should write `"accept admin lamers $uid $gid $supplementary_gids $HOME $USER\n"` on its standard output and then exit.



### Note

There is a general limit of 16000 bytes of output from the "externalchallenge" program

Thus the format of the output from an "externalchallenge" program when challenge based authentication is successful should be:

```
"accept $groups $uid $gid $supplementary_gids $HOME $USER\n"
```

Where

- `$groups` is a space separated list of the group names the user is a member of.
- `$uid` is the UNIX integer user id NSO should use as default when executing commands for this user.
- `$gid` is the UNIX integer group id NSO should use as default when executing commands for this user.
- `$supplementary_gids` is a (possibly empty) space separated list of additional UNIX group ids the user is also a member of.
- `$HOME` is the directory which should be used as HOME for this user when NSO executes commands on behalf of this user.
- `$USER` is the user derived from mapping the challenge id, response.

It is further possible for the program to return a token on successful authentication, by using `"accept_token"` instead of `"accept"`:

```
"accept_token $groups $uid $gid $supplementary_gids $HOME $USER $token\n"
```

Where `$token` is an arbitrary string. NSO will then, for some northbound interfaces, include this token in responses.

It is also possible for the program to return additional information on successful authentication, by using `"accept_info"` instead of `"accept"`:

```
"accept_info $groups $uid $gid $supplementary_gids $HOME $USER $info\n"
```

Where `$info` is some arbitrary text. NSO will then just append this text to the generated audit log message (`CONFD_EXT_LOGIN`).

Yet another possibility is for the program to return a warning that the user's password is about to expire, by using `"accept_warning"` instead of `"accept"`:

```
"accept_warning $groups $uid $gid $supplementary_gids $HOME $USER $warning\n"
```

Where `$warning` is an appropriate warning message. The message will be processed by NSO according to the setting of `/ncs-config/aaa/expiration-warning` in `ncs.conf`.

There is also support for token variations of `"accept_info"` and `"accept_warning"` namely `"accept_token_info"` and `"accept_token_warning"`. Both `"accept_token_info"` and `"accept_token_warning"` expects the external program to output exactly the same as described above with the addition of a token after `$USER`:

```
"accept_token_info $groups $uid $gid $supplementary_gids $HOME $USER $token $info\n"
```

```
"accept_token_warning $groups $uid $gid $supplementary_gids $HOME $USER $token $warning\n"
```

If authentication failed, the program should write `"reject"` or `"abort"`, possibly followed by a reason for the rejection, and a trailing newline. For example `"reject Bad challenge response\n"` or just `"abort\n"`. The difference between `"reject"` and `"abort"` is that with `"reject"`, NSO will try subsequent mechanisms configured for `/ncs-config/aaa/challenge-order` in `ncs.conf` (if any), while with `"abort"`, the challenge response authentication fails immediately. Thus `"abort"` can prevent subsequent mechanisms from being tried. Currently the only available challenge response authentication mechanism is the external one.

Supported by some northbound APIs, such as JSONRPC and CLI over SSH, the external challenge may also choose to issue a new challenge:

```
"challenge $challenge-id $challenge-prompt\n"
```




---

**Note** The challenge-prompt may be multi line, so it must be base64 encoded

---




---

**Note** Note that when using challenges with the CLI over SSH, the `/ncs-config/cli/ssh/use-keyboard-interactive>` needs to be set to true for the challenges to be sent correctly to the client!

---




---

**Note** The configuration of the ssh client used may need to be given the option to allow a higher number of allowed number of password prompts, e.g. `-o NumberOfPasswordPrompts`, else the default number may introduce an unexpected behaviour when the client is presented with multiple challenges.

---

## Package Authentication

The Package Authentication functionality allows for packages to handle the NSO authentication in a customized fashion. Authentication data can e.g. be stored remotely, and a script in the package is used to communicate with the remote system.

Compared to external authentication, the Package Authentication mechanism allows specifying multiple packages to be invoked in the order they appear in the configuration. NSO provides implementations for LDAP, SAMLv2, and TACACS+ protocols with packages available in `$NCS_DIR/packages/auth/`. Additionally, you can implement your own authentication packages as detailed below.

Authentication packages are NSO packages with the required content of an executable file `scripts/authenticate`. This executable basically follows the same API, and limitations, as the external auth script, but with a different input format and some additional functionality. Other than these requirements, it is possible to customize the package arbitrarily.




---

**Note** Package authentication is supported for Single Sign-On (see Chapter 4, *Single Sign-On in Web UI*), JSONRPC, and RESTCONF. Note that Single Sign-On and (non batch) JSON-RPC allows all functionality while the RESTCONF interface will treat anything other than a `"accept_username"` reply from the package as if authentication failed!

---

Package authentication is enabled by setting the `ncs.conf` options `/ncs-config/aaa/package-authentication/enabled` to true, and adding the package by name in the `/ncs-config/aaa/package-authentication/packages` list. The order of the configured packages is the order that the packages will be used when attempting to authenticate a user. See `ncs.conf(5)` in *Manual Pages* for details.

If this feature is configured in `ncs.conf`, NSO will for each configured package invoke `script/authenticate`, and pass username, password, original HTTP request (i.e. the user supplied "next" query parameter) HTTP request, HTTP headers, HTTP body, client source IP, client source port, northbound API context, and protocol on `stdin` using the string notation:

```
"[user;password;orig_request;request;headers;body;src-ip;src-port;ctx;proto;]\n"
```



**Note** The fields user, password, orig\_request, request, headers, body are all base64 encoded.



**Note** If the body length exceeds the `partial_post_size` of the RESTCONF server, the body passed to the authenticate script will only contain the string `'==nso_package_authentication_partial_body=='`.



**Note** The original request will be prefixed with the string `'==nso_package_authentication_next=='` before base64 encoded part. This means supplying the "next" query parameter value `"/my-location"` will pass the following string to the authentication script: `'==nso_package_authentication_next==L215LWxvY2F0aW9u'`.

For example if an unauthenticated user attempts to start a single sign-on process over northbound HTTP based APIs with the `cisco-nso-saml2-auth` package, package authentication is enabled and configured with packages, and also single sign-on is enabled, NSO will, for each configured package, invoke the executable `scripts/authenticate` and write `"[ ; ; R0VUIC9zc28vc2FtbC9sb2dpci8gSFRUUC8xLjE= ; ; 127.0.0.1 ; 59226 ; webui ; https ; ]\n"` on the `stdin` stream for the executable.

For clarity, the base64 decoded contents sent to `stdin` presented: `"[ ; ; GET /sso/saml/login/ HTTP/1.1 ; ; 127.0.0.1 ; 54321 ; webui ; https ; ]\n"`.

The task of the package is then to authenticate the user and also establish the username-to-groups mapping.

For example the package could support a SAMLv2 authentication protocol which communicates with an Identity Provider (IdP) for authentication. If authentication is successful, the program should write either `"accept "`, or `"accept_username "`, depending on if the authentication is started with username or if an external entity handles the entire authentication and supplies the username for a successful authentication. (SAMLv2 uses `accept_username`, since the IdP handles the entire authentication.) The `"accept_username "` is followed by a username and then followed by a space-separated list of groups the user is member of, and additional information as described below. If authentication is successful and the authenticated user Bob is member of the groups "admin" and "wheel", the program should write `"accept_username bob admin wheel 1000 1000 100 /home/bob\n"` on its standard output and then exit.



**Note** There is a general limit of 16000 bytes of output from the "packageauth" program.

Thus the format of the output from a "packageauth" program when authentication is successful should be either the same as from "externalauth" (see [the section called "External authentication"](#)) or the following:

```
"accept_username $USER $groups $uid $gid $supplementary_gids $HOME\n"
```

Where

- `$USER` is the user derived during the execution of the "packageauth" program.
- `$groups` is a space separated list of the group names the user is a member of.
- `$uid` is the UNIX integer user id NSO should use as default when executing commands for this user.

- `$gid` is the UNIX integer group id NSO should use as default when executing commands for this user.
- `$supplementary_gids` is a (possibly empty) space separated list of additional UNIX group ids the user is also a member of.
- `$HOME` is the directory which should be used as `HOME` for this user when NSO executes commands on behalf of this user.

In addition to the "externalauth" API, the authentication packages can also return the following responses:

- `unknown 'reason'` - (*reason* being plain-text) if they can't handle authentication for the supplied input.
- `redirect 'url'` - (*url* being base64 encoded) for an HTTP redirect.
- `content 'content-type' 'content'` - (*content-type* being plain-text mime-type and *content* being base64 encoded) to relay supplied content.
- `accept_username_redirect url $USER $groups $uid $gid $supplementary_gids $HOME` - which combines the `accept_username` and `redirect`.

It is also possible for the program to return additional information on successful authentication, by using "accept\_info" instead of "accept":

```
"accept_info $groups $uid $gid $supplementary_gids $HOME $info\n"
```

Where `$info` is some arbitrary text. NSO will then just append this text to the generated audit log message (NCS\_PACKAGE\_AUTH\_SUCCESS).

Yet another possibility is for the program to return a warning that the user's password is about to expire, by using "accept\_warning" instead of "accept":

```
"accept_warning $groups $uid $gid $supplementary_gids $HOME $warning\n"
```

Where `$warning` is an appropriate warning message. The message will be processed by NSO according to the setting of `/ncs-config/aaa/expiration-warning` in `ncs.conf`.

If authentication fails, the program should write "reject" or "abort", possibly followed by a reason for the rejection, and a trailing newline. For example "reject 'Bad password'\n" or just "abort\n". The difference between "reject" and "abort" is that with "reject", NSO will try subsequent mechanisms configured for `/ncs-config/aaa/auth-order`, and packages configured for `/ncs-config/aaa/package-authentication/packages` in `ncs.conf` (if any), while with "abort", the authentication fails immediately. Thus "abort" can prevent subsequent mechanisms from being tried, but when external authentication is the last mechanism (as in the default order), it has the same effect as "reject".

When package authentication is used, the group list returned by the package executable is prepended by any possible group information stored locally under the `/aaa` tree. Hence when package authentication is used, it is indeed possible to have the entire `/aaa/authentication` tree empty. The group assignment performed by the external program will still be valid and the relevant groups will be used by NSO when the authorization rules are checked.

## Username/password Package Authentication for CLI

Package authentication will invoke the `scripts/authenticate` when a user tries to authenticate using CLI. In this case only the username, password, client source IP, client source port, northbound API context, and protocol will be passed to the script.



**Note**

When serving a username/password request, script output other than accept, challenge or abort will be treated as if authentication failed.

## Package Challenges

When this is enabled, `/ncs-config/aaa/package-authentication/package-challenge/enabled` is set to true, packages will also be used to try to resolve challenges sent to the server and is only supported by CLI over SSH. The script `script/challenge` will be invoked passing challenge id, response, client source IP, client source port, northbound API context, and protocol on `stdin` using the string notation: `"[challengeid;response;src-ip;src-port;ctx;proto;]\n"`. The output should follow that of the authenticate script.

**Note**

The fields `challengeid` and `response` are base64 encoded when passed to the script.

## Restricting the IPC port

NSO listens for client connections on the NSO IPC port. See `/ncs-config/ncs-ipc-address/ip` in `ncs.conf`. Access to this port is by default not authenticated. That means that all users with shell access to the host, can connect to this port. So NSO deployment ends up in either of two cases, untrusted users have, or have not shell access to the host(s) where NSO is deployed. If all shell users on the deployment host(s) are trusted, no further action is required, however if untrusted users do have shell access to the hosts, access to the IPC port must be restricted, see [the section called “Restricting access to the IPC port”](#).

If IPC port access is not used, an untrusted shell user can simply invoke:

```
bob> ncs_cli --user admin
```

to impersonate as the `admin` user, or invoke

```
bob> ncs_load > all.xml
```

to retrieve the entire configuration.

## Group Membership

Once a user is authenticated, group membership must be established. A single user can be a member of several groups. Group membership is used by the authorization rules to decide which operations a certain user is allowed to perform. Thus the NSO AAA authorization model is entirely group based. This is also sometimes referred to as role based authorization.

All groups are stored under `/nacm/groups`, and each group contains a number of usernames. The `ietf-netconf-acm.yang` model defines a group entry:

```
list group {
 key name;

 description
 "One NACM Group Entry. This list will only contain
 configured entries, not any entries learned from
 any transport protocols.";

 leaf name {
 type group-name-type;
```

```

 description
 "Group name associated with this entry.";
 }

 leaf-list user-name {
 type user-name-type;
 description
 "Each entry identifies the username of
 a member of the group associated with
 this entry.";
 }
}

```

The `tailf-acm.yang` model augments this with a `gid` leaf:

```

augment /nacm:nacm/nacm:groups/nacm:group {
 leaf gid {
 type int32;
 description
 "This leaf associates a numerical group ID with the group.
 When a OS command is executed on behalf of a user,
 supplementary group IDs are assigned based on 'gid' values
 for the groups that the use is a member of.";
 }
}

```

A valid group entry could thus look like:

```

<group>
 <name>admin</name>
 <user-name>bob</user-name>
 <user-name>joe</user-name>
 <gid xmlns="http://tail-f.com/yang/acm">99</gid>
</group>

```

The above XML data would then mean that users `bob` and `joe` are members of the `admin` group. The users need not necessarily exist as actual users under `/aaa/authentication/users` in order to belong to a group. If for example PAM authentication is used, it does not make sense to have all users listed under `/aaa/authentication/users`.

By default, the user is assigned to groups by using any groups provided by the northbound transport (e.g. via the `ncs_cli` or `netconf-subsys` programs), by consulting data under `/nacm/groups`, by consulting the `/etc/group` file, and by using any additional groups supplied by the authentication method. If `/nacm/enable-external-groups` is set to "false", only the data under `/nacm/groups` is consulted.

The resulting group assignment is the union of these methods, if it is non-empty. Otherwise, the default group is used, if configured ( `/ncs-config/aaa/default-group` in `ncs.conf`).

A user entry has a UNIX uid and UNIX gid assigned to it. Groups may have optional group ids. When a user is logged in, and NSO tries to execute commands on behalf of that user, the uid/gid for the command execution is taken from the user entry. Furthermore, UNIX supplementary group ids are assigned according to the gids in the groups where the user is a member.

## Authorization

Once a user is authenticated and group membership is established, when the user starts to perform various actions, each action must be authorized. Normally the authorization is done based on rules configured in the AAA data model as described in this section.

The authorization procedure first checks the value of `/nacm/enable-nacm`. This leaf has a default of `true`, but if it is set to `false`, all access is permitted. Otherwise, the next step is to traverse the `rule-list` list:

```
list rule-list {
 key "name";
 ordered-by user;
 description
 "An ordered collection of access control rules.";

 leaf name {
 type string {
 length "1..max";
 }
 description
 "Arbitrary name assigned to the rule-list.";
 }
 leaf-list group {
 type union {
 type matchall-string-type;
 type group-name-type;
 }
 description
 "List of administrative groups that will be
 assigned the associated access rights
 defined by the 'rule' list.

 The string '*' indicates that all groups apply to the
 entry.";
 }
}
// ...
}
```

If the group leaf-list in a `rule-list` entry matches any of the user's groups, the `cmdrule` list entries are examined for command authorization, while the rule entries are examined for rpc, notification, and data authorization.

## Command authorization

The `tailf-acm.yang` module augments the `rule-list` entry in `ietf-netconf-acm.yang` with a `cmdrule` list:

```
augment /nacm:nacm/nacm:rule-list {

 list cmdrule {
 key "name";
 ordered-by user;
 description
 "One command access control rule. Command rules control access
 to CLI commands and Web UI functions.

 Rules are processed in user-defined order until a match is
 found. A rule matches if 'context', 'command', and
 'access-operations' match the request. If a rule
 matches, the 'action' leaf determines if access is granted
 or not.";

 leaf name {
 type string {
 length "1..max";
 }
 }
 }
}
```

```

 }
 description
 "Arbitrary name assigned to the rule.";
}

leaf context {
 type union {
 type nacm:matchall-string-type;
 type string;
 }
 default "*";
 description
 "This leaf matches if it has the value '*' or if its value
 identifies the agent that is requesting access, i.e. 'cli'
 for CLI or 'webui' for Web UI.";
}

leaf command {
 type string;
 default "*";
 description
 "Space-separated tokens representing the command. Refer
 to the Tail-f AAA documentation for further details.";
}

leaf access-operations {
 type union {
 type nacm:matchall-string-type;
 type nacm:access-operations-type;
 }
 default "*";
 description
 "Access operations associated with this rule.

 This leaf matches if it has the value '*' or if the
 bit corresponding to the requested operation is set.";
}

leaf action {
 type nacm:action-type;
 mandatory true;
 description
 "The access control action associated with the
 rule. If a rule is determined to match a
 particular request, then this object is used
 to determine whether to permit or deny the
 request.";
}

leaf log-if-permit {
 type empty;
 description
 "If this leaf is present, access granted due to this rule
 is logged in the developer log. Otherwise, only denied
 access is logged. Mainly intended for debugging of rules.";
}

leaf comment {
 type string;
 description
 "A textual description of the access rule.";
}

```

```
 }
}
```

Each rule has seven leafs. The first is the name list key, the following three leafs are matching leafs. When NSO tries to run a command it tries to match the command towards the matching leafs and if all of context, command, and access-operations match, the fifth field, i.e. the action, is applied.

name	name is the name of the rule. The rules are checked in order, with the ordering given by the the YANG ordered-by user semantics, i.e. independent of the key values.
context	context is either of the strings cli, webui, or * for a command rule. This means that we can differentiate authorization rules for which access method is used. Thus if command access is attempted through the CLI the context will be the string cli whereas for operations via the Web UI, the context will be the string webui.
command	This is the actual command getting executed. If the rule applies to one or several CLI commands, the string is a space separated list of CLI command tokens, for example request system reboot. If the command applies to Web UI operations, it is a space separated string similar to a CLI string. A string which consists of just "*" matches any command.  In general, we do not recommend using command rules to protect the configuration. Use rules for data access as described in the next section to control access to different parts of the data. Command rules should be used only for CLI commands and Web UI operations that cannot be expressed as data rules.  The individual tokens can be POSIX extended regular expressions. Each regular expression is implicitly anchored, i.e. an "^" is prepended and a "\$" is appended to the regular expression.
access-operations	access-operations is used to match the operation that NSO tries to perform. It must be one or both of the "read" and "exec" values from the access-operations-type bits type definition in ietf-netconf-acm.yang, or "*" to match any operation.
action	If all of the previous fields match, the rule as a whole matches and the value of action will be taken. I.e. if a match is found, a decision is made whether to permit or deny the request in its entirety. If action is permit, the request is permitted, if action is deny, the request is denied and an entry written to the developer log.
log-if-permit	If this leaf is present, an entry is written to the developer log for a matching request also when action is permit. This is very useful when debugging command rules.
comment	An optional textual description of the rule.

For the rule processing to be written to the devel log, the /ncs-config/logs/developer-log-level entry in ncs.conf must be set to trace.

If no matching rule is found in any of the cmdrule lists in any rule-list entry that matches the user's groups, this augmentation from tailf-acm.yang is relevant:

```
augment /nacm:nacm {
 leaf cmd-read-default {
 type nacm:action-type;
 default "permit";
 description
```

```

 "Controls whether command read access is granted
 if no appropriate cmdrule is found for a
 particular command read request.";
 }

 leaf cmd-exec-default {
 type nacm:action-type;
 default "permit";
 description
 "Controls whether command exec access is granted
 if no appropriate cmdrule is found for a
 particular command exec request.";
 }

 leaf log-if-default-permit {
 type empty;
 description
 "If this leaf is present, access granted due to one of
 /nacm/read-default, /nacm/write-default, or /nacm/exec-default
 /nacm/cmd-read-default, or /nacm/cmd-exec-default
 being set to 'permit' is logged in the developer log.
 Otherwise, only denied access is logged. Mainly intended
 for debugging of rules.";
 }
}

```

- If "read" access is requested, the value of /nacm/cmd-read-default determines whether access is permitted or denied.
- If "exec" access is requested, the value of /nacm/cmd-exec-default determines whether access is permitted or denied.

If access is permitted due to one of these default leaves, the /nacm/log-if-default-permit has the same effect as the log-if-permit leaf for the cmdrule lists.

## Rpc, notification, and data authorization

The rules in the rule list are used to control access to rpc operations, notifications, and data nodes defined in YANG models. Access to invocation of actions (tailf:action) is controlled with the same method as access to data nodes, with a request for "exec" access. ietf-netconf-acm.yang defines a rule entry as:

```

list rule {
 key "name";
 ordered-by user;
 description
 "One access control rule.

 Rules are processed in user-defined order until a match is
 found. A rule matches if 'module-name', 'rule-type', and
 'access-operations' match the request. If a rule
 matches, the 'action' leaf determines if access is granted
 or not.";

 leaf name {
 type string {
 length "1..max";
 }
 description
 "Arbitrary name assigned to the rule.";
 }
}

```

```

leaf module-name {
 type union {
 type matchall-string-type;
 type string;
 }
 default "*";
 description
 "Name of the module associated with this rule.

 This leaf matches if it has the value '*' or if the
 object being accessed is defined in the module with the
 specified module name.";
}
choice rule-type {
 description
 "This choice matches if all leafs present in the rule
 match the request. If no leafs are present, the
 choice matches all requests.";
 case protocol-operation {
 leaf rpc-name {
 type union {
 type matchall-string-type;
 type string;
 }
 description
 "This leaf matches if it has the value '*' or if
 its value equals the requested protocol operation
 name.";
 }
 }
 case notification {
 leaf notification-name {
 type union {
 type matchall-string-type;
 type string;
 }
 description
 "This leaf matches if it has the value '*' or if its
 value equals the requested notification name.";
 }
 }
 case data-node {
 leaf path {
 type node-instance-identifier;
 mandatory true;
 description
 "Data Node Instance Identifier associated with the
 data node controlled by this rule.

 Configuration data or state data instance
 identifiers start with a top-level data node. A
 complete instance identifier is required for this
 type of path value.

 The special value '/' refers to all possible
 data-store contents.";
 }
 }
}

leaf access-operations {
 type union {

```

```

 type matchall-string-type;
 type access-operations-type;
 }
 default "";
 description
 "Access operations associated with this rule.

 This leaf matches if it has the value '*' or if the
 bit corresponding to the requested operation is set.";
}

leaf action {
 type action-type;
 mandatory true;
 description
 "The access control action associated with the
 rule. If a rule is determined to match a
 particular request, then this object is used
 to determine whether to permit or deny the
 request.";
}

leaf comment {
 type string;
 description
 "A textual description of the access rule.";
}
}

```

tailf-acm augments this with two additional leafs:

```

augment /nacm:nacm/nacm:rule-list/nacm:rule {

 leaf context {
 type union {
 type nacm:matchall-string-type;
 type string;
 }
 default "";
 description
 "This leaf matches if it has the value '*' or if its value
 identifies the agent that is requesting access, e.g. 'netconf'
 for NETCONF, 'cli' for CLI, or 'webui' for Web UI.";
 }

 leaf log-if-permit {
 type empty;
 description
 "If this leaf is present, access granted due to this rule
 is logged in the developer log. Otherwise, only denied
 access is logged. Mainly intended for debugging of rules.";
 }
}

```

Similar to the command access check, whenever a user through some agent tries to access an rpc, a notification, a data item, or an action, access is checked. For a rule to match, three or four leafs must match and when a match is found, the corresponding action is taken.

We have the following leafs in the rule list entry.



name	name is the name of the rule. The rules are checked in order, with the ordering given by the the YANG ordered-by user semantics, i.e. independent of the key values.
module-name	The module-name string is the name of the YANG module where the node being accessed is defined. The special value * (i.e. the default) matches all modules.

**Note**

Since the elements of the path to a given node may be defined in different YANG modules when augmentation is used, rules which have a value other than \* for the module-name leaf may require that additional processing is done before a decision to permit or deny or the access can be taken. Thus if an XPath that completely identifies the nodes that the rule should apply to is given for the path leaf (see below), it may be best to leave the module-name leaf unset.

rpc-name / notification-name / path	<p>This is a choice between three possible leafs that are used for matching, in addition to the module-name:</p> <p>rpc-name</p> <p>The name of a rpc operation, or "*" to match any rpc.</p> <p>notification-name</p> <p>The name of a notification, or "*" to match any notification.</p> <p>path</p> <p>A restricted XPath expression leading down into the populated XML tree. A rule with a path specified matches if it is equal to or shorter than the checked path. Several types of paths are allowed.</p> <ol style="list-style-type: none"> <li>1 Tagpaths that are not containing any keys. For example / ncs/live-device/live-status.</li> <li>2 Instantiated key: as in /devices/ device[name="x1"] /config/interface matches the interface configuration for managed device "x1" It's possible to have partially instantiated paths only containing some keys instantiated - i.e combinations of tagpaths and keypaths. Assuming a deeper tree, the path /devices/ device/config/interface[name="eth0"] matches the "eth0" interface configuration on all managed devices.</li> <li>3 Wild card at end as in: /services/web-site/* does not match the web site service instances, but rather all children of the web site service instances.</li> </ol> <p>Thus the path in a rule is matched against the path in the attempted data access. If the attempted access has a path that is equal to or longer than the rule path - we have a match.</p>
-------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

If none of the leafs rpc-name, notification-name, or path are set, the rule matches for any rpc, notification, data, or action access.

context	context is either of the strings <code>cli</code> , <code>netconf</code> , <code>webui</code> , <code>snmp</code> , or <code>*</code> for a data rule. Furthermore, when we initiate user sessions from MAAPI, we can choose any string we want. Similarly to command rules we can differentiate access depending on which agent is used to gain access.
access-operations	<code>access-operations</code> is used to match the operation that NSO tries to perform. It must be one or more of the "create", "read", "update", "delete" and "exec" values from the <code>access-operations-type</code> bits type definition in <code>ietf-netconf-acm.yang</code> , or <code>"*"</code> to match any operation.
action	This leaf has the same characteristics as the <code>action</code> leaf for command access.
log-if-permit	This leaf has the same characteristics as the <code>log-if-permit</code> leaf for command access.
comment	An optional textual description of the rule.

If no matching rule is found in any of the rule lists in any `rule-list` entry that matches the user's groups, the data model node for which access is requested is examined for presence of the NACM extensions:

- If the `nacm:default-deny-all` extension is specified for the data model node, access is denied.
- If the `nacm:default-deny-write` extension is specified for the data model node, and "create", "update", or "delete" access is requested, access is denied.

If examination of the NACM extensions did not result in access being denied, the value (permit or deny) of the relevant default leaf is examined:

- If "read" access is requested, the value of `/nacm/read-default` determines whether access is permitted or denied.
- If "create", "update", or "delete" access is requested, the value of `/nacm/write-default` determines whether access is permitted or denied.
- If "exec" access is requested, the value of `/nacm/exec-default` determines whether access is permitted or denied.

If access is permitted due to one of these default leaves, this augmentation from `tailf-acm.yang` is relevant:

```
augment /nacm:nacm {
 ...
 leaf log-if-default-permit {
 type empty;
 description
 "If this leaf is present, access granted due to one of
 /nacm/read-default, /nacm/write-default, /nacm/exec-default
 /nacm/cmd-read-default, or /nacm/cmd-exec-default
 being set to 'permit' is logged in the developer log.
 Otherwise, only denied access is logged. Mainly intended
 for debugging of rules.";
 }
}
```

I.e. it has the same effect as the `log-if-permit` leaf for the rule lists, but for the case where the value of one of the default leaves permits the access.

When NSO executes a command, the command rules in the authorization database are searched, The rules are tried in order, as described above. When a rule matches the operation (command) that NSO is attempting, the action of the matching rule is applied - whether permit or deny.

When actual data access is attempted, the data rules are searched. E.g. when a user attempts to execute `delete aaa` in the CLI, the user needs delete access to the entire tree `/aaa`.

Another example is if a CLI user writes `show configuration aaa` TAB it suffices to have read access to at least one item below `/aaa` for the CLI to perform the TAB completion. If no rule matches or an explicit deny rule is found, the CLI will not TAB complete.

Yet another example is if a user tries to execute `delete aaa authentication users`, we need to perform a check on the paths `/aaa` and `/aaa/authentication` before attempting to delete the subtree. Say that we have a rule for path `/aaa/authentication/users` which is an permit rule and we have a subsequent rule for path `/aaa` which is a deny rule. With this rule set the user should indeed be allowed to delete the entire `/aaa/authentication/users` tree but not the `/aaa` tree nor the `/aaa/authentication` tree.

We have two variations on how the rules are processed. The easy case is when we actually try to read or write an item in the configuration database. The execution goes like:

```
foreach rule {
 if (match(rule, path)) {
 return rule.action;
 }
}
```

The second case is when we execute TAB completion in the CLI. This is more complicated. The execution goes like:

```
rules = select_rules_that_may_match(rules, path);
if (any_rule_is_permit(rules))
 return permit;
else
 return deny;
```

The idea being that as we traverse (through TAB) down the XML tree, as long as there is at least one rule that can possibly match later, once we have more data, we must continue.

For example assume we have:

```
1 "/system/config/foo" --> permit
2 "/system/config" --> deny
```

If we in the CLI stand at `"/system/config"` and hit TAB we want the CLI to show `foo` as a completion, but none of the other nodes that exist under `/system/config`. Whereas if we try to execute `delete /system/config` the request must be rejected.

By default, NACM rules are configured for entire `tailf:action` or YANG 1.1 `action` statements, but not for input statement child leafs. To override this behaviour, and enable NACM rules on input leafs, set the following parameter to 'true': `/ncs-config/aaa/action-input-rules/enabled`. When enabled all action input leafs given to an action will be validated for NACM rules. If broad 'deny' NACM rules are used, you might need to add 'permit' rules for the affected action input leafs to allow actions to be used with parameters.

## NACM Rules and Services

By design NACM rules are ignored for changes done by services - FASTMAP, Reactive FASTMAP, or Nano services. The reasoning behind this is that a service package can be seen as a controlled way to

provide limited access to devices for a user group that is not allowed to apply arbitrary changes on the devices.

However, there are NSO installations where this behavior is not desired, and NSO administrators want to enforce NACM rules even on changes done by services. For this purpose, the leaf called `/nacm/enforce-nacm-on-services` is provided. By default, it is set to `false`.

Note however that currently, even with this leaf set to true, there are limitations. Namely, the post-actions for nano-services are run in a user session without any access checks. Besides that, NACM rules are not enforced on the read operations performed in the service callbacks.

It might be desirable to deny everything for a user group and only allow access to a specific service. This pattern could be used to allow an operator to provision the service, but deny everything else. While this pattern works for a normal FASTMAP service, there are some caveats for stacked services, Reactive FASTMAP and Nano services. For these kinds of services, in addition to the service itself, access should be provided to the user group for the following paths:

- In case of stacked services, the user group needs read and write access to the leaf "private/re-deploy-counter" under the bottom service. Otherwise, the user will not be able to re-deploy the service.
- In case of Reactive FASTMAP or Nano services, the user group needs read and write access to the following:
  - `/zombies`
  - `/side-effect-queue`
  - `/kickers`

## Device Group Authorization

In deployments with many devices it can become cumbersome to handle data authorization per device. To help with this there is a rule type that works on device group membership (for more on device groups, see the section called “Device Groups” in *User Guide*). To do this, devices are added to different device-groups and the rule type `device-group-rule` is used.

The IETF NACM rule type is augmented with a new rule type named `device-group-rule` which contains a leafref to the device-groups. See [Example 14, “Device Group model augmentation”](#).

### Example 14. Device Group model augmentation

```
augment "/nacm:nacm/nacm:rule-list/nacm:rule/nacm:rule-type" {
 case device-group-rule {
 leaf device-group {
 type leafref {
 path "/ncs:devices/ncs:device-group/ncs:name";
 }
 description
 "Which device group this rule applies to.";
 }
 }
}
```

In [Example 15, “Device group configuration”](#) we configure two device-groups based on different regions, and add devices to them.

### Example 15. Device group configuration

```
<devices>
 <device-group>
```

```

 <name>us_east</name>
 <device-name>cli0</device-name>
 <device-name>gen0</device-name>
 </device-group>
<device-group>
 <name>us_west</name>
 <device-name>nc0</device-name>
</device-group>
</devices>

```

In [Example 16, “NACM group configuration”](#) we configure an operator for the us\_east region:

#### Example 16. NACM group configuration

```

<nacm>
 <groups>
 <group>
 <name>us_east</name>
 <user-name>us_east_oper</user-name>
 </group>
 </groups>
</nacm>

```

In [Example 17, “Device Group Authorization rules”](#) we configure the device group rules and refer to the device group and the us\_east group.

#### Example 17. Device Group Authorization rules

```

<nacm>
 <rule-list>
 <name>us_east</name>
 <group>us_east</group>
 <rule>
 <name>us_east_read_permit</name>
 <device-group xmlns="http://tail-f.com/yang/ncs-acm/device-group-authorization">us_east</device-group>
 <access-operations>read</access-operations>
 <action>permit</action>
 </rule>
 <rule>
 <name>us_east_create_permit</name>
 <device-group xmlns="http://tail-f.com/yang/ncs-acm/device-group-authorization">us_east</device-group>
 <access-operations>create</access-operations>
 <action>permit</action>
 </rule>
 <rule>
 <name>us_east_update_permit</name>
 <device-group xmlns="http://tail-f.com/yang/ncs-acm/device-group-authorization">us_east</device-group>
 <access-operations>update</access-operations>
 <action>permit</action>
 </rule>
 <rule>
 <name>us_east_delete_permit</name>
 <device-group xmlns="http://tail-f.com/yang/ncs-acm/device-group-authorization">us_east</device-group>
 <access-operations>delete</access-operations>
 <action>permit</action>
 </rule>
 </rule-list>
</nacm>

```

In summary device group authorization gives a more compact configuration for deployments where devices can be grouped and authorization can be done on a device group basis.

Modifications on the device-group subtree is recommended to be controlled by a limited set of users.

## Authorization Examples

Assume that we have two groups, `admin` and `oper`. We want `admin` to be able to see and edit the XML tree rooted at `/aaa`, but we do not want users that are members of the `oper` group to even see the `/aaa` tree. We would have the following rule-list and rule entries. Note, here we use the XML data from `tailf-aaa.yang` to exemplify. The examples apply to all data, for all data models loaded into the system.

```
<rule-list>
 <name>admin</name>
 <group>admin</group>
 <rule>
 <name>tailf-aaa</name>
 <module-name>tailf-aaa</module-name>
 <path>/</path>
 <access-operations>read create update delete</access-operations>
 <action>permit</action>
 </rule>
</rule-list>
<rule-list>
 <name>oper</name>
 <group>oper</group>
 <rule>
 <name>tailf-aaa</name>
 <module-name>tailf-aaa</module-name>
 <path>/</path>
 <access-operations>read create update delete</access-operations>
 <action>deny</action>
 </rule>
</rule-list>
```

If we do not want the members of `oper` to be able to execute the NETCONF operation `edit-config`, we define the following rule-list and rule entries:

```
<rule-list>
 <name>oper</name>
 <group>oper</group>
 <rule>
 <name>edit-config</name>
 <rpc-name>edit-config</rpc-name>
 <context xmlns="http://tail-f.com/yang/acm">netconf</context>
 <access-operations>exec</access-operations>
 <action>deny</action>
 </rule>
</rule-list>
```

To spell it out, the above defines four elements to match. If NSO tries to perform a `netconf` operation, which is the operation `edit-config`, and the user which runs the command is member of the `oper` group, and finally it is an `exec` (execute) operation, we have a match. If so, the action is `deny`.

The `path` leaf can be used to specify explicit paths into the XML tree using XPath syntax. For example the following:

```
<rule-list>
 <name>admin</name>
 <group>admin</group>
 <rule>
 <name>bob-password</name>
```

```

 <path>/aaa/authentication/users/user[name='bob']/password</path>
 <context xmlns="http://tail-f.com/yang/acm">cli</context>
 <access-operations>read update</access-operations>
 <action>permit</action>
 </rule>
</rule-list>

```

Explicitly allows the admin group to change the password for precisely the bob user when the user is using the CLI. Had path been /aaa/authentication/users/user/password the rule would apply to all password elements for all users. Since the path leaf completely identifies the nodes that the rule applies to, we do not need to give tailf-aaa for the module-name leaf.

NSO applies variable substitution, whereby the username of the logged in user can be used in a path. Thus:

```

<rule-list>
 <name>admin</name>
 <group>admin</group>
 <rule>
 <name>user-password</name>
 <path>/aaa/authentication/users/user[name='$USER']/password</path>
 <context xmlns="http://tail-f.com/yang/acm">cli</context>
 <access-operations>read update</access-operations>
 <action>permit</action>
 </rule>
</rule-list>

```

The above rule allows all users that are part of the admin group to change their own passwords only.

A member of oper is able to execute NETCONF operation action if that member has exec access on NETCONF RPC action operation, read access on all instances in the hierarchy of data nodes that identifies the specific action in the datastore, and exec access on the specific action. For example an action is defined as below.

```

container test {
 action double {
 input {
 leaf number {
 type uint32;
 }
 }
 output {
 leaf result {
 type uint32;
 }
 }
 }
}

```

To be able to execute double action through NETCONF RPC, the members of oper need the following rule-list and rule-entries.

```

<rule-list>
 <name>oper</name>
 <group>oper</group>

 <rule>
 <name>allow-netconf-rpc-action</name>
 <rpc-name>action</rpc-name>
 <context xmlns="http://tail-f.com/yang/acm">netconf</context>
 <access-operations>exec</access-operations>
 </rule>

```

```

 <action>permit</action>
 </rule>
 <rule>
 <name>allow-read-test</name>
 <path>/test</path>
 <access-operations>read</access-operations>
 <action>permit</action>
 </rule>
 <rule>
 <name>allow-exec-double</name>
 <path>/test/double</path>
 <access-operations>exec</access-operations>
 <action>permit</action>
 </rule>
</rule-list>

```

Or, a simpler rule set as the following.

```

<rule-list>
 <name>oper</name>
 <group>oper</group>

 <rule>
 <name>allow-netconf-rpc-action</name>
 <rpc-name>action</rpc-name>
 <context xmlns="http://tail-f.com/yang/acm">netconf</context>
 <access-operations>exec</access-operations>
 <action>permit</action>
 </rule>
 <rule>
 <name>allow-exec-double</name>
 <path>/test</path>
 <access-operations>read exec</access-operations>
 <action>permit</action>
 </rule>
</rule-list>

```

Finally if we wish members of the oper group to never be able to execute the request system reboot command, also available as a reboot NETCONF rpc, we have:

```

<rule-list>
 <name>oper</name>
 <group>oper</group>

 <cmdrule xmlns="http://tail-f.com/yang/acm">
 <name>request-system-reboot</name>
 <context>cli</context>
 <command>request system reboot</command>
 <access-operations>exec</access-operations>
 <action>deny</action>
 </cmdrule>

 <!-- The following rule is required since the user can -->
 <!-- do "edit system" -->

 <cmdrule xmlns="http://tail-f.com/yang/acm">
 <name>request-reboot</name>
 <context>cli</context>
 <command>request reboot</command>
 <access-operations>exec</access-operations>
 <action>deny</action>
 </cmdrule>

```



```

<rule>
 <name>netconf-reboot</name>
 <rpc-name>reboot</rpc-name>
 <context xmlns="http://tail-f.com/yang/acm">netconf</context>
 <access-operations>exec</access-operations>
 <action>deny</action>
</rule>

</rule-list>

```

Debugging the AAA rules can be hard. The best way to debug rules that behave unexpectedly is to add the `log-if-permit` leaf to some or all of the rules that have `action permit`. Whenever such a rule triggers a permit action, an entry is written to the developer log.

Finally it is worth mentioning that when a user session is initially created it will gather the authorization rules that are relevant for that user session and keep these rules for the life of the user session. Thus when we update the AAA rules in e.g. the CLI the update will not apply to the current session - only to future user sessions.

## The AAA cache

NSO's AAA subsystem will cache the AAA information in order to speed up the authorization process. This cache must be updated whenever there is a change to the AAA information. The mechanism for this update depends on how the AAA information is stored, as described in the following two sections.

## Populating AAA using CDB

In order to start NSO, the data models for AAA must be loaded. The defaults in the case that no actual data is loaded for these models allow all read and exec access, while write access is denied. Access may still be further restricted by the NACM extensions, though - e.g. the `/nacm` container has `nacm:default-deny-all`, meaning that not even read access is allowed if no data is loaded.

The NSO installation ships with an XML initialization file containing AAA configuration. The file is called `aaa_init.xml` and is, by default, copied to the CDB directory by the NSO install scripts.

The local installation variant, targeting development only, defines two users, `admin` and `oper` with passwords set to `admin` and `oper` respectively for authentication. The two users belong to user groups with NACM rules restricting their authorization level. The system installation `aaa_init.xml` variant, targeting production deployment, defines NACM rules only as users are, by default, authenticated using PAM. The NACM rules target two user groups, `ncsadmin` and `ncsoper`. Users belonging to the `ncsoper` group are limited to read-only access.



### Note

The default `aaa_init.xml` file provided with the NSO system installation must not be used as-is in a deployment without reviewing and verifying that every NACM rule in the file matches

Normally the AAA data will be stored as configuration in CDB. This allows for changes to be made through NSO's transaction-based configuration management. In this case the AAA cache will be updated automatically when changes are made to the AAA data. If changing the AAA data via NSO's configuration management is not possible or desirable, it is alternatively possible to use the CDB operational data store for AAA data. In this case the AAA cache can be updated either explicitly e.g. by using the `maapi_aaa_reload()` function, see the `confd_lib_maapi(3)` in *Manual Pages* manual page, or by triggering a subscription notification by using the "subscription lock" when updating the CDB operational data store, see Chapter 14, *Using CDB in Development Guide*.

## Hiding the AAA tree

Some applications may not want to expose the AAA data to end users in the CLI or the Web UI. Two reasonable approaches exist here and both rely on the `tailf:export` statement. If a module has `tailf:export none` it will be invisible to all agents. We can then either use a transform whereby we define another AAA model and write a transform program which maps our AAA data to the data which must exist in `tailf-aaa.yang` and `ietf-netconf-acm.yang`. This way we can choose to export and expose an entirely different AAA model.

Yet another very easy way out, is to define a set of static AAA rules whereby a set of fixed users and fixed groups have fixed access to our configuration data. Possibly the only field we wish to manipulate is the password field.



## CHAPTER 10

# Upgrade

---

Upgrading the NSO software gives you access to new features and product improvements. Unfortunately, every change presents some risk, and upgrades are no exception.

To minimize the risk and make the upgrade process as painless as possible, this section describes the recommended procedures and practices to follow during an upgrade.

As usual, sufficient preparation avoids many pitfalls and makes the process more straightforward and less stressful.

- [Preparing for Upgrade, page 145](#)
- [Single Instance Upgrade, page 147](#)
- [Recover from Failed Upgrade, page 148](#)
- [NSO HA Version Upgrade, page 149](#)
- [Package Upgrade, page 153](#)
- [Patch Management, page 155](#)

## Preparing for Upgrade

There are multiple aspects that you should consider before starting with the actual upgrade procedure. While the development team tries to provide as much compatibility between software releases as possible, they cannot always avoid all incompatible changes. For example, when a deviation from an RFC standard is found and resolved, it may break clients that depend on the non-standard behavior. For this reason, a distinction is made between maintenance and a major NSO upgrade.

A maintenance NSO upgrade is within the same branch, i.e., when the first two version numbers stay the same (x.y in the x.y.z NSO version). An example is upgrading from version 6.1.1 to 6.1.2. In the case of a maintenance upgrade, the NSO release contains only corrections and minor enhancements, minimizing the changes. It includes binary compatibility for packages, so there is no need to recompile the .fxs files for a maintenance upgrade.

Correspondingly, when the first or second number in the version changes, that is called a full or major upgrade. For example, upgrading version 6.1.1 to 6.2 is a major, non-maintenance upgrade. Due to new features, packages must be recompiled, and some incompatibilities could manifest.

In addition to the above, a package upgrade is when you replace a package with a newer version, such as a NED or a service package. Sometimes, when package changes are not too big, it is possible to supply the new packages as part of the NSO upgrade, but this approach brings additional complexity. Instead,

package upgrade and NSO upgrade should in general, be performed as separate actions and are covered as such.

To avoid surprises during any upgrade, first ensure the following:

- Hosts have sufficient disk space, as some additional space is required for an upgrade.
- The software is compatible with the target OS. However, sometimes a newer version of Java or system libraries, such as glibc, may be required.
- All the required NEDs and custom packages are compatible with the target NSO version.
- Existing packages have been compiled for the new version and are available to you during the upgrade.
- Check whether the existing `ncs.conf` file can be used as-is or needs updating. For example, stronger encryption algorithms may require you to configure additional keying material.
- Review the `CHANGES` file for information on what has changed.
- If upgrading from a no longer supported software version, verify that the upgrade can be performed directly. In situations where the currently installed version is very old, you may have to upgrade to one or more intermediate versions before upgrading to the target version.

In case it turns out any of the packages are incompatible or cannot be recompiled, you will need to contact the package developers for an updated or recompiled version. For an official Cisco-supplied package, it is recommended that you always obtain a pre-compiled version if it is available for the target NSO release, instead of compiling the package yourself.

Additional preparation steps may be required based on the upgrade and the actual setup, such as when using the Layered Service Architecture (LSA) feature. In particular, for a major NSO upgrade in a multi-version LSA cluster, ensure that the new version supports the other cluster members and follow the additional steps outlined in Chapter 3, *Deploying LSA in Layered Service Architecture*.

If you use the High Availability (HA) feature, the upgrade consists of multiple steps on different nodes. To avoid mistakes, you are encouraged to script the process, for which you will need to set up and verify access to all NSO instances with either **ssh**, **nct**, or some other remote management command. For the reference example we use in this chapter, see `examples.ncs/development-guide/high-availability/hcc`. The management station uses shell and Python scripts that use **ssh** to access the Linux shell and NSO CLI and Python Requests for NSO RESTCONF interface access.

Likewise, NSO 5.3 added support for 256-bit AES encrypted strings, requiring the AES256CFB128 key in the `ncs.conf` configuration. You can generate one with the **openssl rand -hex 32** or a similar command. Alternatively, if you use an external command to provide keys, ensure that it includes a value for an `AES256CFB128_KEY` in the output.

Finally, regardless of the upgrade type, ensure that you have a working backup and can easily restore the previous configuration if needed, as described in [the section called “Backup and restore”](#).

**Caution**

The **ncs-backup** (and consequently the **nct backup**) command does not back up the `/opt/ncs/packages` folder. If you make any file changes, back them up separately.

However, the best practice is not to modify packages in the `/opt/ncs/packages` folder. Instead, if an upgrade requires package recompilation, separate package folders (or files) should be used, one for each NSO version.

## Single Instance Upgrade

The upgrade of a single NSO instance requires the following steps:

- 1 Create a backup.
- 2 Perform a system install of the new version.
- 3 Stop the old NSO server process.
- 4 Compact the CDB files write log.
- 5 Update the `/opt/ncs/current` symbolic link.
- 6 If required, update the `ncs.conf` configuration file.
- 7 Update the packages in `/var/opt/ncs/packages/` if recompilation is needed.
- 8 Start the NSO server process, instructing it to reload the packages.

The following steps suppose that you are upgrading to the 6.2 release. They pertain to a system install of NSO, and you must perform them with Super User privileges. As a best practice, always create a backup before trying to upgrade.

```
ncs-backup
```

For the upgrade itself, you must first download to the host and install the new NSO release.

```
sh nso-6.2.linux.x86_64.installer.bin --system-install
```

Then, you stop the currently running server with the help of the `init.d` script or an equivalent command relevant to your system.

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

Compact the CDB files write log using, for example, the **ncs --cdb-compact \$NCS\_RUN\_DIR/cdb** command.

Next, you update the symbolic link for the currently selected version to point to the newly installed one, 6.2 in this case.

```
cd /opt/ncs
rm -f current
ln -s ncs-6.2 current
```

While seldom necessary, at this point, you would also update the `/etc/ncs/ncs.conf` file.

Now, ensure that the `/var/opt/ncs/packages/` directory has appropriate packages for the new version. It should be possible to continue using the same packages for a maintenance upgrade. But for a major upgrade, you must normally rebuild the packages or use pre-built ones for the new version. You must ensure this directory contains the exact same version of each existing package, compiled for the new release, and nothing else.

As a best practice, the available packages are kept in `/opt/ncs/packages/` and `/var/opt/ncs/packages/` only contains symbolic links. In this case, to identify the release for which they were compiled, the package file names all start with the corresponding NSO version. Then, you only need to rearrange the symbolic links in the `/var/opt/ncs/packages/` directory.

```
cd /var/opt/ncs/packages/
rm -f *
for pkg in /opt/ncs/packages/ncs-6.2-*; do ln -s $pkg; done
```

Please note that the above package naming scheme is neither required nor enforced. If your package filesystem names differ from it, you will need to adjust the preceding command accordingly.

Finally, you start the new version of the NSO server with the package reload flag set.

```
/etc/init.d/ncs start-with-package-reload
Starting ncs: ...
```

NSO will perform the necessary data upgrade automatically. However, this process may fail if you have changed or removed any packages. In that case, ensure that the correct versions of all packages are present in `/var/opt/ncs/packages/` and retry the preceding command.

Also, note that with many packages or data entries in the CDB, this process could take more than 90 seconds and result in the following error message:

```
Starting ncs (via systemctl): Job for ncs.service failed
because a timeout was exceeded. See "systemctl status
ncs.service" and "journalctl -xe" for details. [FAILED]
```

The above error does not imply that NSO failed to start, just that it took longer than 90 seconds. Therefore, it is recommended you wait some additional time before verifying.

## Recover from Failed Upgrade

It is imperative you have a working copy of data available from which you can restore. That is why you must always create a backup before starting an upgrade. Only a backup guarantees that you can rerun the upgrade or back out of it, should it be necessary.

The same steps can also be used to restore data on a new, similar host if the OS of the initial host becomes corrupted beyond repair.

First, stop the NSO process if it is running.

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

Verify and, if necessary, revert the symbolic link in `/opt/ncs/` to point to the initial NSO release.

```
cd /opt/ncs
ls -l current
ln -s ncs-VERSION current
```

In the exceptional case where the initial version installation was removed or damaged, you will need to re-install it first and redo the step above.

Verify if the correct (initial) version of NSO is being used.

```
ncs --version
```

Next, restore the backup.

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

Finally, start the NSO server and verify the restore was successful.

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

## NSO HA Version Upgrade

Upgrading NSO in a highly available (HA) setup is a staged process. It entails running various commands across multiple NSO instances at different times.

The procedure described in this chapter is used with the rule-based built-in HA clusters. For HA Raft cluster instructions, please refer to [the section called “Version Upgrade of Cluster Nodes”](#).

The procedure is almost the same for a maintenance and major NSO upgrade. The difference is that a major upgrade requires the replacement of packages with recompiled ones. Still, a maintenance upgrade is often perceived as easier because there are fewer changes in the product.

The stages of the upgrade are:

- 1 First enable read-only mode on the designated *primary*, and then on the *secondary* that is enabled for fail-over.
- 2 Take a full backup on all nodes.
- 3 If using a 3-node setup, disconnect the 3rd, non-fail-over *secondary* by disabling HA on this node.
- 4 Disconnect the HA pair by disabling HA on the designated *primary*, temporarily promoting the designated *secondary* to provide the read-only service (and advertise the shared virtual IP address if it is used).
- 5 Upgrade the designated *primary*.
- 6 Disable HA on the designated *secondary* node, to allow designated *primary* to become actual primary in the next step.
- 7 Activate HA on the designated *primary*, which will assume its assigned (*primary*) role to provide the full service (and again advertise the shared IP if used). However, at this point, the system is without HA.
- 8 Upgrade the designated *secondary* node.
- 9 Activate HA on the designated *secondary*, which will assume its assigned (*secondary*) role, connecting HA again.
- 10 Verify that HA is operational and has converged.
- 11 Upgrade the 3rd, non-fail-over *secondary* if it is used, and verify it successfully re-joins the HA cluster.

Enabling the read-only mode on both nodes is required to ensure the subsequent backup captures the full system state, as well as making sure the *failover-primary* does not start taking writes when it is promoted later on.

Disabling the non-fail-over *secondary* in a 3-node setup right after taking backup is necessary when using the built-in HA rule-based algorithm (enabled by default in NSO 5.8 and later). Without it, the node might connect to the *failover-primary* when the fail over happens, which disables read-only mode.

While not strictly necessary, explicitly promoting the designated *secondary* after disabling HA on the *primary* ensures a fast fail over, avoiding the automatic reconnection attempts. If using a shared IP solution, such as the Tail-f HCC, this makes sure the shared VIP comes back up on the designated *secondary* as soon as possible. In addition, some older NSO versions do not reset the read-only mode upon disabling HA if they are not an acting *primary*.

Another important thing to note is that all packages used in the upgrade *must* match the NSO release. If they do not, the upgrade will fail.

In the case of a major upgrade, you must recompile the packages for the new version. It is highly recommended that you use pre-compiled packages and do not compile them during this upgrade procedure since the compilation can prove nontrivial, and the production hosts may lack all the required (development) tooling. You should use a naming scheme to distinguish between packages compiled for different NSO versions. A good option is for package file names to start with the `ncs-MAJORVERSION-` prefix for a given major NSO version. This ensures multiple packages can co-exist in the `/opt/ncs/packages` folder, and the NSO version they can be used with becomes obvious.

The following is a transcript of a sample upgrade procedure, showing the commands for each step described above, in a 2-node HA setup, with nodes in their initial designated state. The procedure ensures that is also the case at the end.

```
<switch to designated primary CLI>
admin@ncs# show high-availability status mode
high-availability status mode primary
admin@ncs# high-availability read-only mode true

<switch to designated secondary CLI>
admin@ncs# show high-availability status mode
high-availability status mode secondary
admin@ncs# high-availability read-only mode true

<switch to designated primary shell>
ncs-backup

<switch to designated secondary shell>
ncs-backup

<switch to designated primary CLI>
admin@ncs# high-availability disable

<switch to designated secondary CLI>
admin@ncs# high-availability be-primary

<switch to designated primary shell>
<upgrade node>
/etc/init.d/ncs restart-with-package-reload

<switch to designated secondary CLI>
admin@ncs# high-availability disable

<switch to designated primary CLI>
admin@ncs# high-availability enable

<switch to designated secondary shell>
<upgrade node>
/etc/init.d/ncs restart-with-package-reload

<switch to designated secondary CLI>
admin@ncs# high-availability enable
```

Scripting is a recommended way to upgrade the NSO version of an HA cluster. The following example script shows the required commands and can serve as a basis for your own customized upgrade script. In particular, the script requires a specific package naming convention above, and you may need to tailor it to your environment. In addition, it expects the new release version and the designated *primary* and *secondary* node addresses as the arguments. The recompiled packages are read from the `packages-MAJORVERSION/` directory.



For the below example script we configured our *primary* and *secondary* nodes with their nominal roles that they assume at startup and when HA is enabled. Automatic failover is also enabled so that the *secondary* will assume the *primary* role if the *primary* node goes down.

### Example 18. Configuration on Both Nodes

```
<config xmlns="http://tail-f.com/ns/config/1.0">
 <high-availability xmlns="http://tail-f.com/ns/ncs">
 <ha-node>
 <id>n1</id>
 <nominal-role>primary</nominal-role>
 </ha-node>
 <ha-node>
 <id>n2</id>
 <nominal-role>secondary</nominal-role>
 <failover-primary>true</failover-primary>
 </ha-node>
 <settings>
 <enable-failover>true</enable-failover>
 <start-up>
 <assume-nominal-role>true</assume-nominal-role>
 <join-ha>true</join-ha>
 </start-up>
 </settings>
 </high-availability>
</config>
```

### Example 19. Script for HA Major Upgrade (with Packages)

```
#!/bin/bash
set -ex

vsn=$1
primary=$2
secondary=$3
installer_file=nso-${vsn}.linux.x86_64.installer.bin
pkg_vsn=$(echo $vsn | sed -e 's/^\([0-9]\+\.[0-9]\+\)\.[0-9]*$/\1/')
pkg_dir="packages-${pkg_vsn}"

function on_primary() { ssh $primary "$@" ; }
function on_secondary() { ssh $secondary "$@" ; }
function on_primary_cli() { ssh -p 2024 $primary "$@" ; }
function on_secondary_cli() { ssh -p 2024 $secondary "$@" ; }

function upgrade_nso() {
 target=$1
 scp $installer_file $target:
 ssh $target "sh $installer_file --system-install --non-interactive"
 ssh $target "rm -f /opt/ncs/current && \
 ln -s /opt/ncs/ncs-${vsn} /opt/ncs/current"
}

function upgrade_packages() {
 target=$1
 do_pkgs=$(ls "${pkg_dir}/" || echo "")
 if [-n "${do_pkgs}"] ; then
 cd ${pkg_dir}
 ssh $target 'rm -rf /var/opt/ncs/packages/*'
 for p in ncs-${pkg_vsn}/*.gz; do
 scp $p $target:/opt/ncs/packages/
 ssh $target "ln -s /opt/ncs/packages/$p /var/opt/ncs/packages/"
 done
 cd -
 fi
}
```

```

 fi
}

Perform the actual procedure

on_primary_cli 'request high-availability read-only mode true'
on_secondary_cli 'request high-availability read-only mode true'

on_primary 'ncs-backup'
on_secondary 'ncs-backup'

on_primary_cli 'request high-availability disable'
on_secondary_cli 'request high-availability be-primary'
upgrade_nso $primary
upgrade_packages $primary
on_primary '/etc/init.d/ncs restart-with-package-reload'

on_secondary_cli 'request high-availability disable'
on_primary_cli 'request high-availability enable'
upgrade_nso $secondary
upgrade_packages $secondary
on_secondary '/etc/init.d/ncs restart-with-package-reload'

on_secondary_cli 'request high-availability enable'

```

Once the script completes, it is paramount that you manually verify the outcome. First, check that the HA is enabled by using the **show high-availability** command on the CLI of each node. Then connect to the designated secondaries and ensure they have the complete latest copy of the data, synchronized from the primaries.

After the *primary* node is upgraded and restarted, the read-only mode is automatically disabled. This allows the *primary* node to start processing writes, minimizing downtime. However, there is no HA. Should the *primary* fail at this point or you need to revert to a pre-upgrade backup, the new writes would be lost. To avoid this scenario, again enable read-only mode on the *primary* after re-enabling HA. Then disable read-only mode only after successfully upgrading and reconnecting the *secondary*.

To further reduce time spent upgrading, you can customize the script to install the new NSO release and copy packages beforehand. Then, you only need to switch the symbolic links and restart the NSO process to use the new version.

You can use the same script for a maintenance upgrade as-is, with an empty `packages-MAJORVERSION` directory, or remove the `upgrade_packages` calls from the script.

Example implementations that use scripts to upgrade a 2- and 3-node setup using CLI/MAAPI or RESTCONF are available in the NSO example set under `examples.ncs/development-guide/high-availability`.

We have been using a two node HCC layer 2 upgrade reference example elsewhere in the documentation to demonstrate installing NSO and adding the initial configuration. The *upgrade-l2* example referenced in `examples.ncs/development-guide/high-availability/hcc` implements shell and Python scripted steps to upgrade the NSO version using **ssh** to the Linux shell and the NSO CLI or Python Requests RESTCONF for accessing the *paris* and *london* nodes. See the example for details.

If you do not wish to automate the upgrade process, you will need to follow the instructions from [the section called “Single Instance Upgrade”](#) and transfer the required files to each host manually. Additional information on HA is available in [Chapter 7, High Availability](#). However, you can run the `high-availability` actions from the preceding script on the NSO CLI as-is. In this case, please take special care on which host you perform each command, as it can be easy to mix them up.

# Package Upgrade

Package upgrades are frequent and routine in development but require the same care as NSO upgrades in the production environment. The reason is that the new packages may contain an updated YANG model, resulting in a data upgrade process similar to a version upgrade. So, if a package is removed or uninstalled and a replacement is not provided, package-specific data, such as service instance data, will also be removed.

In a single-node environment, the procedure is straightforward. Create a backup with the **ncs-backup** command and ensure the new package is compiled for the current NSO version and available under the `/opt/ncs/packages` directory. Then either manually rearrange the symbolic links in the `/var/opt/ncs/packages` directory or use the **software packages install** command in the NSO CLI. Finally, invoke the **packages reload** command. For example:

```
ncs-backup
INFO Backup /var/opt/ncs/backups/ncs-6.2@2024-01-21T10:34:42.backup.gz created successfully
ls /opt/ncs/packages
ncs-6.2-router-nc-1.0 ncs-6.2-router-nc-1.0.2
ncs_cli -C
admin@ncs# software packages install package router-nc-1.0.2 replace-existing
installed ncs-6.2-router-nc-1.0.2
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 router-nc-1.0.2
 result true
}
```

On the other hand, upgrading packages in an HA setup is an error-prone process. Thus, NSO provides an action, **packages ha sync and-reload**, to minimize such complexity. This action loads new data models into NSO instead of restarting the server process. As a result, it is considerably more efficient, and the time difference to upgrade can be considerable if the amount of data in CDB is huge.



## Note

If the only change in the packages is addition of new NED packages, the `and-add` can replace `and-reload` command for an even more optimized and less intrusive update. See [the section called “Adding NED Packages”](#) for details.

The action executes on the *primary* node. First, it syncs the physical packages from the *primary* node to the *secondary* nodes as tar archive files, regardless if the packages were initially added as directories or tar archives. Then, it performs the upgrade on all nodes in one go. The action does not perform the sync and the upgrade on the node with *none* role.

The **packages ha sync** action distributes new packages to the *secondary* nodes. If a package already exists on the *secondary* node, it will replace it with the one on the *primary* node. Deleting a package on the *primary* node will also delete it on the *secondary* node. Packages found in load paths under the installation destination (by default `/opt/ncs/current`) are not distributed as they belong to the system and should not differ between the *primary* and the *secondary* nodes.

It is crucial to ensure that the load path configuration is identical on both *primary* and *secondary* nodes. Otherwise, the distribution will not start, and the action output will contain detailed error information.

Using the `and-reload` parameter with the action starts the upgrade once packages are copied over. The action sets the *primary* node to read-only mode. After the upgrade is successfully completed, the node is set back to its previous mode.

If the parameter `and-reload` is also supplied with the `wait-commit-queue-empty` parameter, it will wait for the commit queue to become empty on the primary node and prevent other queue items to be added while the queue is being drained.

Using the `wait-commit-queue-empty` parameter is the recommended approach, as it minimizes the risk of upgrade failing due to commit queue items still relying on the old schema.

### Example 20. Package Upgrade Procedure

```
primary@node1# software packages list
package {
 name dummy-1.0.tar.gz
 loaded
}
primary@node1# software packages fetch package-from-file \
$MY_PACKAGE_STORE/dummy-1.1.tar.gz
primary@node1# software packages install package dummy-1.1 replace-existing
primary@node1# packages ha sync and-reload { wait-commit-queue-empty }
```

The **packages ha sync and-reload** command has the following known limitations and side effects:

- The *primary* node is set to read-only mode before the upgrade starts, and it is set back to its previous mode if the upgrade is successfully upgraded. However, the node will always be in read-write mode if an error occurs during the upgrade. It is up to the user to set the node back to the desired mode by using the **high-availability read-only mode** command.
- As a best practice, you should create a backup of all nodes before upgrading. This action creates no backups, you must do that explicitly.

Example implementations that use scripts to upgrade a 2- and 3-node setup using CLI/MAAPI or RESTCONF are available in the NSO example set under `examples.ncs/development-guide/high-availability`.

We have been using a two-node HCC layer 2 upgrade reference example elsewhere in the documentation to demonstrate installing NSO and adding the initial configuration. The *upgrade-l2* example referenced in `examples.ncs/development-guide/high-availability/hcc` implements shell and Python scripted steps to upgrade the *primary paris* package versions and sync the packages to the *secondary london* using **ssh** to the Linux shell and the NSO CLI or Python Requests RESTCONF for accessing the *paris* and *london* nodes. See the example for details.

In some cases, NSO may warn when the upgrade looks "suspicious." For more information on this, please see [the section called "Loading Packages"](#). If you understand the implications and are willing to risk losing data, use the `force` option with **packages reload** or set the `NCS_RELOAD_PACKAGES` environment variable to `force` when restarting NSO. It will force NSO to ignore warnings and proceed with the upgrade. In general, this is not recommended.

In addition, you must take special care of NED upgrades because services depend on them. For example, since NSO 5 introduced the CDM feature, which allows loading multiple versions of a NED, a major NED upgrade requires a procedure involving the **migrate** action.

When a NED contains nontrivial YANG model changes, that is called a major NED upgrade. The NED ID changes, and the first or second number in the NED version changes since NEDs follow the same versioning scheme as NSO. In this case, you cannot simply replace the package, as you would for a

maintenance or patch NED release. Instead, you must load (add) the new NED package alongside the old one and perform the migration.

Migration uses the `/ncs:devices/device/migrate` action to change the ned-id of a single device or a group of devices. It does not affect the actual network device, except possibly reading from it. So, the migration does not have to be performed as part of the package upgrade procedure described above but can be done later, during normal operations. The details are described in [the section called “NED Migration”](#). Once the migration is complete, you can remove the old NED by performing another package upgrade, where you “deinstall” the old NED package. It can be done straight after the migration or as part of the next upgrade cycle.

## Patch Management

NSO can install emergency patches during runtime. These are delivered in the form of `.beam` files. You must copy the files into the `/opt/ncs/current/lib/ncs/patches/` folder and load them with the `ncs-state patches load-modules` command.





# CHAPTER 11

## Deployment Example

This chapter shows examples of a typical deployment for a highly available (HA) setup. A reference to an example implementation of the `tailf-hcc` layer-2 upgrade deployment scenario described here, check the NSO example set under `examples.ncs/development-guide/high-availability/hcc`. The example covers the following topics:

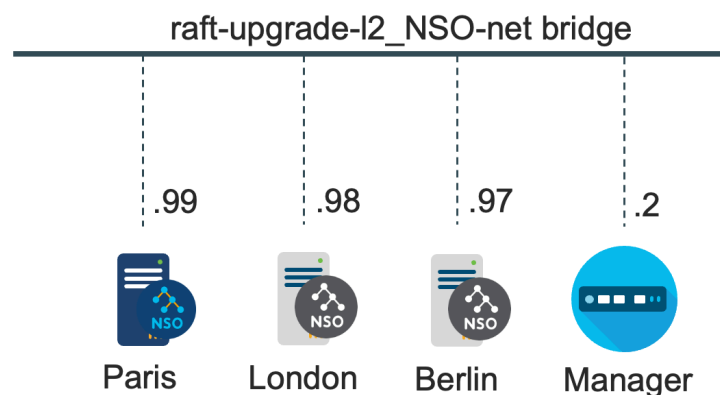
- Installation of NSO on all nodes in an HA setup
- Initial configuration of NSO on all nodes
- HA failover
- Upgrading NSO on all nodes in the HA cluster
- Upgrading NSO packages on all nodes in the HA cluster

The deployment examples use both the legacy rule-based and recommended HA Raft setup. See [Chapter 7, High Availability](#) for HA details. The HA Raft deployment consists of three nodes running NSO and a node managing them, while the rule-based HA deployment uses only two nodes.

Based on the Raft consensus algorithm, the HA Raft version provides the best fault tolerance, performance, and security and is therefore recommended.

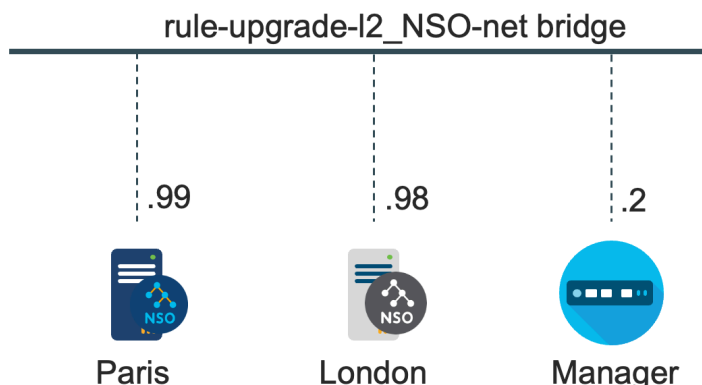
For the HA Raft setup, the NSO nodes *paris.fra*, *london.eng*, and *berlin.ger* nodes make up a cluster of one leader and two followers.

**Figure 21. The HA Raft Deployment Network**



For the rule-based HA setup, the NSO nodes *paris* and *london* make up one HA pair, one primary and one secondary.

Figure 22. The rule-based HA Deployment Network



HA is usually not optional for a deployment. Data resides in CDB, a RAM database with a disk-based journal for persistence. Both HA variants can be set up to avoid the need for manual intervention in a failure scenario, where HA Raft does the best job of keeping the cluster up. See [Chapter 7, High Availability](#) for details.

- [Initial NSO Installation](#), page 158
- [The `ncs.conf` Configuration](#), page 161
- [The `aaa\_init.xml` Configuration](#), page 162
- [The High Availability and VIP Configuration](#), page 163
- [Global Settings and Timeouts](#), page 164
- [Cisco Smart Licensing](#), page 164
- [Log Management](#), page 164
- [Monitoring the Installation](#), page 165
- [Security Considerations](#), page 167

## Initial NSO Installation

An NSO system installation on the NSO nodes is recommended for deployments. For system installation details, see the section called “System Install Steps” in *Getting Started*.

In this container-based example, Docker Compose uses a `Dockerfile` to build the container image and install NSO on multiple nodes, here containers. A shell script uses an SSH client to access the NSO nodes from the manager node to demonstrate HA failover and, as an alternative, a Python script that implements SSH and RESTCONF clients.

- An `admin` user is created on the NSO nodes.  
Password-less **sudo** access is set up to enable the `tailf-hcc` server to run the **ip** command.

The manager's SSH client uses public key authentication, while the RESTCONF client uses a token to authenticate with the NSO nodes.

The example creates two packages using the **ncs-make-package** command: `dummy` and `inert`. A third package, `tailf-hcc`, provides VIPs that point to the current HA leader/primary node.



- The packages are compressed into a `tar.gz` format for easier distribution, but that is not a requirement.

**Note**

While this deployment example uses containers, it is intended as a generic deployment guide. For details on running NSO in a container, such as Docker, see the Containerized NSO chapter. see the [Chapter 13, Containerized NSO](#).

This example uses a minimal Red Hat UBI distribution for hosting NSO with the following added packages:

- NSO's basic dependency requirements are fulfilled by adding the Java Runtime Environment (JRE), OpenSSH, and OpenSSL packages.
- The OpenSSH server is used for shell access and secure copy to the NSO Linux host for NSO version upgrade purposes. The NSO built-in SSH server provides CLI and NETCONF access to NSO.
- The NSO services require Python.
- To fulfill the `tailf-hcc` server dependencies, the `iproute2` utilities and `sudo` packages are installed. See [the section called “Dependencies”](#) in the `tailf-hcc` package chapter for details on the dependencies.
- The `rsyslog` package enables storing an NSO log file from several NSO logs locally and forwarding some logs to the manager.
- The `arp` command from the `net-tools` and `iputils (ping)` packages has been added for demonstration purposes.

The steps in the list below are performed as `root`. Docker Compose will build the container images, i.e., create the NSO installation as `root`.

The `admin` user will only need `root` access to run the `ip` command when `tailf-hcc` adds the Layer 2 VIP address to the leader/primary node interface.

The initialization steps are also performed as `root` for the nodes that make up the HA cluster:

- Create the `ncsadmin` and `ncsoper` Linux user groups.
- Create and add the `admin` and `oper` Linux users to their respective groups.
- Perform a system installation of NSO that runs NSO as the `admin` user.
- The `admin` user is granted access to run the `ip` command from the `vipctl` script as `root` using the `sudo` command as required by the `tailf-hcc` package.
- The `cmdwrapper` NSO program gets access to run the scripts executed by the **generate-token** action for generating RESTCONF authentication tokens as the current NSO user.
- Password authentication is set up for the read-only `oper` user for use with NSO only, which is intended for WebUI access.
- The `root` user is set up for Linux shell access only.
- The NSO installer, `tailf-hcc` package, application YANG modules, scripts for generating and authenticating RESTCONF tokens, and scripts for running the demo are all available to the NSO and manager containers.
- `admin` user permissions are set for the NSO directories and files created by the system install, as well as for the `root`, `admin`, and `oper` home directories.
- The `ncs.crypto_keys` are generated and distributed to all nodes.

**Note**

The `ncs.crypto_keys` file is highly sensitive. It contains the encryption keys for all encrypted CDB data, which often includes passwords for various entities, such as login credentials to managed devices.

**Note**

In an NSO system install setup, not only the TLS certificates (HA Raft) or shared token (rule-based HA) need to match between the HA cluster nodes, but also the configuration for encrypted strings, by default stored in `/etc/ncs/ncs.crypto_keys`, needs to match between the nodes in the HA cluster.

For rule-based HA, the tokens configured on the secondary nodes are overwritten with the encrypted token of type `aes-256-cfb-128-encrypted-string` from the primary node when the secondary connects to the primary. If there is a mismatch between the encrypted-string configuration on the nodes, NSO will not decrypt the HA token to match the token presented. As a result, the primary node denies the secondary node access the next time the HA connection needs to be re-established with a "Token mismatch, secondary is not allowed" error.

- For HA Raft, TLS certificates are generated for all nodes.
- The initial NSO configuration, `ncs.conf`, is updated and in sync (identical) on the nodes.
- The SSH servers are configured to allow only SSH public key authentication (no password). The `oper` user can use password authentication with the WebUI but has read-only NSO access.
- The `oper` user is denied access to the Linux shell.
- The `admin` user can access the Linux shell and NSO CLI using public key authentication.
- New keys for all users are distributed to the HA cluster nodes and the manager node when the HA cluster is initialized.
- The OpenSSH server and the NSO built-in SSH server use the same private and public key pairs located under `~/.ssh/id_ed25519`, while the manager public key is stored in the `~/.ssh/authorized_keys` file for both NSO nodes.
- Host keys are generated for all nodes to allow the NSO built-in SSH and OpenSSH servers to authenticate the server to the client.

Each HA cluster node has its own unique SSH host keys stored under `${NCS_CONFIG_DIR}/ssh_host_ed25519_key`. The SSH client(s), here the manager, has the keys for all nodes in the cluster paired with the node's hostname and the VIP address in its `/root/.ssh/known_hosts` file.

The host keys, like those used for client authentication, are generated each time the HA cluster nodes are initialized. The host keys are distributed to the manager and nodes in the HA cluster before the NSO built-in SSH and OpenSSH servers are started on the nodes.

- As NSO runs in containers, the environment variables are set to point to the system install directories in the Docker Compose `.env` file.

NSO runs as the non-root `admin` user and, therefore, the `ncs` command is used to start NSO instead of the `/etc/init.d/ncs` and `/etc/profile.d` scripts. The environment variables are copied to a `.pam_environment` file so that the `root` and `admin` users can set the required environment variables when those users access the shell via SSH.

- The start script is installed as part of the NSO system install, and it can be customized if you would like to use it to start NSO. The available NSO start script variants can be found under `/opt/ncs/current/src/ncs/package-skeletons/etc`. The scripts may provide what you need and can be used as a starting point.

- If you are running NSO as the root user and using **systemd**, the `init.d` script can be converted for use with **systemd**. Example:

```
$ mkdir -p /etc/init.d
$./nso-NSO_VERSION.linux.x86_64.installer.bin --system-install
$ cp /etc/init.d/ncs /etc/rc.d/init.d/
$ systemctl daemon-reload
$ echo "ExecReload=/etc/rc.d/init.d/ncs reload" << /run/systemd/generator/late/ncs.service
$ cp /run/systemd/generator/late/ncs.service /etc/systemd/system/
$ systemctl start ncs
```

- The OpenSSH **sshd** and **rsyslog** daemons are started.
- The packages from the package store are added to the `${NCS_RUN_DIR}/packages` directory before finishing the initialization part in the `root` context.
- The NSO smart licensing token is set.

## The ncs.conf Configuration

- The NSO IPC socket is configured in `ncs.conf` to only listen to localhost 127.0.0.1 connections, which is the default setting.

By default, the clients connecting to the NSO IPC socket are considered trusted, i.e., no authentication is required, and the use of 127.0.0.1 with the `/ncs-config/ncs-ipc-address` IP address in `ncs.conf` to prevent remote access. See [the section called “Security Considerations”](#) and `ncs.conf(5)` in *Manual Pages* for more details.

- `/ncs-config/aaa/pam` is set to enable PAM to authenticate users as recommended. All remote access to NSO must now be done using the NSO host's privileges. See `ncs.conf(5)` in *Manual Pages* for details.

- Depending on your Linux distribution, you may have to change the `/ncs-config/aaa/pam/service` setting. The default value is `common-auth`. Check the file `/etc/pam.d/common-auth` and make sure it fits your needs. See `ncs.conf(5)` in *Manual Pages* for details.

Alternatively, or as a complement to the PAM authentication, users can be stored in the NSO CDB database or authenticated externally. See [the section called “Authentication”](#) for details.

- RESTCONF token authentication under `/ncs-config/aaa/external-validation` is enabled using a `token_auth.sh` script that was added earlier together with a `generate_token.sh` script. See `ncs.conf(5)` in *Manual Pages* for details.

The scripts allow users to generate a token for RESTCONF authentication through, for example, the NSO CLI and NETCONF interfaces that use SSH authentication or the Web interface.

The token provided to the user is added to a simple YANG list of tokens where the list key is the username.

- The token list is stored in the NSO CDB operational data store and is only accessible from the node's local MAAPI and CDB APIs. See the HA Raft and rule-based HA `upgrade-l2/manager-etc/yang/token.yang` file in the examples.
- The NSO web server HTTPS interface should be enabled under `/ncs-config/webui`, along with `/ncs-config/webui/match-host-name = true` and `/ncs-config/webui/server-name` set to the hostname of the node, following security best practice. See `ncs.conf(5)` in *Manual Pages* for details.



**Note** The SSL certificates that NSO generates are self-signed:

```
$ openssl x509 -in /etc/ncs/ssl/cert/host.cert -text -noout
Certificate:
Data:
Version: 1 (0x0)
Serial Number: 2 (0x2)
Signature Algorithm: sha256WithRSAEncryption
Issuer: C=US, ST=California, O=Internet Widgits Pty Ltd, CN=John Smith
Validity
Not Before: Dec 18 11:17:50 2015 GMT
Not After : Dec 15 11:17:50 2025 GMT
Subject: C=US, ST=California, O=Internet Widgits Pty Ltd
Subject Public Key Info:
.....
```

Thus, if this is a production environment and the JSON-RPC and RESTCONF interfaces using the web server are not used solely for internal purposes, the self-signed certificate must be replaced with a properly signed certificate. See ncs.conf(5) in *Manual Pages* under /ncs-config/webui/transport/ssl/cert-file and /ncs-config/restconf/transport/ssl/certFile for more details.

- Disable /ncs-config/webui/cgi unless needed.
- The NSO SSH CLI login is enabled under /ncs-config/cli/ssh/enabled. See ncs.conf(5) in *Manual Pages* for details.
- The NSO CLI style is set to C-style, and the CLI prompt is modified to include the hostname under /ncs-config/cli/prompt. See ncs.conf(5) in *Manual Pages* for details.

```
<prompt1>\u@nso-\H> </prompt1>
<prompt2>\u@nso-\H% </prompt2>

<c-prompt1>\u@nso-\H# </c-prompt1>
<c-prompt2>\u@nso-\H(\m)# </c-prompt2>
```

- NSO HA Raft is enabled under /ncs-config/ha-raft, and the rule-based HA under /ncs-config/ha. See ncs.conf(5) in *Manual Pages* for details.
- Depending on your provisioned applications, you may want to turn /ncs-config/rollback/enabled off. Rollbacks do not work well with nano service reactive FASTMAP applications or if maximum transaction performance is a goal. If your application performs classical NSO provisioning, the recommendation is to enable rollbacks. Otherwise not. See ncs.conf(5) in *Manual Pages* for details.

## The aaa\_init.xml Configuration

The NSO system installation places an AAA aaa\_init.xml file in the \$NCS\_RUN\_DIR/cdb directory. Compared to a local installation for development, no users are defined for authentication in the aaa\_init.xml file, and PAM is enabled for authentication. NACM rules for controlling NSO access are defined in the file for users belonging to a ncsadmin user group and read-only access for a ncsoper user group. As seen in the previous sections, this example creates Linux root, admin, and oper users, as well as the ncsadmin and ncsoper Linux user groups.

PAM authenticates the users using SSH public key authentication without a passphrase for NSO CLI and NETCONF login. Password authentication is used for the oper user intended for NSO WebUI login and token authentication for RESTCONF login.

Before the NSO daemon is running, and there are no existing CDB files, the default AAA configuration in the `aaa_init.xml` is used. It is restrictive and is used for this demo with only a minor addition to allow the `oper` user to generate a token for RESTCONF authentication.

The NSO authorization system is group-based; thus, for the rules to apply to a specific user, the user must be a member of the group to which the restrictions apply. PAM performs the authentication, while the NSO NACM rules do the authorization.

- Adding the `admin` user to the `ncsadmin` group and the `oper` user to the limited `ncsoper` group will ensure that the two users get properly authorized with NSO.
- Not adding the `root` user to any group matching the NACM groups results in zero access, as no NACM rule will match, and the default in the `aaa_init.xml` file is to deny all access.

The NSO NACM functionality is based on the [Network Configuration Access Control Model](#) IETF RFC 8341 with NSO extensions augmented by `tailf-acm.yang`. See [Chapter 9, The AAA infrastructure](#), for more details.

The manager in this example logs into the different NSO hosts using the Linux user login credentials. This scheme has many advantages, mainly because all audit logs on the NSO hosts will show who did what and when. Therefore, the common bad practice of having a shared `admin` Linux user and NSO local user with a shared password is not recommended.

**Note**

The default `aaa_init.xml` file provided with the NSO system installation must not be used as-is in a deployment without reviewing and verifying that every NACM rule in the file matches the desired authorization level.

## The High Availability and VIP Configuration

This example sets up one HA cluster using HA Raft or rule-based HA with the `tailf-hcc` server to manage virtual IP addresses. See the [the section called “NSO Rule-based HA”](#) and [the section called “Tailf HCC Package”](#) for details.

The NSO HA, together with the `tailf-hcc` package, provides three features:

- All CDB data is replicated from the leader/primary to the follower/secondary nodes.
- If the leader/primary fails, a follower/secondary takes over and starts to act as leader/primary. This is how HA Raft works and how the rule-based HA variant of this example is configured to handle failover automatically.
- At failover, `tailf-hcc` sets up a virtual alias IP address on the leader/primary node only and uses gratuitous ARP packets to update all nodes in the network with the new mapping to the leader/primary node.

Nodes in other networks can be updated using the `tailf-hcc` layer-3 BGP functionality or a load balancer. See the NSO example set under `examples.ncs/development-guide/high-availability`.

See the NSO example set under `examples.ncs/development-guide/high-availability/hcc` for a reference to an HA Raft and rule-based HA `tailf-hcc` Layer 3 BGP examples.

The HA Raft and rule-based HA upgrade-l2 examples also demonstrate HA failover, upgrading the NSO version on all nodes, and upgrading NSO packages on all nodes.

## Global Settings and Timeouts

Depending on your installation, e.g., the size and speed of the managed devices and the characteristics of your service applications, some default values of NSO may have to be tweaked, particularly some of the timeouts.

- Device timeouts. NSO has connect, read, and write timeouts for traffic between NSO and the managed devices. The default value may not be sufficient if devices/nodes are slow to commit, while some are sometimes slow to deliver their full configuration. Adjust timeouts under `/devices/global-settings` accordingly.
- Service code timeouts. Some service applications can sometimes be slow. Adjusting the `/services/global-settings/service-callback-timeout` configuration might be applicable depending on the applications. However, the best practice is to change the timeout per service from the service code using the Java `ServiceContext.setTimeout` function or the Python `data_set_timeout` function.

There are quite a few different global settings for NSO. The two mentioned above often need to be changed.

## Cisco Smart Licensing

NSO uses Cisco Smart Licensing, which is described in detail in [Chapter 3, Cisco Smart Licensing](#). After registering your NSO instance(s), and receiving a token, following steps 1-6 as described in the Create a License Registration Token section of [Chapter 3, Cisco Smart Licensing](#), enter a token from your Cisco Smart Software Manager account on each host. Use the same token for all instances and script entering the token as part of the initial NSO configuration or from the management node:

```
admin@nso-paris# license smart register idtoken YzY2Yj...
admin@nso-london# license smart register idtoken YzY2Yj...
```



### Note

The Cisco Smart Licensing CLI command is present only in the Cisco Style CLI, which is the default CLI for this setup.

## Log Management

### Log Rotate

The NSO system installations performed on the nodes in the HA cluster also install defaults for **logrotate**. Inspect `/etc/logrotate.d/ncs` and ensure that the settings are what you want. Note that the NSO error logs, i.e., the files `/var/log/ncs/ncserr.log*`, are internally rotated by NSO and must not be rotated by **logrotate**.

### Syslog

For the HA Raft and rule-based HA upgrade-l2 examples, see the reference from the `examples.ncs/development-guide/high-availability/hcc/README`; the examples integrate with **rsyslog** to log the `ncs`, `developer`, `upgrade`, `audit`, `netconf`, `snmp`, and `webui-access` logs to syslog with facility set to `daemon` in `ncs.conf`.

**rsyslogd** on the nodes in the HA cluster is configured to write the `daemon` facility logs to `/var/log/daemon.log`, and forward the `daemon` facility logs with severity `info` or higher to the manager node's `/var/log/ha-cluster.log` syslog.

## Audit Network Log and NED Traces

Use the `audit-network-log` for recording southbound traffic towards devices. Enable by setting `/ncs-config/logs/audit-network-log/enabled` and `/ncs-config/logs/audit-network-log/file/enabled` to `true` in `$NCS_CONFIG_DIR/ncs.conf`. See `ncs.conf(5)` in *Manual Pages* for more information.

NED trace logs are a crucial tool for debugging NSO installations and not recommended for deployment. These logs are very verbose and for debugging only. Do not enable these logs in production.

Note that the NED logs include everything, even potentially sensitive data is logged. No filtering is done. The NED trace logs are controlled through the CLI under: `/device/global-settings/trace`. It is also possible to control the NED trace on a per-device basis under `/devices/device[name='x']/trace`.

There are three different settings for trace output. For various historical reasons, the setting that makes the most sense depends on the device type.

- For all CLI NEDs, use the `raw` setting.
- For all ConfD and netsim-based NETCONF devices, use the `pretty` setting. This is because ConfD sends the NETCONF XML unformatted, while `pretty` means that the XML is formatted.
- For Juniper devices, use the `raw` setting. Juniper devices sometimes send broken XML that cannot be formatted appropriately. However, their XML payload is already indented and formatted.
- For generic NED devices - depending on the level of trace support in the NED itself, use either `pretty` or `raw`.
- For SNMP-based devices, use the `pretty` setting.

Thus, it is usually not good enough to control the NED trace from `/devices/global-settings/trace`.

## Python Logs

While there is a global log for, for example, compilation errors in `/var/log/ncs/ncs-python-vm.log`, logs from user application packages are written to separate files for each package, and the log file naming is `ncs-python-vm-pkg_name.log`. The level of logging from Python code is controlled on a per package basis. See the section called “Debugging of Python packages” in *Development Guide* for more details.

## Java Logs

User application Java logs are written to `/var/log/ncs/ncs-java-vm.log`. The level of logging from Java code is controlled per Java package. See the section called “Logging” in *Development Guide* Java VM chapter for more details.

## Internal NSO Log

The internal NSO log resides at `/var/log/ncs/ncserr.*`. The log is written in a binary format. To view the internal error log, run the following command:

```
$ ncs --printlog /var/log/ncs/ncserr.log.1
```

## Monitoring the Installation

All large-scale deployments employ monitoring systems. There are plenty of good tools to choose from, open source and commercial. All good monitoring tools can script (using various protocols) what should

be monitored. It is recommended that a special read-only Linux user without shell access be set up like the `oper` user earlier in this chapter. A few commonly used checks include:

- At startup, check that NSO has been started using the `$NCS_DIR/bin/ncs_cmd -c "wait-start 2"` command.
- Use the `ssh` command to verify SSH access to the NSO host and NSO CLI.
- Check disk-usage using, for example, the `df` utility.
- For example, use `curl` or the Python requests library to verify that the RESTCONF API is accessible.
- Check that the NETCONF API is accessible using, for example, the `$NCS_DIR/bin/netconf-console` tool with a `hello` message.
- Verify the NSO version using, for example, the `$NCS_DIR/bin/ncs --version` or `RESTCONF /restconf/data/tailf-ncs-monitoring:ncs-state/version`.
- Check if HA is enabled using, for example, `RESTCONF /restconf/data/tailf-ncs-monitoring:ncs-state/ha`.

## Alarms

RESTCONF can be used to view the NSO alarm table and subscribe to alarm notifications. NSO alarms are not events. Whenever an NSO alarm is created, a RESTCONF notification and SNMP trap is also sent, assuming that you have a RESTCONF client registered with the alarm stream or configured a proper SNMP target. Some alarms, like the rule-based HA `ha-secondary-down` alarm, require the intervention of an operator. Thus, a monitoring tool should also fetch the NSO alarm list.

```
$ curl -ik -H "X-Auth-Token: TsZTNwJZoYWBhOPuOaMC6l4lCyXl+oDaasYqQZqqok=" \
https://paris:8888/restconf/data/tailf-ncs-alarms:alarms
```

Or subscribe to the `ncs-alarms` RESTCONF Notification stream.

## Metric - Counters, Gauges and Rate of Change Gauges

NSO metric has different contexts all containing different counters, gauges and rate of change gauges. There is a `sysadmin`, a `developer` and a `debug` context. Note that only the `sysadmin` context is enabled by default, as it is designed to be lightweight. Consult the YANG module `tailf-ncs-metric.yang` to learn the details of the different contexts.

### Counters

You may read counters by e.g. CLI, as in this example

```
admin@ncs# show metric sysadmin counter session cli-total
metric sysadmin counter session cli-total 1
```

### Gauges

You may read gauges by e.g. CLI, as in this example

```
admin@ncs# show metric sysadmin gauge session cli-open
metric sysadmin gauge session cli-open 1
```

### Rate of change gauges

You may read rate of change gauges by e.g. CLI, as in this example

```
admin@ncs# show metric sysadmin gauge-rate session cli-open
NAME RATE

```



```
1m 0.0
5m 0.2
15m 0.066
```

## Security Considerations

The presented configuration enables the built-in web server for the WebUI and RESTCONF interfaces. It is paramount for security that you only enable HTTPS access with `/ncs-config/webui/match-host-name` and `/ncs-config/webui/server-name` properly set.

The AAA setup described so far in this deployment document is the recommended AAA setup. To reiterate:

- Have all users that need access to NSO authenticated through Linux PAM. This may then be through `/etc/passwd`. Avoid storing users in CDB.
- Given the default NACM authorization rules, you should have three different types of users on the system.
  - Users with shell access are members of the `ncsadmin` Linux group and are considered fully trusted because they have full access to the system.
  - Users without shell access who are members of the `ncsadmin` Linux group have full access to the network. They have access to the NSO SSH shell and can execute RESTCONF calls, access the NSO CLI, make configuration changes, etc. However, they cannot manipulate backups or perform system upgrades unless such actions are added to by NSO applications.
  - Users without shell access that are members of the `ncsoper` Linux group have read-only access. They can access the NSO SSH shell, read data using RESTCONF calls, etc. However, they cannot change the configuration, manipulate backups and perform system upgrades.

If you have more fine-grained authorization requirements than read-write and read-only, additional Linux groups can be created, and the NACM rules can be updated accordingly. See [the section called “The `aaa\_init.xml` Configuration”](#) from earlier in this chapter on how the reference example implements users, groups, and NACM rules to achieve the above.

The default `aaa_init.xml` file must not be used as-is before reviewing and verifying that every NACM rule in the file matches the desired authorization level.

For a detailed discussion of the configuration of authorization rules through NACM, see [Chapter 9, \*The AAA infrastructure\*](#), particularly [the section called “Authorization”](#).

A considerably more complex scenario is when users require shell access to the host but are either untrusted or should not have any access to NSO at all. NSO listens to a so-called IPC socket configured through `/ncs-config/ncs-ipc-address`. This socket is typically limited to local connections and defaults to `127.0.0.1:4569` for security. The socket multiplexes several different access methods to NSO.

The main security-related point is that *no AAA checks* are performed on this socket. If you have access to the socket, you also have complete access to all of NSO.

To drive this point home, when you invoke the `ncs_cli` command, a small C program that connects to the socket and *tells* NSO who you are, NSO assumes that authentication has already been performed. There is even a documented flag `--noaaa`, which tells NSO to skip all NACM rule checks for this session.

You must protect the socket to prevent untrusted Linux shell users from accessing the NSO instance using this method. This is done by using a file in the Linux file system. The file `/etc/ncs/ncs_ipc_access` gets created and populated with random data at install time. Enable `/ncs-config/ncs-ipc-access-check/enabled` in `ncs.conf` and ensure that trusted users can read the `/etc/ncs/ncs_ipc_access` file, for example, by changing group access to the file. See `ncs.conf(5)` in *Manual Pages* for details.

```
$ cat /etc/ncs/ipc_access
cat: /etc/ncs/ipc_access: Permission denied
$ sudo chown root:ncsadmin /etc/ncs/ipc_access
$ sudo chmod g+r /etc/ncs/ipc_access
$ ls -lat /etc/ncs/ipc_access
$ cat /etc/ncs/ipc_access
.....
```

For an HA setup, HA Raft is based on the Raft consensus algorithm and provides the best fault tolerance, performance, and security. It is therefore recommended over the legacy rule-based HA variant. The `raft-upgrade-12` project, referenced from the NSO example set under `examples.ncs/development-guide/high-availability/hcc`, together with this Deployment Example chapter, describes a reference implementation. See [the section called “NSO HA Raft”](#) for more HA Raft details.



## CHAPTER 12

# Administration

---

- [User Management, page 169](#)
- [Packages, page 170](#)
- [Configuring NSO, page 173](#)
- [Monitoring NSO, page 173](#)
- [Backup and Restore, page 173](#)

## User Management

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

```
admin@ncs(config)# show full-configuration aaa authentication users user
aaa authentication users user admin
 uid 1000
 gid 1000
 password 1GNwimSPV$E82za8AaDxukAi8Ya8eSR.
 ssh_keydir /var/ncs/homes/admin/.ssh
 homedir /var/ncs/homes/admin
!
aaa authentication users user oper
 uid 1000
 gid 1000
 password 1yOstEhXy$nYKOQgs1CPyv9metoQALA.
 ssh_keydir /var/ncs/homes/oper/.ssh
 homedir /var/ncs/homes/oper
!...
```

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

```
admin@ncs(config)# show full-configuration nacm
nacm write-default permit
nacm groups group admin
 user-name [admin private]
!
nacm groups group oper
 user-name [oper public]
!
nacm rule-list admin
 group [admin]
 rule any-access
 action permit
 !
```

```

!
nacm rule-list any-group
group [*]
rule tailf-aaa-authentication
 module-name tailf-aaa
 path /aaa/authentication/users/user[name='$USER']
 access-operations read,update
 action permit
!

```

So, adding a user includes the following steps:

- 
- Step 1** Create the user: **admin@ncs(config)# aaa authentication users user <user-name>**
  - Step 2** Add the user to a NACM group: **admin@ncs(config)# nacm groups <group-name> admin user-name <user-name>**
  - Step 3** Verify/change access rules.
- 

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

```

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

```

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

```

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

```

## Packages

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

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

```

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

```

```

component cisco-ios
 ned cli ned-id cisco-ios
 ned cli java-class-name com.tailf.packages.ned.ios.IOSNedCli
 ned device vendor Cisco
NAME VALUE

show-tag interface

oper-status up

```

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

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

```

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

```

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

## Adding and upgrading a package

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

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

```
$ ncs-netsim stop
```

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

```

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

```

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

```

admin@ncs# show packages packages package
cisco-ios ... admin@ncs# packages reload

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

```

```
>>> System upgrade has completed successfully.
reload-result {
 package cisco-ios
 result true
}
reload-result {
 package cisco-nx
 result true
}
```

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

## Simulating the new device

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

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

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

## Adding the new devices to NSO

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

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

# Configuring NSO

## ncs.conf

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

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

## Run-time configuration

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

```
admin@ncs(config)# devices global-settings
```

Possible completions:

<code>backlog-auto-run</code>	Auto-run the backlog at successful connection
<code>backlog-enabled</code>	Backlog requests to non-responding devices
<code>commit-queue</code>	
<code>commit-retries</code>	Retry commits on transient errors
<code>connect-timeout</code>	Timeout in seconds for new connections
<code>ned-settings</code>	Control which device capabilities NCS uses
<code>out-of-sync-commit-behaviour</code>	Specifies the behaviour of a commit operation involving a device
<code>read-timeout</code>	Timeout in seconds used when reading data
<code>report-multiple-errors</code>	By default, when the NCS device manager commits data southbound, it will report the first error to the operator, this flag makes NCS report all errors
<code>trace</code>	Trace the southbound communication to devices
<code>trace-dir</code>	The directory where trace files are stored
<code>write-timeout</code>	Timeout in seconds used when writing data
<code>data</code>	

## Monitoring NSO

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

## Backup and Restore

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

The most convenient way to do backup and restore is to use the `ncs-backup` command. In that case the following procedure is used.

## Backup

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

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

```
ncs-backup
```

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

For more information on backup, refer to `ncs-backup(1)` in *Manual Pages*.

## NSO Restore

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



### Note

---

It is always advisable to stop NSO before performing Restore.

---

- First stop NSO if NSO is not stopped yet.

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

Then take the backup

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

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

- Start NSO.

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





## CHAPTER 13

# Containerized NSO

NSO can be deployed in your environment using a container, such as Docker. Cisco offers two pre-built images for this purpose that you can use to run NSO and build packages.



### Note

If you are migrating from an existing NSO System Install to a container-based setup, refer to the guidelines in [the section called “Migration to Containerized NSO”](#).





Running NSO in a container offers several benefits that you would generally expect from a containerized approach, such as ease of use and convenient distribution. More specifically, a containerized NSO approach allows you to:









- Run a container image of a specific version of NSO and your packages which can then be distributed as one unit.
- Deploy and distribute the same version across your production environment.
- Use the Development Image containing the necessary environment for compiling NSO packages.
- [Overview of NSO Images, page 175](#)
- [System Requirements, page 177](#)
- [Administrative Information, page 177](#)
- [Examples, page 183](#)

## Overview of NSO Images

Cisco provides the following two NSO images based on Red Hat UBI:

- Production Image
- Development Image

Intended Use	NSO Package Development	Build NSO Packages	Run NSO	NSO Installation Type
Development Host 				None or Local Install

Development Image 				System Install
Production Image 				System Install



**Note**

The Red Hat UBI is an OCI-compliant image that is freely distributable and independent of platform and technical dependencies. You can read more about Red Hat UBI [here](#), and about Open Container Initiative (OCI) [here](#).

## Production Image

The Production Image is a production-ready NSO image for system-wide deployment and use. It is a pre-built Red Hat UBI-based NSO image created on System Install in *Getting Started* and available from the [Cisco Software Download](#) site.

Use the pre-built image as the base image in the container file (e.g., Dockerfile) and mount your own packages (such as NEDs and service packages) to run a final image for your production environment (see examples below).



**Note**

Consult the Installation guide in *Getting Started* for information concerning installing NSO on a Docker host, building NSO packages, and more.



**Note**

See Chapter 5, *Developing and Deploying a Nano Service* in *Getting Started* for an example that uses the container to deploy an SSH-key-provisioning nano service. The `$NCS_DIR/examples.ncs/development-guide/nano-services/netsim-sshkey/README` provides a link to the container based deployment variant of the example. See the `setup_ncip.sh` script and `README` in the `netsim-sshkey` deployment example for details.

## Development Image

The Development Image is a separate standalone NSO image with the necessary environment and software for building packages. It is also a pre-built Red Hat UBI-based image provided specifically to address the developer needs of building packages. The image is available as a signed package (e.g., `nso-VERSION.container-image-dev.linux.ARCH.signed.bin`) from the Cisco [Software Download](#) site. You can run the Development Image in different ways and a simple tool for defining and running multi-container Docker applications is [Docker Compose](#) (described below). The container provides the necessary environment to build custom packages.

The Development Image adds a few Linux packages that are useful for development, such as Ant, JDK, net-tools, pip, etc. Additional Linux packages can be added using, for example, the `dnf` command. The `dnf list installed` command lists all the installed packages.

## Downloading and Extracting the Images

To fetch and extract NSO images:

**Step 1**

Go to Cisco's official [Software Download](#) site and search for "Network Services Orchestrator". Select the relevant NSO version in the drop-down list, e.g., "Crosswork Network Services Orchestrator 6" and click "Network Services Orchestrator Software". Locate the binary, which is delivered as a signed package (e.g., `nso-6.4.container-image-prod.linux.x86_64.signed.bin`).

**Note**

The signed archive file name has the pattern `nso-VERSION.container-image-PROD_DEV.linux.ARCH.signed.bin`, where:

- VERSION is the image's NSO version.
- PROD\_DEV is the type of the container, i.e., either `prod` for Production, or `dev` for Development.
- ARCH is the CPU architecture.

**Step 2**

Extract the image and other files from the signed package, for example:

```
sh nso-6.4.container-image-prod.linux.x86_64.signed.bin
```

## System Requirements

To run the images, make sure that your system meets the following requirements:

- A system running Linux x86\_64 or ARM64, or macOS x86\_64 or Apple Silicon. Linux for production.
- A container platform, such as Docker. Docker is the recommended platform and is used as an example in this guide for running NSO images. You may, however, use another container runtime of your choice.

**Note**

Since this guide uses Docker as an example, therefore the corresponding CLI commands and examples used in this guide are also Docker-specific. If you use another container runtime, make sure to use the respective commands.

**Note**

Docker on Mac uses a Linux VM to run the Docker engine, which is compatible with the normal Docker images built for Linux. You do not need to recompile your NSO-in-Docker images when moving between a Linux machine and Docker on Mac as they both essentially run Docker on Linux.

## Administrative Information

This section covers the necessary administrative information about the NSO Production Image.

## Migration to Containerized NSO

If you have NSO installed for production use using System Install, you can migrate to the Containerized NSO setup by following the instructions in this section. Migrating your Network Services Orchestrator (NSO) to a containerized setup can provide numerous benefits, including improved scalability, easier version management, and enhanced isolation of services.

The migration process is designed to ensure a smooth transition from a System-Installed NSO to a container-based deployment. Detailed steps guide you through preparing your existing environment,

exporting the necessary configurations and state data, and importing them into your new containerized NSO instance. During the migration, consider the container runtime you plan to use, as this impacts the migration process.

## Before You Start

- We recommend that you read through this guide to better understand the expectations, requirements, and functioning aspects of a containerized deployment.
- Verify the compatibility of your current system configurations with the containerized NSO setup. See [the section called “System Requirements”](#) for more information.
- Determine and install the container orchestration tool you plan to use (e.g., Docker, etc.).
- Ensure that your current NSO installation is fully operational and backed up and that you have a clear rollback strategy in case any issues arise. Pay special attention to customizations and integrations that your current NSO setup might have, and verify their compatibility with the containerized version of NSO.
- Have a contingency plan in place for quick recovery in case any issues are encountered during migration.

## Migration Steps

### Prepare

- 1 Document your current NSO environment's specifics, including custom configurations and packages.
- 2 Perform a complete backup of your existing NSO instance, including configurations, packages, and data.
- 3 Set up the container environment and download/extract the NSO production image. See [the section called “Downloading and Extracting the Images”](#) for details.

### Migrate

- 1 Stop the current NSO instance.
- 2 Save the run directory from the NSO instance in an appropriate place.
- 3 Use the same `ncs.conf` and High Availability (HA) setup previously used with your System Install. We assume that the `ncs.conf` follows the best practice and uses the `NCS_DIR`, `NCS_RUN_DIR`, `NCS_CONFIG_DIR`, and `NCS_LOG_DIR` variables for all paths. The `ncs.conf` can be added to a volume and mounted to `/nso/etc` in the container.

```
docker container create --name temp -v NSO-evol:/nso/etc hello-world
docker cp ncs.conf temp:/nso/etc
docker rm temp
```

- 4 Add the run directory as a volume, mounted to `/nso/run` in the container and copy the CDB data, packages, etc., from the previous System Install instance.

```
cd path-to-previous-run-dir
docker container create --name temp -v NSO-rvol:/nso/run hello-world
docker cp . temp:/nso/run
docker rm temp
```

- 5 Create a volume for the log directory.

```
docker volume create --name NSO-lvol
```

- 6 Start the container. Example:

```
docker run -v NSO-rvol:/nso/run -v NSO-evol:/nso/etc -v NSO-lvol:/log -itd \
--name cisco-nso -e EXTRA_ARGS=--with-package-reload -e ADMIN_USERNAME=admin \
-e ADMIN_PASSWORD=admin cisco-nso-prod:6.4
```

### Finalize

- 1 Ensure that the containerized NSO instance functions as expected and validate system operations.
- 2 Plan and execute your cutover transition from the System-Installed NSO to the containerized version with minimal disruption.
- 3 Monitor the new setup thoroughly to ensure stability and performance.

## ncs.conf File Configuration and Preference

The `run-nso.sh` script runs a check at startup to determine which `ncs.conf` file to use. The order of preference is as below:

- 1 The `ncs.conf` file specified in the Dockerfile (i.e., `ENV $NCS_CONFIG_DIR /etc/ncs/`) is used as the first preference.
- 2 The second preference is to use the `ncs.conf` file mounted in the `/nso/etc/run` directory.
- 3 If no `ncs.conf` file is found at either `/etc/ncs` or `/nso/etc`, the default `ncs.conf` file provided with the NSO image in `/defaults` is used.



#### Note

If the `ncs.conf` file is edited after startup, it can be reloaded using MAAPI `reload_config()`. Example: `$ ncs_cmd -c "reload"`.

## Pre- and Post-Start Scripts

If you need to perform operations before or after the `ncs` process is started in the Production container, you can use Python and/or Bash scripts to achieve this. Add the scripts to the `$NCS_CONFIG_DIR/pre-ncs-start.d/` and `$NCS_CONFIG_DIR/post-ncs-start.d/` directories to have the `run-nso.sh` script run them.

## Admin User Creation

An admin user can be created on startup by the run script in the container. There are three environment variables that control the addition of an admin user:

- `ADMIN_USERNAME`: Username of the admin user to add, default is `admin`.
- `ADMIN_PASSWORD`: Password of the admin user to add.
- `ADMIN_SSHKEY`: Private SSH key of the admin user to add.

As `ADMIN_USERNAME` already has a default value, only `ADMIN_PASSWORD`, or `ADMIN_SSHKEY` need to be set in order to create an admin user. For example:

```
docker run -itd --name cisco-nso -e ADMIN_PASSWORD=admin cisco-nso-prod:6.4
```

This can be useful when starting up a container in CI for testing or development purposes. It is typically not required in a production environment where there is a permanent CDB that already contains the required user accounts.

**Note**

When using a permanent volume for CDB, etc., and restarting the NSO container multiple times with a different `ADMIN_USERNAME` or `ADMIN_PASSWORD`, note that the start script uses the `ADMIN_USERNAME` and `ADMIN_PASSWORD` environment variables to generate an XML file to the CDB directory which NSO reads at startup. When restarting NSO, if the persisted CDB configuration file already exists in the CDB directory, NSO will only load the persisted configuration and no XML files at startup, and the generated `add_admin_user.xml` in the CDB directory needs to be loaded by the application, using, for example, the **`ncs_load`** command.

**Note**

The default `ncs.conf` file performs authentication using only the Linux PAM, with local authentication disabled. For the `ADMIN_USERNAME`, `ADMIN_PASSWORD`, and `ADMIN_SSHKEY` variables to take effect, NSO's local authentication, in `/ncs-conf/aaa/local-authentication`, needs to be enabled. Alternatively, you can create a local Linux admin user that is authenticated by NSO using Linux PAM.

## Exposing Ports

The default `ncs.conf` NSO configuration file does not enable any northbound interfaces, and no ports are exposed externally to the container. Ports can be exposed externally of the container when starting the container with the northbound interfaces and their ports enabled in `ncs.conf`.

## Backup and Restore

The backup behavior of running NSO in vs. outside the container is largely the same, except that when running NSO in a container, the SSH and SSL certificates are not included in the backup produced by the `ncs-backup` script. This is different from running NSO outside a container where the default configuration path `/etc/ncs` is used to store the SSH and SSL certificates, i.e., `/etc/ncs/ssh` for SSH and `/etc/ncs/ssl` for SSL.

### Take a Backup

Let's assume we start a production image container using:

```
docker run -d --name cisco-nso -v NSO-vol:/nso -v NSO-log-vol:/log cisco-nso-prod:6.4
```

To take a backup:

- Run the **`ncs-backup`** command. The backup file is written to `/nso/run/backups`.

```
docker exec -it cisco-nso ncs-backup
INFO Backup /nso/run/backups/ncs-6.4@2024-11-03T11:31:07.backup.gz created successfully
```

### Restore a Backup

To restore a backup, NSO must not be running. As you likely only have access to the `ncs-backup` tool, the volume containing CDB, and other run-time data from inside the NSO container, this poses a slight challenge. Additionally, shutting down NSO will terminate the NSO container.

To restore a backup:

**Step 1** Shut down the NSO container.

```
docker stop cisco-nso
docker rm cisco-nso
```

**Step 2**

Run the **ncs-backup --restore** command. Start a new container with the same persistent shared volumes mounted but with a different command. Instead of running the `/run-nso.sh`, which is the normal command of the NSO container, run the **restore** command.

```
docker run -it --rm --volumes-from cisco-nso -v NSO-vol:/nso -v NSO-log-vol:/log \
--entrypoint ncs-backup cisco-nso-prod:6.4 \
--restore /nso/run/backups/ncs-6.4@2024-11-03T11:31:07.backup.gz
```

```
Restore /etc/ncs from the backup (y/n)? y
Restore /nso/run from the backup (y/n)? y
INFO Restore completed successfully
```

**Step 3**

Restoring an NSO backup should move the current run directory (`/nso/run` to `/nso/run.old`) and restore the run directory from the backup to the main run directory (`/nso/run`). After this is done, start the regular NSO container again as usual.

```
docker run -d --name cisco-nso -v NSO-vol:/nso -v NSO-log-vol:/log cisco-nso-prod:6.4
```

## SSH Host Key

The NSO image `/run-nso.sh` script looks for an SSH host key named `ssh_host_ed25519_key` in the `/nso/etc/ssh` directory to be used by the NSO built-in SSH server for the CLI and NETCONF interfaces.

If an SSH host key exists, which is for a typical production setup stored in a persistent shared volume, it remains the same after restarts or upgrades of NSO. If no SSH host key exists, the script generates a private and public key.

In a high-availability (HA) setup, the host key is typically shared by all NSO nodes in the HA group and stored in a persistent shared volume. I.e., each NSO node does not generate its host key to avoid fetching the public host key after each failover from the new primary to access the primary's NSO CLI and NETCONF interfaces.

## HTTPS TLS Certificate

NSO expects to find a TLS certificate and key at `/nso/ssl/cert/host.cert` and `/nso/ssl/cert/host.key` respectively. Since the `/nso` path is usually on persistent shared volume for production setups, the certificate remains the same across restarts or upgrades.

If no certificate is present, one will be generated. It is a self-signed certificate valid for 30 days making it possible to use both in development and staging environments. It is not meant for the production environment. You should replace it with a properly signed certificate for production and it is encouraged to do so even for test and staging environments. Simply generate one and place it at the provided path, for example using the following, which is the command used to generate the temporary self-signed certificate:

```
openssl req -new -newkey rsa:4096 -x509 -sha256 -days 30 -nodes \
-out /nso/ssl/cert/host.cert -keyout /nso/ssl/cert/host.key \
-subj "/C=SE/ST=NA/L=/O=NSO/OU=WebUI/CN=Mr. Self-Signed"
```

## YANG Model Changes (destructive)

The database in NSO, called CDB, uses YANG models as the schema for the database. It is only possible to store data in CDB according to the YANG models that define the schema.

If the YANG models are changed, particularly if the nodes are removed or renamed (rename is the removal of one leaf and an addition of another), any data in CDB for those leaves will also be removed. NSO normally warns about this when you attempt to load new packages, for example, **request packages reload** command refuses to reload the packages if the nodes in the YANG model have disappeared. You would then have to add the **force** argument, e.g., **request packages reload force**.

## Health Check

The base Production Image comes with a basic container health check. It uses **ncs\_cmd** to get the state that NCS is currently in. Only the result status is observed to check if **ncs\_cmd** was able to communicate with the **ncs** process. The result indicates if the **ncs** process is responding to IPC requests.



### Note

The default `--health-start-period` duration in health check is set to 60 seconds. NSO will report an unhealthy state if it takes more than 60 seconds to start up. To resolve this, set the `--health-start-period` duration value to a relatively higher value, such as 600 seconds, or however long you expect NSO will take to start up.

To disable the health check, use the **--no-healthcheck** command.

## NSO System Dump and Disable Memory Overcommit

By default, the Linux kernel allows overcommit of memory. However, memory overcommit produces an unexpected and unreliable environment for NSO since the Linux Out Of Memory Killer, or OOM-killer, may terminate NSO without restarting it if the system is critically low on memory.

Also, when the OOM-killer terminates NSO, NSO will *not produce a system dump file*, and the *debug information will be lost*. Thus, it is strongly recommended that overcommit is disabled with Linux NSO production container hosts with an overcommit ratio of less than 100% (max).

See the section called “4. Run the Installer” in *Getting Started* for information on the memory overcommit recommendations for a Linux system hosting NSO production containers.



### Note

By default, NSO writes a system dump to the NSO run-time directory, default `NCS_RUN_DIR=/nso/run`. If the `NCS_RUN_DIR` is not mounted on the host or to give the NSO system dump file a unique name, the `NCS_DUMP="/path/to/mounted/dir/ncs_crash.dump.<my-timestamp>"` variable need to be set.



### Note

The **docker run** command **--memory="[ram]"** and **--memory-swap="[ram+swap]"** option settings can be used to limit Docker container memory usage. The default setting is max, i.e., all of the host memory is used. Suppose the Docker container reaches a memory limit set by the `--memory` option. In that case, the default Docker setting is to have Docker terminate the container, *no NSO system dump will be generated*, and the *debug information will be lost*.

## Startup Arguments

The `/nso-run.sh` script that starts NSO is executed as an ENTRYPOINT instruction and the `CMD` instruction can be used to provide arguments to the `entrypoint-script`. Another alternative is to use the `EXTRA_ARGS` variable to provide arguments. The `/nso-run.sh` script will first check the `EXTRA_ARGS` variable before the `CMD` instruction.



An example using **docker run** with the CMD instruction:

```
docker run --name nso -itd cisco-nso-prod:6.4 --with-package-reload \
--ignore-initial-validation
```

With the EXTRA\_ARGS variable:

```
docker run --name nso \
-e EXTRA_ARGS='--with-package-reload --ignore-initial-validation' \
-itd cisco-nso-prod:6.4
```

An example using a Docker Compose file, `compose.yaml`, with the CMD instruction:

```
services:
 nso:
 image: cisco-nso-prod:6.4
 container_name: nso
 command:
 - --with-package-reload
 - --ignore-initial-validation
```

With the EXTRA\_ARGS variable:

```
services:
 nso:
 image: cisco-nso-prod:6.4
 container_name: nso
 environment:
 - EXTRA_ARGS=--with-package-reload --ignore-initial-validation
```

## Examples

This section provides examples to exhibit the use of NSO images.

### Running the Production Image using Docker CLI

This example shows how to run the standalone NSO Production Image using the Docker CLI.

The instructions and CLI examples used in this example are Docker-specific. If you are using a non-Docker container runtime, you will need to: fetch the NSO image from the Cisco software download site, then load and run the image with packages and networking, and finally log in to NSO CLI to run commands.

If you intend to run multiple images (i.e., both Production and Development), Docker Compose is a tool that simplifies defining and running multi-container Docker applications. See the example ([the section called “Running the NSO Images using Docker Compose”](#)) below for detailed instructions.

#### Steps

Follow the steps below to run the Production Image using Docker CLI:

- 
- |               |                                                                                                                                                                                                                                                                            |
|---------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Step 1</b> | Start your container engine.                                                                                                                                                                                                                                               |
| <b>Step 2</b> | Next, load the image and run it. Navigate to the directory where you extracted the base image and load it. This will restore the image and its tag:                                                                                                                        |
|               | <pre>docker load -i nso-6.4.container-image-prod.linux.x86_64.tar.gz</pre>                                                                                                                                                                                                 |
| <b>Step 3</b> | Start a container from the image. Supply additional arguments to mount the packages and <code>ncs.conf</code> as separate volumes ( <b>-v flag</b> ), and publish ports for networking ( <b>-p flag</b> ) as needed. The container starts NSO using the <code>/run-</code> |

nso.sh script. To understand how the ncs.conf file is used, see [the section called “ncs.conf File Configuration and Preference”](#).

```
docker run -itd --name cisco-nso \
-v NSO-vol:/nso \
-v NSO-log-vol:/log \
--net=host \
-e ADMIN_USERNAME=admin\
-e ADMIN_PASSWORD=admin\
cisco-nso-prod:6.4
```



#### Note

Overriding basic environment variables (NCS\_CONFIG\_DIR, NCS\_LOG\_DIR, NCS\_RUN\_DIR, etc.) is not supported and therefore should be avoided. Using, for example, the NCS\_CONFIG\_DIR environment variable to mount a configuration directory will result in an error. Instead, to mount your configuration directory, do it appropriately in the correct place, which is under /nso/etc.

The following examples show how to run the image with and without named volumes.

#### Running without a named volume

This is the minimal way of running the image but does not provide any persistence when the container is destroyed.

```
docker run -itd --name cisco-nso \
-p 8888:8888 \
-e ADMIN_USERNAME=admin\
-e ADMIN_PASSWORD=admin\
cisco-nso-prod
```

#### Running with a single named volume

This provides persistence for the NSO mount point with a NSO-vol volume. Logs, however, are not persistent.

```
docker run -itd --name cisco-nso \
-v NSO-vol:/nso \
-p 8888:8888 \
-e ADMIN_USERNAME=admin\
-e ADMIN_PASSWORD=admin\
cisco-nso-prod
```

#### Running with two named volumes

This provides full persistence for both the NSO and the log mount points.

```
docker run -itd --name cisco-nso \
-v NSO-vol:/nso \
-v NSO-log-vol:/log \
-p 8888:8888 \
-e ADMIN_USERNAME=admin\
-e ADMIN_PASSWORD=admin\
cisco-nso-prod
```



**Note** Loading the packages by mounting the default load path `/nso/run` as a volume is preferred. You can also load the packages by copying them manually into the `/nso/run/packages` directory in the container. During development, a bind mount of the package directory on the host machine makes it easy to update packages in NSO by simply changing the packages on the host.



**Note** The default load path is configured in the `ncs.conf` file as `$NCS_RUN_DIR/packages`, where `$NCS_RUN_DIR` expands to `/nso/run` in the container. To find the load path, check the `ncs.conf` file in the `/etc/ncs/` directory.

```
<load-path>
<dir>${NCS_RUN_DIR}/packages</dir>
<dir>${NCS_DIR}/etc/ncs</dir>
...
</load-path>
```

With the Production Image, use a shared volume to persist data across restarts. If remote (Syslog) logging is used, there is little need to persist logs. If local logging is used, then persistent logging is recommended.



**Note** NSO starts a cron job to handle logrotate of NSO logs by default. i.e., the `CRON_ENABLE` and `LOGROTATE_ENABLE` variables are set to `true` using the `/etc/logrotate.conf` configuration. See the `/etc/ncs/post-ncs-start.d/10-cron-logrotate.sh` script. To set how often the cron job runs, use the `crontab` file.

#### Step 4

Finally, log in to NSO CLI to run commands. Open an interactive shell on the running container and access the NSO CLI.

```
docker exec -it cisco-nso bash
ncs_cli -u admin
admin@ncs>
```



**Note** You can also use the `docker exec -it cisco-nso ncs_cli -u admin` command to access the CLI from the host's terminal.

## Upgrading NSO using Docker CLI

This example describes how to upgrade your NSO to run a newer NSO version in container. The overall upgrade process is outlined in the steps below. In the example below, NSO is to be upgraded from version 6.3 to 6.4.

To upgrade your NSO version:

#### Step 1

Start a container with the `docker run` command. In the example below, it mounts the `/nso` directory in the container to the `NSO-vol` named volume to persist the data. Another option is using a bind mount of the directory on the host machine. At this point, the `/cdb` directory is empty.

- Step 2** `docker run -itd --name cisco-nso -v NSO-vol:/nso cisco-nso-prod:6.3`  
Perform a backup, either by running the **docker exec** command (make sure that the backup is placed somewhere we have mounted) or by creating a tarball of `/data/nso` on the host machine.
- Step 3** `docker exec -it cisco-nso ncs-backup`  
Stop the NSO by issuing the following command, or by stopping the container itself which will run the **ncs stop** command automatically.
- Step 4** `docker exec -it cisco-nso ncs --stop`  
Remove the old NSO
- Step 5** `docker rm -f cisco-nso`  
Start a new container and mount the `/nso` directory in the container to the `NSO-vol` named volume. This time the `/cdb` folder is not empty, so instead of starting a fresh NSO, an upgrade will be performed.
- `docker run -itd --name cisco-nso -v NSO-vol:/nso cisco-nso-prod:6.4`

---

At this point, you only have one container that is running the desired version 6.4 and you do not need to uninstall the old NSO.

## Running the NSO Images using Docker Compose

This example covers the necessary information to manifest the use of NSO images to compile packages and run NSO. Using Docker Compose is not a requirement, but a simple tool for defining and running a multi-container setup where you want to run both the Production and Development images in an efficient manner.

### Packages

Packages used in this example are taken from the `examples.ncs/development-guide/nano-services/netsim-sshkey` example:

- `distkey`: A simple Python + template service package that automates the setup of SSH public key authentication between netsim (ConfD) devices and NSO using a nano service.
- `ne`: A NETCONF NED package representing a netsim network element that implements a configuration subscriber Python application that adds or removes the configured public key, which the netsim (ConfD) network element checks when authenticating public key authentication clients.

### docker-compose.yml - Docker Compose File Example

A basic Docker Compose file is shown in the example below. It describes the containers running on a machine:

- The Production container runs NSO.
- The Development container builds the NSO packages.
- A third `example` container runs the netsim device.

Note that the packages use a shared volume in this simple example setup. In a more complex production environment, you may want to consider a dedicated redundant volume for your packages.

```
version: '1.0'
```

```

volumes:
 NSO-1-rvol:

networks:
 NSO-1-net:

services:
 NSO-1:
 image: cisco-nso-prod:6.4
 container_name: nsol
 profiles:
 - prod
 environment:
 - EXTRA_ARGS=--with-package-reload
 - ADMIN_USERNAME=admin
 - ADMIN_PASSWORD=admin
 networks:
 - NSO-1-net
 ports:
 - "2024:2024"
 - "8888:8888"
 volumes:
 - type: bind
 source: /path/to/packages/NSO-1
 target: /nso/run/packages
 - type: bind
 source: /path/to/log/NSO-1
 target: /log
 - type: volume
 source: NSO-1-rvol
 target: /nso
 healthcheck:
 test: ncs_cmd -c "wait-start 2"
 interval: 5s
 retries: 5
 start_period: 10s
 timeout: 10s

 BUILD-NSO-PKGS:
 image: cisco-nso-dev:6.4
 container_name: build-nso-pkgs
 network_mode: none
 profiles:
 - dev
 volumes:
 - type: bind
 source: /path/to/packages/NSO-1
 target: /nso/run/packages

 EXAMPLE:
 image: cisco-nso-prod:6.4
 container_name: ex-netsim
 profiles:
 - example
 networks:
 - NSO-1-net
 healthcheck:
 test: test -f /nso-run-prod/etc/ncs.conf && ncs-netsim --dir /netsim is
 interval: 5s
 retries: 5
 start_period: 10s
 timeout: 10s

```

```

entrypoint: bash
command: -c 'rm -rf /netsim
&& mkdir /netsim
&& ncs-netsim --dir /netsim create-network /network-element 1 ex
&& PYTHONPATH=/opt/ncs/current/src/ncs/pyapi ncs-netsim --dir
/netsim start
&& mkdir -p /nso-run-prod/run/cdb
&& echo "<devices xmlns=\"http://tail-f.com/ns/ncs\">
<authgroups><group><name>default</name>
<umap><local-user>admin</local-user>
<remote-name>admin</remote-name><remote-password>
admin</remote-password></umap></group>
</authgroups></devices>"
> /nso-run-prod/run/cdb/init1.xml
&& ncs-netsim --dir /netsim ncs-xml-init >
/nso-run-prod/run/cdb/init2.xml
&& sed -i.orig -e "s|127.0.0.1|ex-netsim|"
/nso-run-prod/run/cdb/init2.xml
&& mkdir -p /nso-run-prod/etc
&& sed -i.orig -e "s|</cli>|<style>c</style>
</cli>|" -e "/<ssh>/{n;s|<enabled>false
</enabled>|
<enabled>true</enabled>|}" defaults/ncs.conf
&& sed -i.bak -e "/<local-authentication>/{n;s|
<enabled>false</enabled>|<enabled>true
</enabled>|}" defaults/ncs.conf
&& sed "/<ssl>/{n;s|<enabled>false</enabled>|
<enabled>true</enabled>|}" defaults/ncs.conf
> /nso-run-prod/etc/ncs.conf
&& mv defaults/ncs.conf.orig defaults/ncs.conf
&& tail -f /dev/null'
volumes:
- type: bind
source: /path/to/packages/NSO-1/ne
target: /network-element
- type: volume
source: NSO-1-rvol
target: /nso-run-prod

```

## Explanation of Docker Compose File

A description of noteworthy Compose file items is given below.

- **profiles:** Profiles can be used to group containers in a Compose file, and they work perfectly for the Production, Development, and netsim containers. By adding multiple containers on the same machine (as a developer normally would), you can easily start the Production, Development, and netsim containers using their respective profiles (prod, dev, and example).
- **Command used in netsim example:** Creates a directory called /netsim where the netsims will be set up, then starts the netsims, followed by generating two init.xml files and editing the ncs.conf file for the Production container. Finally, it keeps the container running. If you want this to be more elegant, you need a netsim container image with a script in it that is well-documented.
- **volumes:** The Production and Development images are configured intentionally to have the same bind mount with /path/to/packages/NSO-1 as the source and /nso/run/packages as the target. The Production Image mounts both the /log and /nso directories in the container. The /log directory is simply a bind mount, while the /nso directory is an actual volume.  
Named volumes are recommended over bind mounts as described by the Docker Volumes documentation. The NSO /run directory should therefore be mounted as a named volume. However, you can make the /run directory a bind mount as well.

The Compose file, typically named `docker-compose.yaml`, declares a volume called `NSO-1-rvol`. This is a named volume and will be created automatically by Compose. You can create this volume externally, at which point this volume must be declared as external. If the external volume doesn't exist, the container will not start.

The example `netnsim` container will mount the network element `NED` in the `packages` directory. This package should be compiled. Note that the `NSO-1-rvol` volume is used by the example container to share the generated `init.xml` and `ncs.conf` files with the NSO Production container.

- **healthcheck:** The image comes with its own health check (similar to the one shown here in Compose), and this is how you configure it yourself. The health check for the `netnsim` example container checks if the `ncs.conf` file has been generated, and the first `Netsim` instance started in the container. You could, in theory, start more `netnsims` inside the container.

## Steps

Follow the steps below to run the images using Docker Compose:

**Step 1** Start the Development container. This starts the services in the Compose file with the profile `dev`.

```
docker compose --profile dev up -d
```

**Step 2** Copy the packages from the `netnsim-sshkey` example and compile them in the NSO Development container. The easiest way to do this is by using the **docker exec** command, which gives more control over what to build and the order of it. You can also do this with a script to make it easier and less verbose. Normally you populate the package directory from the host. Here, we use the packages from an example.

```
docker exec -it build-nso-pkgs sh -c 'cp -r ${NCS_DIR}/examples.ncs/development-guide \
 /nano-services/netnsim-sshkey/packages ${NCS_RUN_DIR}'
```

```
docker exec -it build-nso-pkgs sh -c 'for f in ${NCS_RUN_DIR}/packages/*/src; \
 do make -C "$f" all || exit 1; done'
```

**Step 3** Start the `netnsim` container. This outputs the generated `init.xml` and `ncs.conf` files to the NSO Production container. The `--wait` flag instructs to wait until the health check returns healthy.

```
docker compose --profile example up --wait
```

**Step 4** Start the NSO Production container.

```
docker compose --profile prod up --wait
```

At this point, NSO is ready to run the service example to configure the `netnsim` device(s). A bash script (`demo.sh`) that runs the above steps and showcases the **netnsim-sshkey** example is given below:

```
#!/bin/bash
set -eu # Abort the script if a command returns with a non-zero exit code or if
 # a variable name is dereferenced when the variable hasn't been set
GREEN='\033[0;32m'
PURPLE='\033[0;35m'
NC='\033[0m' # No Color

printf "${GREEN}##### Reset the container setup\n${NC}";
docker compose --profile dev down
docker compose --profile example down -v
docker compose --profile prod down -v
```

```

rm -rf ./packages/NSO-1/* ./log/NSO-1/*

printf "${GREEN}##### Start the dev container used for building the NSO NED
and service packages\n${NC}"
docker compose --profile dev up -d

printf "${GREEN}##### Get the packages\n${NC}"
printf "${PURPLE}##### NOTE: Normally you populate the package directory from the host.
Here, we use packages from an NSO example\n${NC}"
docker exec -it build-nso-pkgs sh -c 'cp -r
${NCS_DIR}/examples.ncs/development-guide/nano-services/netsim-sshkey/packages ${NCS_RUN_DIR}'

printf "${GREEN}##### Build the packages\n${NC}"
docker exec -it build-nso-pkgs sh -c 'for f in ${NCS_RUN_DIR}/packages/*/src;
do make -C "$f" all || exit 1; done'

printf "${GREEN}##### Start the simulated device container and setup the example\n${NC}"
docker compose --profile example up --wait

printf "${GREEN}##### Start the NSO prod container\n${NC}"
docker compose --profile prod up --wait

printf "${GREEN}##### Showcase the netsim-sshkey example from NSO on the prod container\n${NC}"
if [[$# -eq 0]] ; then # Ask for input only if no argument was passed to this script
 printf "${PURPLE}##### Press any key to continue or ctrl-c to exit\n${NC}"
 read -n 1 -s -r
fi
docker exec -it nsol sh -c 'sed -i.orig -e "s/make/#make/"
${NCS_DIR}/examples.ncs/development-guide/nano-services/netsim-sshkey/showcase.sh'
docker exec -it nsol sh -c 'cd ${NCS_RUN_DIR};
${NCS_DIR}/examples.ncs/development-guide/nano-services/netsim-sshkey/showcase.sh 1'

```

## Upgrading NSO using Docker Compose

This example describes how to upgrade NSO when using Docker Compose.

### Upgrade to a New Minor or Major Version

To upgrade to a new minor or major version, for example, from 6.3 to 6.4, follow the steps below:

- Step 1** Change the image version in the Compose file to the new version, i.e., 6.4.
- Step 2** Run the **docker compose up --profile dev -d** command to start up the Development container with the new image.
- Step 3** Compile the packages using the Development container.

```

docker exec -it build-nso-pkgs sh -c 'for f in
${NCS_RUN_DIR}/packages/*/src;do make -C "$f" all || exit 1; done'

```

- Step 4** Run the **docker compose up --profile prod --wait** command to start the Production container with the new packages that were just compiled.

### Upgrade to a New Maintenance Release Version

To upgrade to a new maintenance release version, for example, to 6.4.1, follow the steps below:

- Step 1** Change the image version in the Compose file to the new version, i.e., 6.4.1.



**Step 2**

Run the **docker compose up --profile prod --wait** command.

Upgrading in this way does not require a recompile. Docker detects changes and upgrades the image in the container to the new version.

---





## CHAPTER 14

# NED Administration

---

- [Introduction, page 193](#)
- [Types of NED Packages, page 193](#)
- [Purpose of NED ID, page 197](#)
- [NED Installation in NSO, page 199](#)
- [Configuring a device with the new Cisco-provided NED, page 201](#)
- [Managing Cisco-provided third Party YANG NEDs, page 203](#)
- [NED Migration, page 211](#)
- [Revision Merge Functionality, page 213](#)

## Introduction

This guide provides necessary information on NED (Network Element Driver) administration with a focus on Cisco-provided NEDs. If you're planning to use NEDs not provided by Cisco, refer to the Chapter 20, *NED Development in Development Guide* to build your own NED packages.

NED represents a key NSO component that makes it possible for NSO core system to communicate southbound with network devices in most deployments. NSO has a built-in client that can be used to communicate southbound with NETCONF-enabled devices. Many network devices are, however, not NETCONF-enabled, and there exist a wide variety of methods and protocols for configuring network devices, ranging from simple CLI to HTTP/REST-enabled devices. For such cases, it is necessary to use a NED to allow NSO communicate southbound with the network device.

Even for NETCONF-enabled devices, it is possible that the NSO's built-in NETCONF client cannot be used, for instance, if the devices do not strictly follow the specification for the NETCONF protocol. In such cases, one must also use a NED to seamlessly communicate with the device. See [the section called “Managing Cisco-provided third Party YANG NEDs”](#) for more information on third party YANG NEDs.

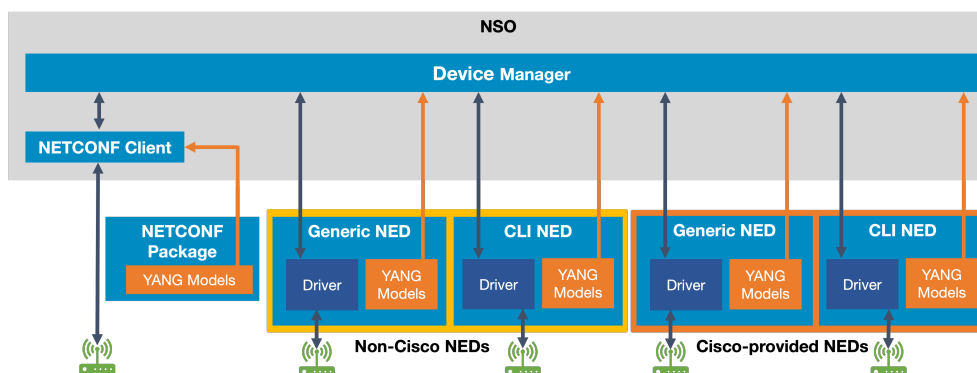
## Types of NED Packages

A NED package is a package that NSO uses to manage a particular type of device. A NED is a piece of code that enables communication with a particular type of managed device. You add NEDs to NSO as a special kind of package, called NED packages.

A NED package must provide a device YANG model as well as define means (protocol) to communicate with the device. The latter can either leverage the NSO built-in NETCONF and SNMP support, or use a

custom implementation. When a package provides custom protocol implementation, typically written in Java, it is called a CLI NED or a Generic NED.

Cisco provides and supports a number of such NEDs. With these Cisco-provided NEDs, a major category are CLI NEDs which communicate with a device through its CLI instead of a dedicated API.



NED Package Types

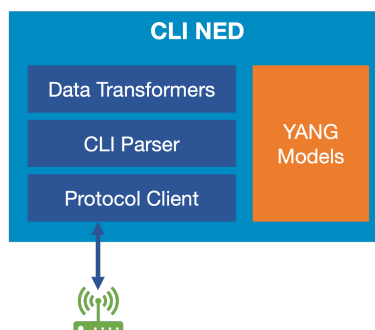
## CLI NED

This NED category is targeted at devices that use CLI as a configuration interface. Cisco-provided CLI NEDs are available for various network devices from different vendors. Many different CLI syntaxes are supported.

The driver element in a CLI NED implemented by the Cisco NSO NED team typically consists of the following three parts:

- The protocol client, responsible for connecting to and interacting with the device. The protocols supported are SSH and Telnet.
- A fast and versatile CLI parser (+ emitter), usually referred to as the turbo parser.
- Various transform engines capable of converting data between NSO and device formats.

The YANG models in a CLI NED are developed and maintained by the Cisco NSO NED team. Usually, the models for a CLI NED are structured to mimic the CLI command hierarchy on the device.



CLI NED

## Generic NED

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 in many cases require a generic NED to function properly with NSO.

The driver element in a Generic NED implemented by the Cisco NED team typically consists of the following parts:

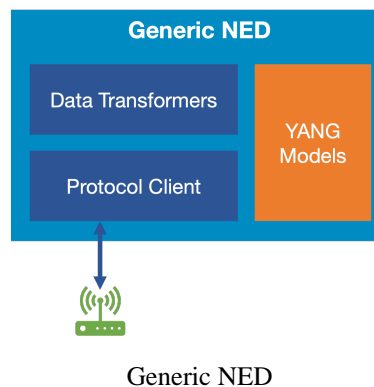
- The protocol client, responsible for interacting with the device.
- Various transform engines capable of converting data between NSO and the device formats, usually JSON and/or XML transformers.

There are two types of Generic NEDs maintained by the Cisco NSO NED team:

- NEDs with Cisco-owned YANG models. These NEDs have models developed and maintained by the Cisco NSO NED team.
- NEDs targeted at YANG models from third party vendors, also known as, third party YANG NEDs.

### Generic Cisco-provided NEDs with Cisco-owned YANG Models

Generic NEDs belonging to the first category typically handle devices that are model-driven. For instance, devices using proprietary protocols based on REST, SOAP, Corba, etc. The YANG models for such NEDs are usually structured to mimic the messages used by the proprietary protocol of the device.

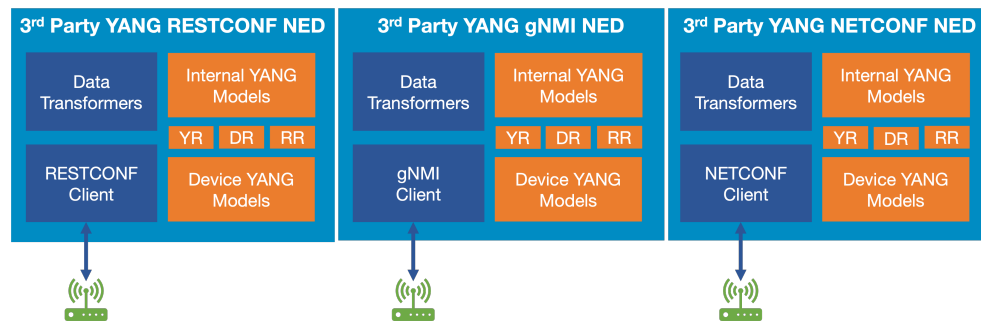


### Third party YANG NEDs

As the name implies, this NED category is used for cases where the device YANG models are not implemented, maintained, or owned by the Cisco NSO NED team. Instead, the YANG models are typically provided by the device vendor itself, or by organizations like IETF, IEEE, ONF, or OpenConfig.

This category of NEDs has some special characteristics that set them apart from all other NEDs developed by the Cisco NSO NED team:

- Targeted for devices supporting model-driven protocols like NETCONF, RESTCONF, and gNMI.
- Delivered from the software.cisco.com portal without any device YANG models included. There are several reasons for this, such as legal restrictions that prevent Cisco from re-distributing YANG models from other vendors, or availability of several different version bundles for open source YANG, like OpenConfig. The version used by the NED must match the version used by the targeted device.
- The NEDs can be bundled with various fixes to solve shortcomings in the YANG models, the download sources, and/or in the device. These fixes are referred to as recipes.



Third Party YANG NEDs

Since the third party NEDs are delivered without any device YANG models, there are additional steps required to make this category of NEDs operational:

- 1 The device models need to be downloaded and copied into the NED package source tree. This can be done by using a special (optional) downloader tool bundled with each third party YANG NED, or in any custom way.
- 2 The NED must be rebuilt with the downloaded YANG models.

This procedure is thoroughly described in [the section called “Managing Cisco-provided third Party YANG NEDs”](#).

## Recipes

A third party YANG NED can be bundled with up to three types of recipe modules. These recipes are used by the NED to solve various types of issues related to:

- The source of the YANG files.
- The YANG files.
- The device itself.

The recipes represent the characteristics and the real value of a third party YANG NED. Recipes are typically adapted for a certain bundle of YANG models and/or certain device types. This is why there exist many different third party YANG NEDs, each one adapted for a specific protocol, a specific model package, and/or a specific device.



### Note

The NSO NED team does not provide any super third party YANG NEDs, for instance, a super RESTCONF NED that can be used with any models and any device.

## Download Recipes

When downloading the YANG files, it is first of all important to know which source to use. In some cases, the source is the device itself. For instance, if the device is enabled for NETCONF and sometimes for RESTCONF (in rare cases).

In other cases, the device does not support model download. This applies to all gNMI-enabled devices and most RESTCONF devices too. In this case, the source can be a public Git repository or an archive file provided by the device vendor.

Another important question is what YANG models and what versions to download. To make this task easier, third party NEDs can be bundled with the download recipes. These are presets to be used with the downloader tool bundled with the NED. There can be several profiles, each representing a preset that has

been verified to work by the Cisco NSO NED team. A profile can point out a certain source to download from. It can also limit the scope of the download so that only certain YANG files are selected.

### YANG Recipes (YR)

Third party YANG files can often contain various types of errors, ranging from real bugs that cause compilation errors to certain YANG constructs that are known to cause runtime issues in NSO. To ensure that the files can be built correctly, the third-party NEDs can be bundled with YANG recipes. These recipes patch the downloaded YANG files before they are built by the NSO compiler. This procedure is performed automatically by the make system when the NED is rebuilt after downloading the device YANG files. For more information, refer to [the section called “Rebuilding the NED with a Unique NED ID”](#).

### Runtime Recipes (RR)

Many devices enabled for NETCONF, RESTCONF, or gNMI sometimes deviate in their runtime behavior. This can make it impossible to interact properly with NSO. These deviations can be on any level in the runtime behavior, such as:

- The configuration protocol is not properly implemented, i.e., the device lacks support for mandatory parts of, for instance, the RESTCONF RFC.
- The device returns "dirty" configuration dumps, for instance, JSON or XML containing invalid elements.
- Special quirks are required when applying new configuration on a device. May also require additional transforms of the payload before it is relayed by the NED.
- The device has aliasing issues, possibly caused by overlapping YANG models. If leaf X in model A is modified, the device will automatically modify leaf Y in model B as well.

A third party YANG NED can be bundled with runtime recipes to solve these kinds of issues, if necessary. How this is implemented varies from NED to NED. In some cases, a NED has a fixed set of recipes that are always used. Alternatively, a NED can support several different recipes, which can be configured through a NED setting, referred to as a runtime profile. For example, a multi-vendor third party YANG NED might have one runtime profile for each device type supported:

```
admin@ncs(config)# devices device dev-1 ned-settings
onf-tapi_rc restconf profile vendor-xyz
```

## NED Settings

NED settings are YANG models augmented as configurations in NSO and control the behavior of the NED. These settings are augmented under:

- /devices/global-settings/ned-settings
- /devices/profiles/ned-settings
- /devices/device/ned-settings

Most NEDs are instrumented with a large number of NED settings that can be used to customize the device instance configured in NSO. The README file in the respective NED contains more information on these.

## Purpose of NED ID

Each managed device in NSO has a device type that informs NSO how to communicate with the device. When managing NEDs, the device type is either `cli` or `generic`. The other two device types, `netconf` and `snmp`, are used in NETCONF and SNMP packages and are further described in this guide.

In addition, a special NED ID identifier is needed. Simply put, this identifier is a handle in NSO pointing to the NED package. NSO uses the identifier when it is about to invoke the driver in a NED package. The identifier ensures that the driver of the correct NED package is called for a given device instance. For more information on how to set up a new device instance, see [the section called “Configuring a device with the new Cisco-provided NED”](#).

Each NED package has a NED ID, which is mandatory. The NED ID is a simple string that can have any format. For NEDs developed by the Cisco NSO NED team, the NED ID is formatted as `<NED NAME>-<gen | cli>-<NED VERSION MAJOR>.<NED VERSION MINOR>`.

### Examples

- `onf-tapi_rc-gen-2.0`
- `cisco-iosxr-cli-7.43`

The NED ID for a certain NED package stays the same from one version to another, as long as no backwards incompatible changes have been done to the YANG models. Upgrading a NED from one version to another, where the NED ID is the same, is simple as it only requires replacing the old NED package with the new one in NSO and then reloading all packages.

Upgrading a NED package from one version to another, where the NED ID is not the same (typically indicated by a change of major or minor number in the NED version), requires additional steps. The new NED package first needs to be installed side-by-side with the old one. Then, a NED migration needs to be performed. This procedure is thoroughly described in the [the section called “NED Migration”](#).

The Cisco NSO NED team ensures that our CLI NEDs, as well as Generic NEDs with Cisco-owned models, have version numbers and NED ID that indicate any possible backwards incompatible YANG model changes. When a NED with such an incompatible change is released, the minor digit in the version is always incremented. The case is a bit different for our third party YANG NEDs, since it is up to the end user to select the NED ID to be used. This is further described in [the section called “Managing Cisco-provided third Party YANG NEDs”](#).

## NED Versioning Scheme

A NED is assigned a version number consisting of a sequence of numbers separated by dots. The first two numbers represent the major and minor version, and the third number represents the maintenance version.

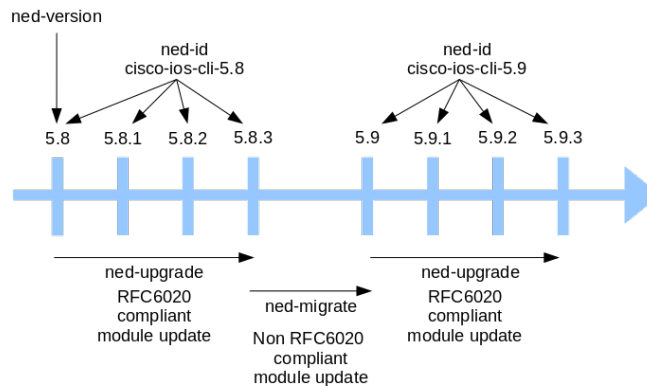
For example, the number 5.8.1 indicates a maintenance release (1) for the minor release 5.8. Incompatible YANG model changes require either the major or minor version number to be changed. This means that any version within the 5.8.x series is backward compatible with the previous versions.

When a newer maintenance release with the same major/minor version replaces a NED release, NSO can perform a simple data model upgrade to handle stored instance data in the CDB (Configuration Database). This type of upgrade does not pose a risk of data loss.

However, when a NED is replaced by a new major/minor release, it becomes a NED migration. These migrations are complex because the YANG model changes can potentially result in the loss of instance data if not handled correctly.



Figure 23. NED Version Scheme



## NED Installation in NSO

This section describes the NED installation in NSO for Local and System installs. Consult the `README.md` supplied with the NED for most up-to-date installation description.

### Local Install of NED in NSO

This section describes how to install a NED package on a locally installed NSO. See the section called “Local Install Steps” in *Getting Started* for more information.

Follow the instructions below to install a NED package:

**Step 1** Download the latest production-grade version of the NED from [software.cisco.com](http://software.cisco.com) using the URLs provided on your NED license certificates. All NED packages are files with the `.signed.bin` extension named using the following rule: `ncs-<NSO VERSION>-<NED NAME>-<NED VERSION>.signed.bin`

The NED package `ncs-6.0-cisco-iosxr-7.43.signed.bin` will be used in the example below. It is assumed the NED package has been downloaded into the directory named `/tmp/ned-package-store`. The environment variable `NSO_RUNDIR` needs to be configured to point to the NSO runtime directory. Example:

```
> export NSO_RUNDIR=~/.nso-lab-rundir
```

**Step 2** Unpack the NED package and verify its signature.

```
> cd /tmp/ned-package-store
> chmod u+x ncs-6.0-cisco-iosxr-7.43.signed.bin
> ./ncs-6.0-cisco-iosxr-7.43.signed.bin
```



#### Note

In case the signature cannot be verified (for instance, if access to internet is down), do as below instead:

```
> ./ncs-6.0-cisco-iosxr-7.43.signed.bin --skip-verification
```

The result of the unpacking is a `tar.gz` file with the same name as the `.bin` file.

```
> ls *.tar.gz
```

**Step 3** Untar the `tar.gz` file. The result is a subdirectory named like `<NED NAME>-<NED MAJOR VERSION DIGIT>.<NED MINOR VERSION DIGIT>`

```
ncs-6.0-cisco-iosxr-7.43.tar.gz
> tar xzf ncs-6.0-cisco-iosxr-7.43.tar.gz
> ls -d */
cisco-iosxr-7.43
```

**Step 4** Install the NED into NSO, using the `ncs-setup` tool.

```
> ncs-setup --package cisco-iosxr-7.43 --dest $NSO_RUNDIR
```

**Step 5** Finally, open an NSO CLI session and load the new NED package like below:

```
> ncs_cli -C -u admin
admin@ncs# packages reload
reload-result {
package cisco-iosxr-cli-7.43
result true
}
```

---

Alternatively, the `tar.gz` file can be installed directly into NSO. In this case, skip steps 3 and 4, and do as below instead:

```
> ncs-setup --package cisco-iosxr-7.43.tar.gz --dest $NSO_RUNDIR
```

## System Install of Cisco-provided NED in NSO

This section describes how to install a NED package on a system installed NSO. See the section called “System Install Steps” in *Getting Started* for more information.

**Step 1** Download the latest production-grade version of the NED from [software.cisco.com](http://software.cisco.com) using the URLs provided on your NED license certificates. All NED packages are files with the `.signed.bin` extension named using the following rule: `ncs-<NSO_VERSION>-<NED NAME>-<NED VERSION>.signed.bin`.

The NED package `ncs-6.0-cisco-iosxr-7.43.signed.bin` will be used in the example below. It is assumed that the package has been downloaded into the directory named `/tmp/ned-package-store`.

**Step 2** Unpack the NED package and verify its signature.

```
> cd /tmp/ned-package-store
> chmod u+x ncs-6.0-cisco-iosxr-7.43.signed.bin
> ./ncs-6.0-cisco-iosxr-7.43.signed.bin
```



### Note

In case the signature cannot be verified (for instance, if access to internet is down), do as below instead.

```
> ./ncs-6.0-cisco-iosxr-7.43.signed.bin --skip-verification
```

---

The result of the unpacking is a `tar.gz` file with the same name as the `.bin` file.

```
> ls *.tar.gz
ncs-6.0-cisco-iosxr-7.43.tar.gz
```

**Step 3** Perform an NSO backup before installing the new NED package.

```
> $NCS_DIR/bin/ncs-backup
```

**Step 4** Start an NSO CLI session.

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

**Step 5** Fetch the NED package.

```
admin@ncs# software packages fetch package-from-file
/tmp/ned-package-store/ncs-6.0-cisco-iosxr-7.43.tar.gz
admin@ncs# software packages list
package {
 name ncs-6.0-cisco-iosxr-7.43.tar.gz
 installable
}
```

**Step 6** Install the NED package (add the argument **replace-existing** if a previous version has been loaded).

```
admin@ncs# software packages install cisco-iosxr-7.43
admin@ncs# software packages list
package {
 name ncs-6.0-cisco-iosxr-7.43.tar.gz
 installed
}
```

**Step 7** Finally, load the NED package.

```
admin@ncs# packages reload
admin@ncs# software packages list
package {
 name cisco-iosxr-cli-7.43
 loaded
}
```

---

## Configuring a device with the new Cisco-provided NED

The basic steps for configuring a device instance using the newly installed NED package are described in this section. Only the most basic configuration steps are covered here.

Many NEDs require additional custom configuration to be operational. This applies in particular to Generic NEDs. Information about such additional configuration can be found in the files `README.md` and `README-ned-settings.md` bundled with the NED package.

The following info is necessary to proceed with the basic setup of a device instance in NSO:

- NED ID of the new NED.
- Connection information for the device to connect to (address and port).
- Authentication information to the device (username and password).

### CLI NED Setup

For CLI NEDs, it is mandatory to specify the protocol to be used, either SSH or Telnet.

The following values will be used for this example:

- NED ID: `cisco-iosxr-cli-7.43`
- Address: `10.10.1.1`
- Port: `22`
- Protocol: `ssh`
- User: `cisco`
- Password: `cisco`

Do the CLI NED setup as below:

- 
- Step 1** Start an NSO CLI session.
- ```
> ncs_cli -C -u admin
```
- Step 2** Enter the configuration mode.
- ```
admin@ncs# configure
Entering configuration mode terminal
admin@ncs(config)#
```
- Step 3** Configure a new authentication group to be used for this device.
- ```
admin@ncs(config)# devices authgroup my-xrgroup default-map
remote-name cisco remote-password cisco
```
- Step 4** Configure the new device instance.
- ```
admin@ncs(config)# devices device xrdev-1 address 10.10.1.1
admin@ncs(config)# devices device xrdev-1 port 22
admin@ncs(config)# devices device xrdev-1 device-type cli ned-id cisco-iosxr-cli-7.43 protocol ssh
admin@ncs(config)# devices device xrdev-1 state admin-state unlocked
admin@ncs(config)# devices device xrdev-1 authgroup my-xrgroup
```
- Step 5** Next, check the README.md and README-ned-settings.md bundled with the NED package for further information on additional settings to make the NED fully operational.
- Step 6** Finally commit the configuration.
- ```
admin@ncs(config)# commit
```
- In case of SSH, run also:
- ```
admin@ncs(config)# devices device xrdev-1 ssh fetch-host-keys
```
- 

## Cisco-provided Generic NED Setup

This example shows a simple setup of a generic NED.

The following values will be used for this example:

- NED ID: onf-tapi\_rc-gen-2.0
- Address: 10.10.1.2
- Port: 443
- User: admin
- Password: admin

Do the Generic NED setup as below:

- 
- Step 1** Start an NSO CLI session.
- ```
> ncs_cli -C -u admin
```
- Step 2** Enter the configuration mode.
- ```
admin@ncs# configure
Entering configuration mode terminal
admin@ncs(config)#
```
- Step 3** Configure a new authentication group to be used for this device.
- ```
admin@ncs(config)# devices authgroup my-tapigroup default-map remote-name admin
remote-password admin
```

Step 4 Configure the new device instance.

```
admin@ncs(config)# devices device tapidev-1 address 10.10.1.2
admin@ncs(config)# devices device tapidev-1 port 443
admin@ncs(config)# devices device tapidev-1 device-type generic ned-id onf-tapi_rc-gen-2.0
admin@ncs(config)# devices device tapidev-1 state admin-state unlocked
admin@ncs(config)# devices device tapidev-1 authgroup my-tapigroup
```

Step 5 Next, check the README.md and README-ned-settings.md bundled with the NED package for further information on additional settings to make the NED fully operational.

Step 6 Finally commit the configuration.

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

Managing Cisco-provided third Party YANG NEDs

The third party YANG NED type is a special category of the generic NED type targeted for devices supporting protocols like NETCONF, RESTCONF, and gNMI. As the name implies, this NED category is used for cases where the device YANG models are not implemented or maintained by the Cisco NSO NED Team. Instead, the YANG models are typically provided by the device vendor itself or by organizations like IETF, IEEE, ONF, or OpenConfig.

A third party YANG NED package is delivered from the software.cisco.com portal without any device YANG models included. It is required that the models are first downloaded, followed by a rebuild and reload of the package, before the NED can become fully operational. This task needs to be performed by the NED user.

Downloading with the NED Built-in Download Tool

This section gives a brief instruction on how to download the device YANG models using the special downloader tool that is bundled with each third party YANG NED. Each specific NED can contain specific requirements regarding downloading/rebuilding. Before proceeding, check the file README-rebuild.md bundled with the NED package. Furthermore, it is recommended to use a non-production NSO environment for this task.

Step 1 Download and install the third party YANG NED package into NSO, see [the section called “Local Install of NED in NSO”](#).

Step 2 Configure a device instance using as usual. See [the section called “Cisco-provided Generic NED Setup”](#) for more information. The device name dev-1 will be used in this example.

Step 3 Open an NCS CLI session (non-configure mode).

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

Step 4 The installed NED is now basically empty. It contains no YANG models except some used by the NED internally. This can be verified with the following CLI commands:

```
admin@ncs# devices device dev-1 connect
result true
info (admin) Connected to dev-1 - 127.0.0.1:7888
admin@ncs# show devices device dev-1 module
NAME                                REVISION    FEATURE    DEVIATION
-----
ietf-restconf-monitoring           2017-01-26  -          -
tailf-internal-rpcs                2022-07-08  -          -
tailf-ned-onf-tapi_rc-stats        2022-10-17  -          -
```

Step 5 The built-in downloader tool consists of a couple of NSO RPCs defined in one of the NED internal YANG files.

Step 6

```
admin@ncs# devices device dev-1 rpc ?
```

```
Possible completions:
```

```
rpc-get-modules  rpc-list-modules  rpc-list-profiles  rpc-show-default-local-dir
```

Start with checking the default local directory. This directory will be used as a target for the device YANG models to be downloaded.

```
admin@ncs# devices device dev-1 rpc rpc-show-default-local-dir show-default-local-dir
result /nso-lab-rundir/packages/onf-tapi_rc-2.0/src/yang
admin@ncs#
```

**Note**

This RPC will throw an error if the NED package was installed directly using the tar.gz file. See [the section called “NED Installation in NSO”](#) for more information.

```
admin@ncs# devices device dev-1 rpc rpc-show-default-local-dir show-default-local-dir
Error: External error in the NED implementation for device nokia-srlinux-1: default
local directory does not exist (/nso-lab-rundir/packages/onf-tapi_rc-2.0/src/yang)
admin@ncs#
```

If this error occurs, it is necessary to unpack the NED package in some other directory and use that as target for the download. In the example below it is /tmp/ned-package-store/onf-tapi_rc-2.0/src/yang.

```
> cd /tmp/ned-package-store
> chmod u+x ncs-6.0-onf-tapi_rc-2.0.3.signed.bin
> ./ncs-6.0-onf-tapi_rc-2.0.3.signed.bin
> tar xzf ncs-6.0-onf-tapi_rc-2.0.3.tar.gz
> ls -ld */
onf-tapi_rc-2.0
```

Step 7

Continue with listing the models supported by the connected device.

```
admin@ncs# devices device netsim-0 rpc rpc-list-modules list-modules
module {
  name tapi-common
  revision 2020-04-23
  namespace urn:onf:otcc:yang:tapi-common
  schema https://localhost:7888/restconf/tailf/modules/tapi-common/2020-04-23
}
module {
  name tapi-connectivity
  revision 2020-06-16
  namespace urn:onf:otcc:yang:tapi-connectivity
  schema https://localhost:7888/restconf/tailf/modules/tapi-connectivity/2020-06-16
}
module {
  name tapi-dsr
  revision 2020-04-23
  namespace urn:onf:otcc:yang:tapi-dsr
  schema https://localhost:7888/restconf/tailf/modules/tapi-dsr/2020-04-23
}
module {
  name tapi-equipment
  revision 2020-04-23
  namespace urn:onf:otcc:yang:tapi-equipment
  schema https://localhost:7888/restconf/tailf/modules/tapi-equipment/2020-04-23
}
...
```

The size of the displayed list is device-dependent and so is the level of detail in each list entry. The only mandatory field is the name. Furthermore, not all devices are actually capable of advertising the models supported. If the currently connected device lacks this support, it is usually emulated by the NED instead. Check the `README-rebuild.md` for more information regarding this.

Step 8

Next, list the download profiles currently supported by the device.

```
admin@ncs# devices device dev-1 rpc rpc-list-profiles list-profiles
profile {
    name onf-tapi-from-device
    description Download the ONF TAPI YANG models. Download is done directly from device.
}
profile {
    name onf-tapi-from-git
    description Download the ONF TAPI YANG models. Download is done from the ONF TAPI github repo.
}
profile {
    name onf-tapi
    description Download the ONF TAPI YANG models. Download source must be specified explicitly.
}
```

A download profile is a preset for the built-in download tool. Its purpose is to make the download procedure as easy as possible. A profile can, for instance, define a certain source from where the device YANG models will be downloaded. Another usage can be to limit the scope of the YANG files to download. For example, one profile to download the native device models, and another for the OpenConfig models. All download profiles are defined and verified by the Cisco NSO NED team. There is usually at least one profile available, otherwise, check the `README-rebuild.md` bundled in the NED package.

Step 9

Finally, try downloading the YANG models using a profile. In case a non-default local directory is used as a target, it must be explicitly specified.

```
admin@ncs# devices device dev-1 rpc rpc-get-modules get-modules profile
onf-tapi-from-device local-dir /tmp/ned-package-store/onf-tapi_rc-2.0/src/yang
```

In case the default local directory is used, no further arguments are needed.

```
admin@ncs# devices device dev-1 rpc rpc-get-modules get-modules profile onf-tapi-from-device
```

The tool will output a list with each file downloaded. It automatically scans each YANG file for dependencies and tries to download them as well.

```
result
Fetching modules:
tapi-common - urn:onf:otcc:yang:tapi-common (32875 bytes)
tapi-connectivity - urn:onf:otcc:yang:tapi-connectivity (40488 bytes)
    fetching imported module tapi-path-computation
    fetching imported module tapi-topology
tapi-dsr - urn:onf:otcc:yang:tapi-dsr (11172 bytes)
tapi-equipment - urn:onf:otcc:yang:tapi-equipment (33406 bytes)
tapi-eth - urn:onf:otcc:yang:tapi-eth (93152 bytes)
    fetching imported module tapi-oam
tapi-notification - urn:onf:otcc:yang:tapi-notification (23864 bytes)
tapi-oam - urn:onf:otcc:yang:tapi-oam (30409 bytes)
tapi-odu - urn:onf:otcc:yang:tapi-odu (45327 bytes)
tapi-path-computation - urn:onf:otcc:yang:tapi-path-computation (19628 bytes)
tapi-photonic-media - urn:onf:otcc:yang:tapi-photonic-media (52848 bytes)
tapi-topology - urn:onf:otcc:yang:tapi-topology (43357 bytes)
tapi-virtual-network - urn:onf:otcc:yang:tapi-virtual-network (13278 bytes)
fetched and saved 12 yang module(s) to /tmp/ned-package-store/onf-tapi_rc-2.0/src/yang
```

Step 10

Verify that the downloaded files have been stored properly in the configured target directory.

```
> ls -l /tmp/ned-package-store/onf-tapi_rc-2.0/src/yang
total 616
-rw-r--r-- 1 nso-user staff 109607 Nov 11 13:15 tailf-common.yang
-rw-r--r-- 1 nso-user staff 32878 Nov 11 13:15 tapi-common.yang
-rw-r--r-- 1 nso-user staff 40503 Nov 11 13:15 tapi-connectivity.yang
-rw-r--r-- 1 nso-user staff 11172 Nov 11 13:15 tapi-dsr.yang
-rw-r--r-- 1 nso-user staff 33406 Nov 11 13:15 tapi-equipment.yang
-rw-r--r-- 1 nso-user staff 93152 Nov 11 13:15 tapi-eth.yang
-rw-r--r-- 1 nso-user staff 23864 Nov 11 13:15 tapi-notification.yang
-rw-r--r-- 1 nso-user staff 30409 Nov 11 13:15 tapi-oam.yang
-rw-r--r-- 1 nso-user staff 45327 Nov 11 13:15 tapi-odu.yang
-rw-r--r-- 1 nso-user staff 19628 Nov 11 13:15 tapi-path-computation.yang
-rw-r--r-- 1 nso-user staff 52848 Nov 11 13:15 tapi-photonic-media.yang
-rw-r--r-- 1 nso-user staff 43357 Nov 11 13:15 tapi-topology.yang
-rw-r--r-- 1 nso-user staff 13281 Nov 11 13:15 tapi-virtual-network.yang
```

Rebuilding NED with Downloaded YANG Files

The NED must be rebuilt when the device YANG models have been downloaded and stored properly. Compiling third party YANG files is often combined with various types of issues caused by bad or odd YANG constructs. Such issues typically cause compiler errors or unwanted runtime errors in NSO. A third party YANG NED is configured to take care of all currently known build issues. It will automatically patch the problematic files such that they build properly for NSO. This is done using a set of YANG build recipes bundled with the NED package.



Note

Adapting the YANG build recipes is a continuous process. If new issues are found, the Cisco NED team updates the recipes accordingly and releases a new version of the NED.

It is strongly recommended that end users report newly found YANG build issues to the Cisco NSO NED team through a support request.

Before rebuilding the NED, it is important to know the path to the target directory used for the downloaded YANG files. This is the same as the local directory if the built-in NED downloader tool was used, see [the section called “Downloading with the NED Built-in Download Tool”](#).

This example uses the environment variable `NED_YANG_TARGET_DIR` to represent the target directory.

To rebuild the NED with the downloaded YANG file:

Step 1 Enter the NED build directory, which is the parent directory to the target directory.

```
> echo $NED_YANG_TARGET_DIR
/tmp/ned-package-store/onf-tapi_rc-2.0/src/yang
> cd $NED_YANG_TARGET_DIR/..
```

Step 2 Run the **make clean all** command. The output from the **make** command can be massive, depending on the number of YANG files, etc. After this step, the NED is rebuilt with the device YANG models included. Lines like below indicate that the NED has applied a number of YANG recipes (patches) to solve known issues with the YANG files:

```
> make clean all
===== RUNNING YANG PRE-PROCESSOR (YPP) WITH THE FOLLOWING VARIABLES:
tools/ypp --var NCS_VER=6.0 --var NCS_VER_NUMERIC=6000000
--var SUPPORTS_CDM=YES --var SUPPORTS_ROLLBACK_FILES_OCTAL=YES
--var SUPPORTS_SHOW_STATS_PATH=YES \
```



```

\
--from=' NEDCOM_SECRET_TYPE' --to=' string' \
'tmp-yang/*.yang'
touch tmp-yang/ypp_ned

===== REMOVE PRESENCE STATEMENT ON CONTEXT TOP CONTAINER
tools/ypp --from="(presence \"Root container\"" \
--to="//\g<1>" \
'tmp-yang/tapi-common.yang'

===== ADDING EXTRA ENUM WITH CORRECT SPELLING: NO_PROTECTION
tools/jypp --add-stmt=/typedef#protection-type/type::"enum NO_PROTECTION;" \
'tmp-yang/tapi-topology.yang' || true

===== ADDING EXTRA IDENTITIES USED BY CERTAIN TAPI DEVICES
tools/jypp --add-stmt=/::"identity DIGITAL_SIGNAL_TYPE_400GBASE-R { base DIGITAL_SIGNAL_TYPE; }" \
--add-stmt=/::"identity DIGITAL_SIGNAL_TYPE_GigE_CONV { base DIGITAL_SIGNAL_TYPE; }" \
--add-stmt=/::"identity DIGITAL_SIGNAL_TYPE_ETHERNET { base DIGITAL_SIGNAL_TYPE; }" \
'tmp-yang/tapi-dsr.yang' || true

```

Reloading the NED Package into NSO

This is the final step to make a third party YANG NED operational. If the NED built-in YANG downloader tool was used together with no `local-dir` argument specified (i.e., the default), the only thing required is a package reload in NSO, which you can do by running the **packages reload** or the **packages add** command.

```

> 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.
reload-result {
  package onf-tapi_rc-gen-2.0
  result true
}
admin@ncs#

```

If another target directory was used for the YANG file download, it is necessary to first do a proper re-install of the NED package. See [the section called “NED Installation in NSO”](#).

Rebuilding the NED with a Unique NED ID

A common use case is to have many different versions of a certain device type in the network. All devices can be managed by the same third party YANG NED. However, each device will likely have its unique set of YANG files (or versions) which this NED has to be rebuilt for.

To set up NSO for this kind of scenario, some additional steps need to be taken:

- Each flavor of the NED needs to be built in a separate source directory, i.e., untar the third party YANG NED package at multiple locations.
- Each flavor of the re-built NED must have its own unique NED-ID. This will make NSO allow multiple versions of the same NED package to co-exist.

The default NED ID for a third party YANG NED typically looks like: `<NED NAME>-gen-<NED VERSION MAJOR DIGIT>.<NED VERSION MINOR DIGIT>`

The NED build system allows for a customized NED ID by setting one or several of three make variables in any combination when rebuilding the NED:

- NED_ID_SUFFIX
- NED_ID_MAJOR
- NED_ID_MINOR

Do as follows to build each flavor of the third party YANG NED. Do it in iterations, one at a time:

-
- Step 1** Unpack the empty NED package as described in [the section called “NED Installation in NSO”](#).
- Step 2** Unpack the NED package again in a separate location. Rename the NED directory to something unique.
- ```
> cd /tmp/ned-package-store
> chmod u+x ncs-6.0-onf-tapi_rc-2.0.3.signed.bin
> ./ncs-6.0-onf-tapi_rc-2.0.3.signed.bin
> tar xzf ncs-6.0-onf-tapi_rc-2.0.3.tar.gz
> ls -ld */
onf-tapi_rc-2.0
> mv onf-tapi_rc-2.0 onf-tapi_rc-2.0-variant-1
```
- Step 3** Configure a device instance using the installed NED, as described in [the section called “Cisco-provided Generic NED Setup”](#). Configure it to connect to the first variant of the device.
- Step 4** Follow the instructions in [the section called “Downloading with the NED Built-in Download Tool”](#) to download the YANG files. Configure `local-dir` to point to the location configured in [the section called “Rebuilding NED with Downloaded YANG Files”](#).
- ```
> ncs_cli -C -u admin
admin@ncs# devices device dev-1 rpc rpc-get-modules get-modules profile
onf-tapi-from-device local-dir /tmp/ned-package-store/onf-tapi_rc-2.0-variant-1/src/yang
```
- Step 5** Rebuild a NED package from the location configured in [the section called “Rebuilding NED with Downloaded YANG Files”](#). Use a suitable combination of the NED_ID_SUFFIX, NED_ID_MAJOR, NED_ID_MINOR.
- Example 1:
- ```
> make clean all NED_ID_SUFFIX=_tapi_v2.1.3
```
- This will result in the NED ID: `onf-tapi_rc_tapi_v2.1.3-gen-2.0`.
- Example 2:
- ```
> make clean all NED_ID_MAJOR=2 NED_ID_MINOR=1.3
```
- This will result in the NED ID: `onf-tapi_rc-gen-2.1.3`.
- Step 6** Install the newly built NED package into NSO, side-by-side with the original NED package. See [the section called “Configuring a device with the new Cisco-provided NED”](#) for further information.
- Example:
- ```
> cd /tmp/ned-package-store
> tar cfz onf-tapi_rc-2.0-variant-1.tar.gz onf-tapi_rc-2.0-variant-1
> ncs-setup --package onf-tapi_rc-2.0-variant-1.tar.gz --dest $NSO_RUNDIR
> ncs_cli -C -u admin
admin@ncs# packages reload
```
- Step 7** Configure a new device instance using the newly installed NED package. Configure it to connect to the first variant of the device, as done in step 3.
- Step 8** Verify functionality by executing a **sync-from** on the configured device instance.
-

## Upgrading a Cisco-provided Third Party YANG NED to a Newer Version

The NSO procedure to upgrade a NED package to a newer version uses the following approach:

- If there are no backwards incompatible changes in the schemas (YANG models) of respective NEDs, simply replace the old NED with the new one and reload all packages in NSO.
- In case there are backwards incompatible changes present in the schemas, some administration is required: the new NED needs to be installed side-by-side with the old NED, after which a NED migration must be performed to properly update the data in CDB using the new schemas. More information about NED migration is available in [the section called “NED Migration”](#).

Whether or not there are backwards incompatible differences present between two versions of the same NED, is determined by the NED ID. If the versions have the same NED ID, they are fully compatible; otherwise, the NED IDs will differ, typically indicated by the major and/or minor number in the NED ID.

The third party YANG NEDs add some extra complexity to the NED migration feature. This is because the device YANG models are not included in the NED package. It is up to the end user to select the YANG model versions to use and also to configure the NED ID. If the same NED, at a later stage, needs to be upgraded and rebuilt with newer versions of the YANG model, a decision has to be made regarding the NED ID: Is it safe to use the same NED ID or should a new one be used?

Using a unique NED ID for each NED package is always the safe option. It minimizes the risk of data loss during package upgrade, etc. However, in some cases, it might be beneficial to use the same NED ID when upgrading a NED package since it minimizes the administration in NSO, i.e., simply replace the old NED package with the new one without any need of NED migration.

This kind of use case can occur when the firmware is upgraded on an NSO-controlled device. For example, assume that we have an optical device that supports the TAPI YANG models from Open Networking Foundation. Current firmware supports version 2.1.3 of the TAPI bundle. The third party YANG NED `onf-tapi_rc` has been rebuilt accordingly with TAPI version 2.1.3 and the default NED ID `onf-tapi_rc-gen-2.0`. This NED package is installed in NSO and a device instance named `dev-1` is configured using it. Next, the optical device is upgraded with the new firmware that supports the TAPI bundle version 2.3.1 instead. The `onf-tapi_rc` NED needs to be upgraded accordingly. The question is what NED ID to use?

To upgrade a Cisco-provided third party YANG NED to a newer version:

**Step 1** Unpack a fresh copy of the `onf-tapi_rc` NED package.

```
> cd /tmp/ned-package-store
> chmod u+x ncs-6.0-onf-tapi_rc-2.0.3.signed.bin
> ./ncs-6.0-onf-tapi_rc-2.0.3.signed.bin
> tar xzf ncs-6.0-onf-tapi_rc-2.0.3.tar.gz
> ls -ld */
onf-tapi_rc-2.0
> mv onf-tapi_rc-2.0 onf-tapi_rc-2.0-for-new-firmware
```

**Step 2** Download the TAPI models v2.3.1 from the TAPI public Git repository.

```
> ncs_cli -C -u admin
admin@ncs# devices device dev-1 rpc rpc-get-modules get-modules
 profile onf-tapi-from-git remote { git { checkout v2.3.1 } }
 local-dir /tmp/ned-package-store/onf-tapi_rc-2.0-new-firmware/src/yang
```

**Step 3** Rebuild the NED package with a temporary unique NED ID for this rebuild. Any unique NED ID works for this.

```
> cd /tmp/ned-package-store/onf-tapi_rc-2.0-for-new-firmware/src/yang
> make clean all NED_ID_MAJOR=2 NED_ID_MINOR=3.1
```

This will generate the NED ID: onf-tapi\_rc-gen-2.3.1.

#### Step 4

Install the new onf-tapi\_rc NED package into NSO, side by side with the old one.

```
> cd /tmp/ned-package-store
> tar cfz onf-tapi_rc-2.0-variant-1.tar.gz onf-tapi_rc-2.0-variant-1
> ncs-setup --package onf-tapi_rc-2.0-variant-1.tar.gz --dest $NSO_RUNDIR
> 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.
reload-result {
 package onf-tapi_rc-gen-2.0
 result true
}
reload-result {
 package onf-tapi_rc-gen-2.3.1
 result true
}
```

#### Step 5

Now, execute a dry-run of the NSO NED migration feature. This command generates a list of all schema differences found between the two packages, like below:

```
admin@ncs# devices device dev-1 migrate new-ned-id onf-tapi_rc-gen-2.3.1 dry-run

modified-path {
 path /tapi-common:context/tapi-virtual-network:virtual-network-context/
 virtual-nw-service/vnw-constraint/service-layer
 info leaf-list type stack has changed
 backward-compatible false
}
modified-path {
 path /tapi-common:context/tapi-virtual-network:virtual-network-context/
 virtual-nw-service/vnw-constraint/requested-capacity/bandwidth-profile
 info sub-tree has been deleted
 backward-compatible false
}
modified-path {
 path /tapi-common:context/tapi-virtual-network:virtual-network-context/
 virtual-nw-service/vnw-constraint/latency-characteristic/queing-latency-characteristic
 info sub-tree has been deleted
 backward-compatible false
}
modified-path {
 path /tapi-common:context/tapi-virtual-network:virtual-network-context/
 virtual-nw-service/vnw-constraint
 info min/max has been relaxed
 backward-compatible true
}
modified-path {
 path /tapi-common:context/tapi-virtual-network:virtual-network-context/
 virtual-nw-service/vnw-constraint
 info list key has changed; leaf 'local-id' has changed type
 backward-compatible false
}
modified-path {
 path /tapi-common:context/tapi-virtual-network:virtual-network-context/
 virtual-nw-service/layer-protocol-name
 info node is no longer mandatory
}
```

```

 backward-compatible true
 }
 modified-path {
 path /tapi-common:context/tapi-virtual-network:virtual-network-context/
 virtual-nw-service/layer-protocol-name
 info leaf-list type stack has changed
 backward-compatible false
 }
}

```

If the goal is to rebuild the new NED package again using the same NED ID as the old NED package, there are two things to look out for in the list:

- 1 Does the list contain any items with backward-compatible false?
- 2 If the answer is yes, is the affected schema node relevant for any use case, i.e., referenced by any service code running in NSO?

Any item listed as backward-compatible false can potentially result in data loss if the old NED is simply replaced with the new one. This might however be acceptable if the affected schema node is not relevant for any use case.

## NED Migration

If you upgrade a managed device (such as installing a new firmware), the device data model can change in a significant way. If this is the case, you usually need to use a different and newer NED with an updated YANG model.

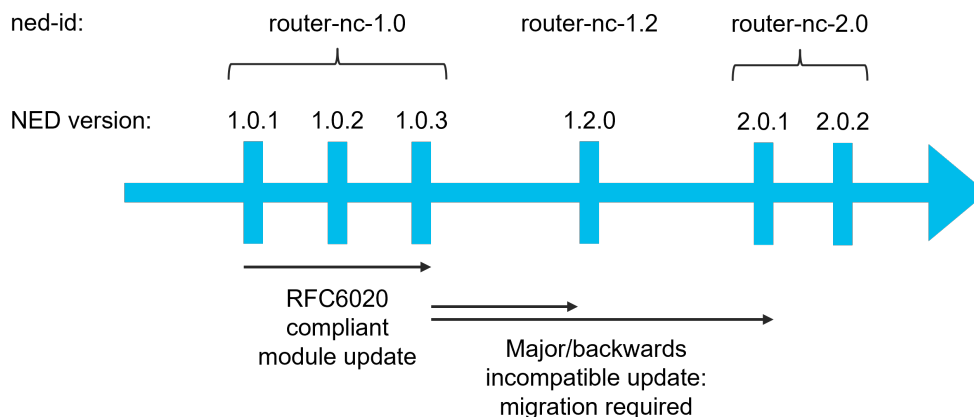
When the changes in the NED are not backwards compatible, the NED is assigned a new ned-id to avoid breaking existing code. On the plus side, this allows you to use both versions of the NED at the same time, so some devices can use the new version and some can use the old one. As a result, there is no need to upgrade all devices at the same time. The downside is, NSO doesn't know the two NEDs are related and will not perform any upgrade on its own due to different ned-ids. Instead, you must manually change the NED of a managed device through a *NED migration*.



### Note

For third party NEDs, the end user is required to configure the NED ID and also be aware of the backward incompatibilities. See [the section called “Upgrading a Cisco-provided Third Party YANG NED to a Newer Version”](#) for an example.

Migration is required when upgrading a NED and the ned-id changes, which is signified by a change in either the first or the second number in the NED package version. For example, if you're upgrading the existing router-nc-1.0.1 NED to router-nc-1.2.0 or router-nc-2.0.2, you must perform NED migration. On the other hand, upgrading to router-nc-1.0.2 or router-nc-1.0.3 retains the same ned-id and you can upgrade the router-1.0.1 package in place, directly replacing it with the new one. However, note that some third party, non-Cisco packages may not adhere to this standard versioning convention. In that case, you must check the ned-id values to see whether migration is needed.

**Figure 24. Sample NED package versioning**

A potential issue with a new NED is that it can break an existing service or other packages that rely on it. To help service developers and operators verify or upgrade the service code, NSO provides additional options of migration tooling for identifying the paths and service instances that may be impacted. Therefore, ensure that all the other packages are compatible with the new NED before you start migrating devices.

To prepare for the NED migration process, first load the new NED package into NSO with either **packages reload** or **packages add** command. Then, use the **show packages** command to verify that both NEDs, the new and the old, are present. Finally, you may perform the migration of devices either one-by-one or multiple at a time.

Depending on your operational policies, this may be done during normal operations and does not strictly require a maintenance window, as the migration only reads from and doesn't write to a network device. Still, it is recommended that you create an NSO backup before proceeding.

Note that changing a ned-id also affects device templates if you use them. To make existing device templates compatible with the new ned-id, you can use the `copy` action. It will copy the configuration used for one ned-id to another, as long as the schema nodes used haven't changed between the versions. The following example demonstrates the `copy` action usage:

```
admin@ncs(config)# devices template acme-ntp ned-id router-nc-1.0
copy ned-id router-nc-1.2
```

For individual devices, use the `/devices/device/migrate` action, with the `new-ned-id` parameter. Without additional options, the command will read and update the device configuration in NSO. As part of this process, NSO migrates all the configuration and service meta-data. Use the `dry-run` option to see what the command would do and `verbose` to list all impacted service instances.

You may also use the `no-networking` option to prevent NSO from generating any southbound traffic towards the device. In this case, only the device configuration in the CDB is used for the migration but then NSO can't know if the device is in sync. Afterward, you must use the **compare-config** or the **sync-from** action to remedy this.

For migrating multiple devices, use the `/devices/migrate` action, which takes the same options. However, with this action, you must also specify the `old-ned-id`, which limits the migration to devices using the old NED. You can further restrict the action with the `device` parameter, selecting only specific devices.

It is possible for a NED migration to fail if the new NED is not entirely backwards compatible with the old one and the device has an active configuration that is incompatible with the new NED version. In

such cases, NSO will produce an error with the YANG constraint that is not satisfied. Here, you must first manually adjust the device configuration to make it compatible with the new NED, and then you can perform the migration as usual.

Depending on what changes are introduced by the migration and how these impact the services, it might be good to **re-deploy** the affected services before removing the old NED package. It is especially recommended in the following cases:

- When the service touches a list key that has changed. As long as the old schema is loaded, NSO is able to perform an upgrade.
- When a namespace that was used by the service has been removed. The service diffset, that is, the recorded configuration changes created by the service, will no longer be valid. The diffset is needed for the correct **get-modifications** output, **deep-check-sync**, and similar operations.

## Revision Merge Functionality

The YANG modeling language supports a notion of a module `revision`. It allows users to distinguish between different versions of a module, so the module can evolve over time. If you wish to use a new revision of a module for a managed device, for example to access new features, you generally need to create a new NED.

When a model evolves quickly and you have many devices that require the use of a lot of different revisions, you will need to maintain a high number of NEDs, which are mostly the same. This can become especially burdensome during NSO version upgrades, when all NEDs may need to be recompiled.

When a YANG module is only updated in a backwards compatible way (following the upgrade rules in RFC6020 or RFC7950), the NSO compiler, **ncsc**, allows you to pack multiple module revisions into the same package. This way, a single NED with multiple device model revisions can be used, instead of multiple NEDs. Based on the capabilities exchange, NSO will then use the correct revision for communication with each device.

However, there is a major downside to this approach. While the exact revision is known for each communication session with the managed device, the device model in NSO does not have that information. For that reason, the device model always uses the latest revision. When pushing configuration to a device that only supports an older revision, NSO silently drops the unsupported parts. This may have surprising results, as the NSO copy can contain configuration that is not really supported on the device. Use the **no-revision-drop** commit parameter when you want to make sure you are not committing config that is not supported by a device.

If you still wish to use this functionality, you can create a NED package with the **ncs-make-package --netconf-ned** command as you would otherwise. But the supplied source YANG directory should contain YANG modules with different revisions. The files should follow the *module-or-submodule-name@revision-date.yang* naming convention, as specified in the RFC6020. Some versions of the compiler require you to use the **--no-fail-on-warnings** option with the **ncs-make-package** command or the build process may fail.

The `examples.ncs/development-guide/ned-upgrade/yang-revision` example shows how you can perform a YANG model upgrade. The original, 1.0 version of the router NED uses the `router@2020-02-27.yang` YANG model. First, it is updated to the version 1.0.1 `router@2020-09-18.yang` using a revision merge approach. This is possible because the changes are backward-compatible.

In the second part of the example, the updates in `router@2022-01-25.yang` introduce breaking changes, therefore the version is increased to 1.1 and a different ned-id is assigned to the NED. In this case, you can't use revision merge and the usual NED migration procedure is required.

