



## **Administration Guide**

**Release:** NSO 6.1

**Published:** May 17, 2010

**Last Modified:** April 13, 2023

### **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, 2021-2023 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

###### Dynamic configuration 4

###### Built-in or external SSH server 4

##### Starting NSO 5

##### Licensing NSO 5

##### Monitoring NSO 5

###### NSO status 5

###### Monitoring the NSO daemon 6

###### Logging 6

###### syslog 8

###### Log messages and formats 8

###### Trace ID 26

##### Backup and restore 27

###### Backup 27

###### NSO Restore 27

##### Disaster management 27

###### NSO fails to start 28

###### NSO failure after startup 29

###### Transaction commit failure 29

##### Troubleshooting 30

###### Installation Problems 30

###### Problems Starting NSO 30

Problems Running Examples	30
Problems Using and Developing Services	31
General Troubleshooting Strategies	31

---

## CHAPTER 3

<b>Cisco Smart Licensing</b>	<b>35</b>
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

<b>NSO Alarms</b>	<b>43</b>
Overview	43
Alarm type structure	43
Alarm type descriptions	44

---

## CHAPTER 5

<b>NSO Packages</b>	<b>53</b>
Package Overview	53
Loading Packages	54
Redeploying Packages	55
Adding NED Packages	55
NED Migration	56
Managing Packages	58
Package repositories	58
Actions	58

---

## CHAPTER 6

<b>Advanced Topics</b>	<b>61</b>
Locks	61
Global locks	61
Transaction locks	62
Northbound agents and global locks	62
External data providers	62

CDB	62
Lock impact on user sessions	63
Compaction	63
Automatic Compaction	63
Manual Compaction	63
Delayed Compaction	64
IPC ports	64
Restricting access to the IPC port	65
Restart strategies for service manager	65
Security issues	65
Running NSO as a non privileged user	67
Using IPv6 on northbound interfaces	67

---

## CHAPTER 7

<b>High Availability</b>	<b>69</b>
Introduction to NSO High Availability	69
NSO built-in HA	71
Prerequisites	71
HA Member configuration	71
HA Roles	72
Failover	72
Startup	74
Actions	75
Status Check	75
Tail-f HCC Package	76
Overview	76
Dependencies	76
Running the HCC Package with NSO as a Non-Root User	77
Tail-f HCC Compared with HCC Version 4.x and Older	77
Upgrading	77
Layer-2	77
Layer-3 BGP	79
Usage	80
Data Model	83
Setup with an External Load Balancer	86
NB listen addresses on HA primary for Load Balancers	89

HA framework requirements	89
Mode of operation	90
Security aspects	92
API	92
Ticks	92
Relay secondaries	93
CDB replication	94

---

## CHAPTER 8

### Rollbacks 95

Introduction	95
Configuration	95

---

## CHAPTER 9

### The AAA infrastructure 97

The problem	97
Structure - data models	97
Data model contents	98
AAA related items in ncs.conf	98
Authentication	99
Public Key Login	101
Password Login	102
PAM	102
External authentication	103
External token validation	105
External multi factor authentication	107
Package authentication	109
Restricting the IPC port	111
Group Membership	111
Authorization	112
Command authorization	113
Rpc, notification, and data authorization	116
NACM Rules and Services	121
Authorization Examples	122
The AAA cache	125
Populating AAA using CDB	125
Hiding the AAA tree	125

---

**CHAPTER 10**

<b>Upgrade</b>	<b>127</b>
Preparing for Upgrade	127
Single Instance Upgrade	129
Recover from Failed Upgrade	130
NSO HA Version Upgrade	131
Package Upgrade	134
Patch Management	137

---

**CHAPTER 11**

<b>Deployment Example</b>	<b>139</b>
Initial NSO Installation	140
Initial NSO Configuration	143
The <code>ncs.conf</code> Configuration	143
The <code>aaa_init.xml</code> Configuration	145
The High-Availability and VIP Configuration	145
Global Settings and Timeouts	147
Initial Package Setup	147
Cisco Smart Licensing	148
Verifying the Initial NSO Configuration	148
Log Management	148
Log Rotate	149
Syslog	149
NED Logs	149
Python Logs	149
Java Logs	149
Internal NSO Log	150
Monitoring the Installation	150
Alarms	150
Metric - Counters, Gauges and Rate of Change Gauges	150
Counters	150
Gauges	150
Rate of change gauges	150
Security Considerations	151

---

**CHAPTER 12**

<b>Administration</b>	<b>153</b>
User Management	153

Packages	154
Adding and upgrading a package	155
Simulating the new device	156
Adding the new devices to NSO	156
Configuring NSO	157
ncs.conf	157
Run-time configuration	157
Monitoring NSO	157
Backup and Restore	157
Backup	158
NSO Restore	158
<b>CHAPTER 13</b>	
<b>Running NSO in Containers</b>	<b>159</b>
Introduction	159
Getting Started	159
System Requirements	159
Running the Image	160
Administration	161
ncs.conf File Configuration and Preference	161
Admin User Creation	162
Exposed Ports	162
Backup and Restore	163
SSH Host Key	164
HTTPS TLS Certificate	164
NSO Upgrade	164
YANG Model Changes (destructive)	165
Health Check	165





## CHAPTER 1

# Introduction

---

Cisco Network Service Orchestrator (NSO) version 6.1 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 27](#)
- [Disaster management, page 27](#)
- [Troubleshooting, page 30](#)

## Introduction

Cisco Network Service Orchestrator enabled by Tail-f (NSO) version 6.1 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.

## 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 augmented)
session	Global default CLI session parameters
snmp	Top-level container for SNMP related configuration and status objects
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-monitoring.yang` and `tailf-ncs-monitoring.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-monitoring.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:ServiceforVMstart
```

- ```
id=vnf-catalogue-esc daemonId=13 daemonName=ncs-dp-9-vnf-catalogue-esc:ESCVNFCatalogueServ
```
- *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.l*, *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.
- *rollbackNNNN*: 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 25, *Progress Trace in Development Guide*.

## 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
<b>Comment</b>	
<b>Format String</b>	
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.	



Symbol	Severity
<b>Comment</b>	
<b>Format String</b>	
"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
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"	

Symbol	Severity
<b>Comment</b>	
<b>Format String</b>	
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
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

Symbol	Severity
<b>Comment</b> <b>Format String</b>	
Automatic CDB upgrade failed. This means that the data model has been changed in a non-supported way. "CDB: Upgrade failed: ~s"	
CGI_REQUEST CGI script requested. "CGI: '~s' script with method ~s"	INFO
CLI_CMD_ABORTED CLI command aborted. "CLI aborted"	INFO
CLI_CMD_DONE CLI command finished successfully. "CLI done"	INFO
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.	CRIT

Symbol	Severity
<b>Comment</b>	
<b>Format String</b>	
"Daemon ~s died"	
DAEMON_TIMEOUT	CRIT
An external database daemon did not respond to a query.	
"Daemon ~s timed out"	
DEVEL_AAA	INFO
Developer aaa log message	
"~s"	
DEVEL_CAPI	INFO
Developer C api log message	
"~s"	
DEVEL_CDB	INFO
Developer CDB log message	
"~s"	
DEVEL_CONFD	INFO
Developer ConfD log message	
"~s"	
DEVEL_ECONFD	INFO
Developer econfd api log message	
"~s"	
DEVEL_SLS	INFO
Developer smartlicensing api log message	
"~s"	
DEVEL_SNMPA	INFO
Developer snmp agent log message	
"~s"	
DEVEL_SNMPGW	INFO
Developer snmp GW log message	
"~s"	
DEVEL_WEBUI	INFO
Developer webui log message	
"~s"	
DUPLICATE_NAMESPACE	CRIT
Duplicate namespace found.	
"The namespace ~s is defined in both module ~s and ~s."	
DUPLICATE_PREFIX	CRIT

Symbol	Severity
<b>Comment</b>	
<b>Format String</b>	
Duplicate prefix found. "The prefix ~s is defined in both ~s and ~s."	
ERRLOG_SIZE_CHANGED	INFO
Notify change of log size for error log "Changing size of error log (~s) to ~s (was ~s)"	
EVENT_SOCKET_TIMEOUT	CRIT
An event notification subscriber did not reply within the configured timeout period "Event notification subscriber with bitmask ~s timed out, waiting for ~s"	
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

Symbol	Severity
<b>Comment</b>	
<b>Format String</b>	
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"	
FILE_ERROR	CRIT
File error "~s: ~s"	
FILE_LOAD	DEBUG
System loaded a file. "Loaded file ~s"	
FILE_LOAD_ERR	CRIT
System tried to load a file in its load path and failed. "Failed to load file ~s: ~s"	
FILE_LOADING	DEBUG
System starts to load a file. "Loading file ~s"	
FXS_MISMATCH	ERR
A secondary connected to a primary where the fxs files are different "Fxs mismatch, secondary is not allowed"	
GROUP_ASSIGN	INFO
A user was assigned to a set of groups. "assigned to groups: ~s"	
GROUP_NO_ASSIGN	INFO
A user was logged in but wasn't assigned to any groups at all. "Not assigned to any groups - all access is denied"	
HA_BAD_VSN	ERR
A secondary connected to a primary with an incompatible HA protocol version "Incompatible HA version (~s, expected ~s), secondary is not allowed"	
HA_DUPLICATE_NODEID	ERR
A secondary arrived with a node id which already exists	

Symbol	Severity
<b>Comment</b>	
<b>Format String</b>	
"Nodeid ~s already exists"	
HA_FAILED_CONNECT	ERR
An attempted library become secondary call failed because the secondary couldn't connect to the primary	
"Failed to connect to primary: ~s"	
HA_SECONDARY_KILLED	ERR
A secondary node didn't produce its ticks	
"Secondary ~s killed due to no ticks"	
INTERNAL_ERROR	CRIT
A ConfD internal error - should be reported to support@tail-f.com.	
"Internal error: ~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).	

Symbol	Severity
<b>Comment</b>	
<b>Format String</b>	
"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. "~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"	



Symbol	Severity
<b>Comment</b>	
<b>Format String</b>	
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)"	
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"	
NETCONF_HDR_ERR	ERR

Symbol	Severity
<b>Comment</b>	
<b>Format String</b>	
The cleartext header indicating user and groups was badly formatted. "Got bad NETCONF TCP header"	
NETCONF NETCONF traffic log message "~s"	INFO
NOAAA_CLI_LOGIN A user used the --noaaa flag to confd_cli "logged in from the CLI with aaa disabled"	INFO
NO_CALLPOINT 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"	CRIT
NO_SUCH_IDENTITY 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"	CRIT
NO_SUCH_NS A nonexistent namespace was referred to. Typically this means that a .fxs was missing from the loadPath. "No such namespace ~s, used by ~s"	CRIT
NO_SUCH_TYPE 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"	CRIT
NOTIFICATION_REPLAY_STORE_FAILURE A failure occurred in the builtin notification replay store "~s"	CRIT
NS_LOAD_ERR2 System tried to process a loaded namespace and failed. "Failed to process namespaces: ~s"	CRIT
NS_LOAD_ERR System tried to process a loaded namespace and failed. "Failed to process namespace ~s: ~s"	CRIT
OPEN_LOGFILE Indicate target file for certain type of logging "Logging subsystem, opening log file '~s' for ~s"	INFO
PAM_AUTH_FAIL A user failed to authenticate through PAM.	INFO

Symbol	Severity
<b>Comment</b>	
<b>Format String</b>	
"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	
"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

Symbol	Severity
<b>Comment</b>	
<b>Format String</b>	
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
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.	

Symbol	Severity
<b>Comment</b>	
<b>Format String</b>	
"Skipping file ~s: ~s"	
SNMP_AUTHENTICATION_FAILED An SNMP authentication failed. "SNMP authentication failed: ~s"	INFO
SNMP_CANT_LOAD_MIB The SNMP Agent failed to load a MIB file "Can't load MIB file: ~s"	CRIT
SNMP_MIB_LOADING SNMP Agent loading a MIB file "Loading MIB: ~s"	DEBUG
SNMP_NOT_A_TRAP An UDP package was received on the trap receiving port, but it's not an SNMP trap. "SNMP gateway: Non-trap received from ~s"	INFO
SNMP_READ_STATE_FILE_FAILED Read SNMP agent state file failed "Read state file failed: ~s: ~s"	CRIT
SNMP_REQUIRES_CDB The SNMP agent requires CDB to be enabled in order to be started. "Can't start SNMP. CDB is not enabled"	WARNING
SNMP_TRAP_NOT_FORWARDED An SNMP trap was to be forwarded, but couldn't be. "SNMP gateway: Can't forward trap from ~s; ~s"	INFO
SNMP_TRAP_NOT_RECOGNIZED 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."	INFO
SNMP_TRAP_OPEN_PORT The port for listening to SNMP traps could not be opened. "SNMP gateway: Can't open trap listening port ~s: ~s"	ERR
SNMP_TRAP_UNKNOWN_SENDER 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"	INFO
SNMP_TRAP_V1 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"	INFO

Symbol	Severity
<b>Comment</b>	
<b>Format String</b>	
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"	
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).	

Symbol	Severity
<b>Comment</b>	
<b>Format String</b>	
"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	
"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

Symbol	Severity
<b>Comment</b>	
<b>Format String</b>	
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"	
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	



Symbol	Severity
<b>Comment</b>	
<b>Format String</b>	
"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	
"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

Symbol	Severity
<b>Comment</b>	
<b>Format String</b>	
External authentication failed for a user. "failed to login using externalauth: ~s"	
NO_SUCH_LOCAL_USER A non existing local user tried to login. "no such local user"	INFO
PAM_LOGIN_FAILED A user failed to login through PAM. "pam phase ~s failed to login through PAM: ~s"	INFO
PAM_NO_LOGIN A user failed to login through PAM "failed to login through PAM: ~s"	INFO
SSH_LOGIN A user logged into ConfD's builtin ssh server. "logged in over ssh from ~s with authmeth:~s"	INFO
SSH_LOGOUT A user was logged out from ConfD's builtin ssh server. "Logged out ssh <~s> user"	INFO
SSH_NO_LOGIN A user failed to login to ConfD's builtin SSH server. "Failed to login over ssh: ~s"	INFO
WEB_LOGIN A user logged in through the WebUI. "logged in through Web UI from ~s"	INFO
WEB_LOGOUT A Web UI user logged out. "logged out from Web UI"	INFO

## 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" .

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,  $x\ 1024 == 5120$ . 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 a 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 SSH implementation Paramiko. Since you are seeing this message, your operating system doesn't have the python-module Paramiko installed. The Paramiko package, in turn, depends on a Python crypto library (pycrypto).

Resolution: Install Paramiko (and pycrypto, if necessary) using the standard installation mechanisms for your OS. An alternative approach is to go to the project home pages to fetch, build and install the missing packages.

- <https://www.lag.net/paramiko/>
- <https://www.amk.ca/python/code/crypto>

These packages come with simple installation instructions. You will need root privileges to install these packages, however. When properly installed, you should 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 or Pycrypto. 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 14, *Developing NSO 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 `top` to examine resource utilization, and `free -mh` 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 `df -h` command. In case there is

	<p>enough space on disk and you still encounter ENOSPC errors, check the inode usage with <code>df -i</code> command.</p>
Status	<p>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</p> <pre>\$ ncs --status</pre>
Check data provider	<p>NSO status information is also available as operational data under <code>/ncs-state</code>.</p> <p>If you are implementing a data provider (for operational or configuration data), you can verify that it works for all possible data items using</p>
Debug dump	<pre>\$ ncs --check-callbacks</pre> <p>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</p> <pre>\$ ncs --debug-dump mydump1</pre> <p>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.</p> <p>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.</p>
Error log	<p>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.</p>
System dump	<p>If NSO aborts due to failure to allocate memory (see <a href="#">the section called "Disaster management"</a>), 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 name is <code>ncs_crash.dump</code> and it could be changed by setting the environment variable <code>\$NCS_DUMP</code> before starting NSO. The dumped information is only comprehensible to the NSO development team, so send the dump to your support contact.</p>
System call trace	<p>To catch certain types of problems, especially relating to system start and configuration, the operating system's system call trace can</p>



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:

*Enter person's company 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:

*Account Name is typically the compai*

## 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:

Background text: Cisco Software Central, My Smart Account, Account Properties, Users, New User..., User Name, Adam Groudan, 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

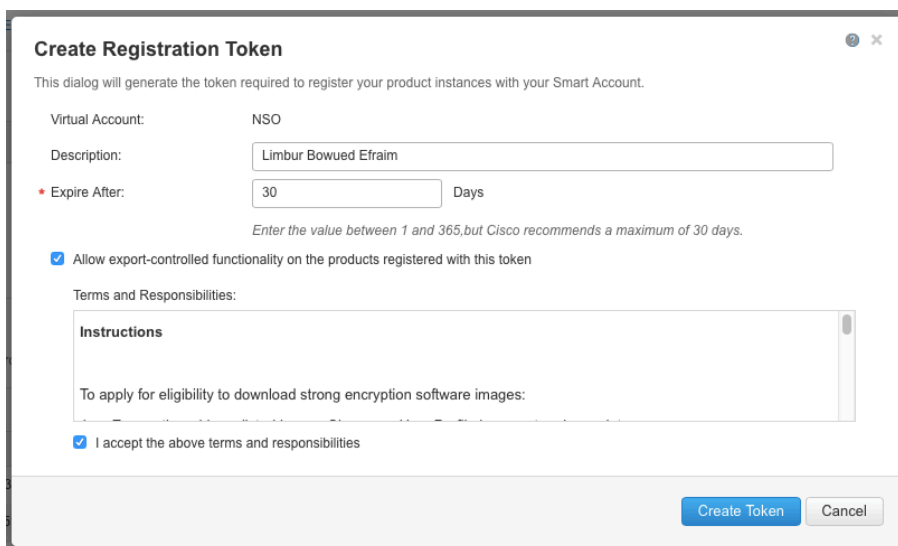
**Product Instance Registration Tokens**

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

[New Token...](#)

Token	Expiration Date	Description	Export-Controlled
YjQ2YzhiNWMTYTM1My00NzQ...	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**

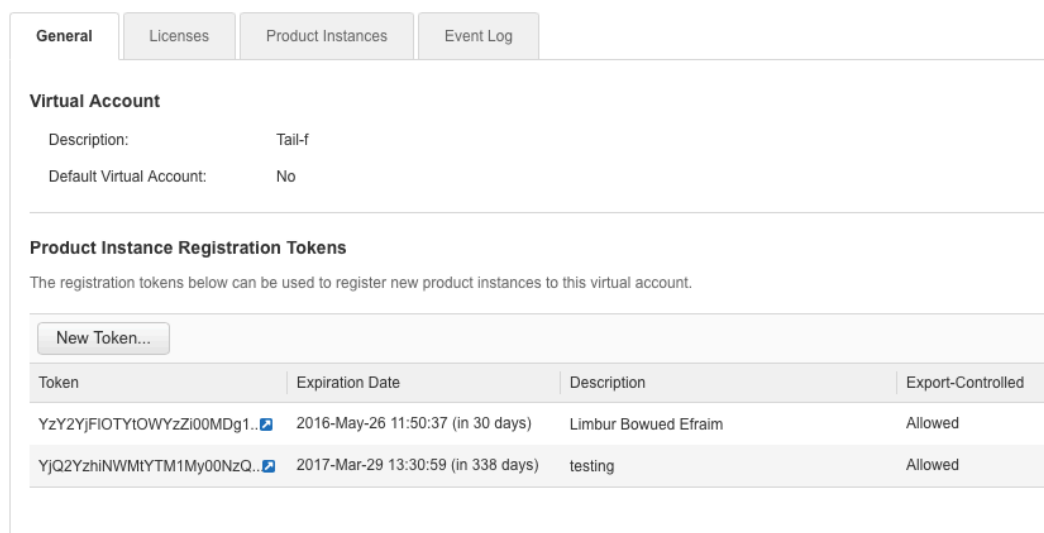
To apply for eligibility to download strong encryption software images:

☒ 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

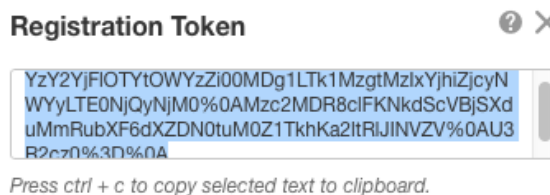
**Product Instance Registration Tokens**

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

<b>show license all</b>	Displays all information
<b>show license status</b>	Displays status information
<b>show license summary</b>	Displays summary
<b>show license tech</b>	Displays license tech support information
<b>show license usage</b>	Displays usage information

### Available Show Commands

<b>debug smart_lic all</b>	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 north-bound 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
    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-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)**

<b>Alarm Identity</b>	<b>Initial Perceived Severity</b>
abort-error	major
<b>Description</b>	<b>Recommended Action</b>
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.
<b>Alarm message(s)</b>	
<ul style="list-style-type: none"> <li>• Device {dev} is locked</li> <li>• Device {dev} is southbound locked</li> <li>• abort error</li> </ul>	
<b>Clear condition(s)</b>	
If NCS achieves sync with the device, or receives a transaction id for a netconf session towards the device, the alarm is cleared.	
<b>Alarm Identity</b>	
alarm-type	
<b>Description</b>	
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.	
<b>Alarm Identity</b>	<b>Initial Perceived Severity</b>
bad-user-input	critical
<b>Description</b>	<b>Recommended Action</b>
Invalid input from user. NCS cannot recognize parameters needed to connect to device.	Verify that the user supplied input are correct.
<b>Alarm message(s)</b>	
<ul style="list-style-type: none"> <li>• Resource {resource} doesn't exist</li> </ul>	
<b>Clear condition(s)</b>	

This alarm is not cleared.

<b>Alarm Identity</b> cluster-subscriber-failure	<b>Initial Perceived Severity</b> critical
<b>Description</b> Failure to establish a notification subscription towards a remote node.	<b>Recommended Action</b> 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.

<b>Alarm Identity</b> commit-through-queue-blocked	<b>Initial Perceived Severity</b> 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.

<b>Alarm Identity</b> commit-through-queue-failed	<b>Initial Perceived Severity</b> critical
------------------------------------------------------	-----------------------------------------------

<b>Description</b> A queued commit failed.	<b>Recommended Action</b> 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}
- Device {dev} is locked
- the configuration database is locked by session {id} {identification}
- {Reason}
- {Dev}: Device is locked in a {Op} operation by session {session-id}
- resource denied

- 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-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.

<b>Alarm Identity</b> final-commit-error	<b>Initial Perceived Severity</b> critical
<b>Description</b> A managed device validated a configuration change, but failed to commit. When this happens, NCS and the device are out of sync.	<b>Recommended Action</b> 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****Initial Perceived Severity**

ha-secondary-down	critical
<b>Description</b> The node lost the connection to a secondary node.	<b>Recommended Action</b> Investigate why the secondary node went down, fix the connectivity issue and reconnect the secondary to the HA cluster.
<b>Alarm message(s)</b> • Lost connection to secondary	
<b>Clear condition(s)</b> This alarm is cleared when the secondary node is reconnected to the HA cluster.	
<b>Alarm Identity</b> missing-transaction-id	<b>Initial Perceived Severity</b> warning
<b>Description</b> A device announced in its NETCONF hello message that it supports the transaction-id as defined in <a href="http://tail-f.com/yang/netconf-monitoring">http://tail-f.com/yang/netconf-monitoring</a> . However when NCS tries to read the 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.	<b>Recommended Action</b> Verify NACM rules on the concerned device.
<b>Alarm message(s)</b> • {Reason}	
<b>Clear condition(s)</b> If NCS successfully reads a transaction id for which it had previously failed to do so, the alarm is cleared.	
<b>Alarm Identity</b> ncs-cluster-alarm	
<b>Description</b> Base type for all alarms related to cluster. This is never reported, sub-identities for the specific cluster alarms are used in the alarms.	
<b>Alarm Identity</b> ncs-dev-manager-alarm	
<b>Description</b> 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.	
<b>Alarm Identity</b> ncs-package-alarm	
<b>Description</b> 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**

revision-error

**Initial Perceived Severity**

major

**Description**

A managed device arrived with a known module, but too new revision.

**Recommended Action**

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**

service-activation-failure

**Initial Perceived Severity**

critical

**Description**

A service failed during re-deploy.

**Recommended Action**

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**

transaction-lock-time-violation

**Initial Perceived Severity**

warning

**Description**

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.

**Recommended Action**

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 55](#)
- [NED Migration, page 56](#)
- [Managing Packages, page 58](#)

## 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. NED contains device YANG model and the code, specifying how NSO should connect to the device. For NETCONF devices, NSO includes tools to help you build a NED, as shown in Chapter 3, *NETCONF NED Builder* in *NED Development*, but often a vendor provides the required NEDs. 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. In this case, 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 the reload.

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 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.

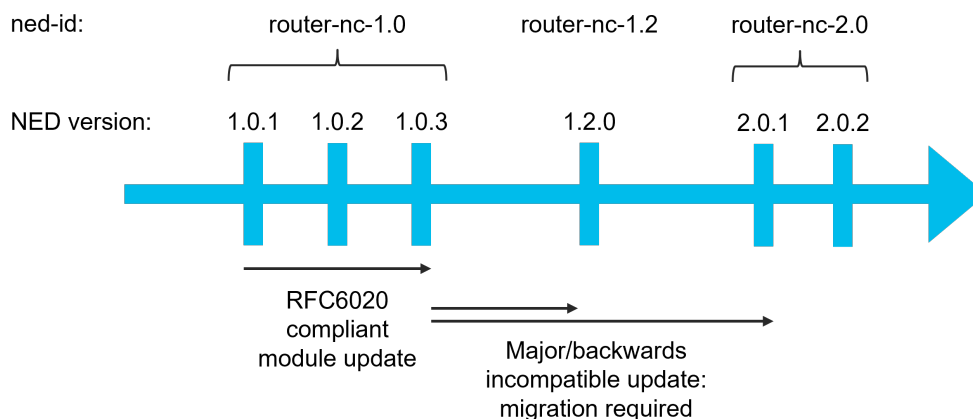
## NED Migration

If you upgrade a managed device, such as installing a new firmware, the device data model can change in some significant way. If that is the case, you usually need to use a different, 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. So, 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 won't 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*.

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 the 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 3rd-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 3. 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 to migration tooling for identifying the paths and service instances that may be impacted. So, please ensure 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 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. Afterwards, 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.

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.

## Managing Packages

In a "system install" of NSO, management of pre-built packages is supported through a number of actions, as well as the possibility to configure remote software repositories from which packages can be retrieved. 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.

## Package repositories

The `/software/repository` list allows for configuration of one or more remote repositories. The Tail-f delivery server can be configured as a repository, but it is also possible to set up a local server for this purpose - the (simple) requirements are described in detail in the YANG submodule. Authenticated access to the repositories via HTTP or HTTPS is supported.

### Example 4. Configuring a remote repository

```
admin@ncs(config)# software repository tail-f
Value for 'url' (<string>): https://support.tail-f.com/delivery
admin@ncs(config-repository-tail-f)# user name
admin@ncs(config-repository-tail-f)# password
(<AES encrypted string>): *****
admin@ncs(config-repository-tail-f)# commit
Commit complete.
```

## Actions

Actions are provided to list local and repository packages, to fetch packages from a repository or 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.
- **software repository *name* packages list [...]** List packages available in the repository identified by *name*. The list can be filtered via the `name-pattern` option.
- **software repository *name* packages fetch package *package-name*** Fetch a package from the repository identified by *name*, making it *installable*.

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 61](#)
- [Compaction, page 63](#)
- [IPC ports, page 64](#)
- [Restart strategies for service manager, page 65](#)
- [Security issues, page 65](#)
- [Running NSO as a non privileged user, page 67](#)
- [Using IPv6 on northbound interfaces, page 67](#)

## 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 13, *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:

<b>config exclusive</b>	The running datastore global lock will be taken.
<b>config terminal</b>	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 13, *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 24, *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 <b><code>ncs --reload</code></b> , 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, <b><code>ncs_cli</code></b> , 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 <b>-P</b> 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/priv/ncs/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/priv/ncs/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@ncs(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 69](#)
- [NSO built-in HA, page 71](#)
- [Tail-f HCC Package, page 76](#)
- [Setup with an External Load Balancer, page 86](#)
- [NB listen addresses on HA primary for Load Balancers, page 89](#)
- [HA framework requirements, page 89](#)
- [Mode of operation, page 90](#)
- [Security aspects, page 92](#)
- [API, page 92](#)
- [Ticks, page 92](#)
- [Relay secondaries, page 93](#)
- [CDB replication, page 94](#)

## Introduction to NSO High Availability

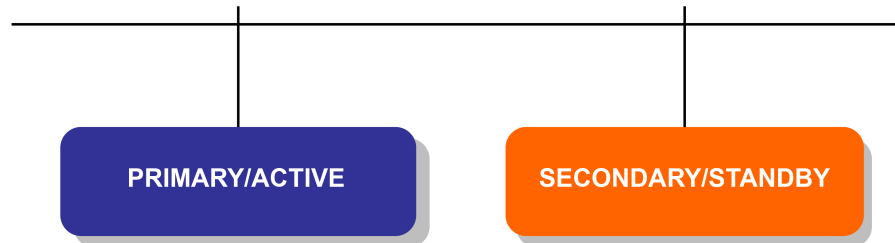
NSO supports replication of the CDB configuration as well as of the operational data kept in CDB. The replication architecture is that of one active primary and a number of passive secondaries.

A group of NSO hosts consisting of a primary, and one or more secondaries, is referred to as an HA group (and sometimes as an HA cluster - however this is completely independent and separate from the Layered Service Architecture cluster feature).

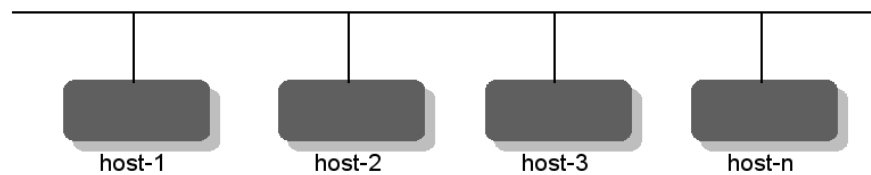
All configuration write operations must occur at the primary and NSO will automatically distribute the configuration updates to the set of live secondaries. Operational data in CDB may be replicated or not based on the `tailf:persistent` statement in the data model. All write operations for replicated operational data must also occur at the primary, with the updates distributed to the live secondaries, whereas non-replicated operational data can also be written on the secondaries.

As of NSO 5.4, NSO has built capabilities for defining HA group members, managing assignment of roles, handling failover etc. Alternatively, an external HA framework can be used. If so, NSO built-in HA must be disabled. The concepts described in the following sections applies both for NSO built-in HA and external HA framework.

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

Furthermore it is assumed that the entire cluster configuration is equal on all hosts in the cluster. This means that node specific configuration must be kept in different node specific subtrees, for example as in [Example 5, “A data model divided into common and node specific subtrees”](#).

#### Example 5. A data model divided into common and node specific subtrees

```
container cfg {
  container shared {
    leaf dnserver {
      type inet:ipv4-address;
    }
    leaf defgw {
      type inet:ipv4-address;
    }
    leaf token {
      type AESCFB128EncryptedString;
    }
    ...
  }
  container cluster {
    list host {
      key ip;
      max-elements 8;
    }
  }
}
```

```

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

```

## NSO built-in HA

NSO has capabilities for managing HA groups out of the box as of version 5.4 and greater. The built-in capabilities 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, enabled/disable built-in HA through actions
- View the state of current HA setup

NSO Built-in HA is defined in `tailf-ncs-high-availability.yang`, and is found under `/high-availability/`.

NSO built-in 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 built-in HA. You can read more about the HCC package in the [following chapter](#)

Note: External HA frameworks are supported but should not be used in parallel with NSO built-in HA.

## Prerequisites

In order to use NSO built-in HA, HA must first be enabled `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 built in 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 built-in 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-l2* 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 Built-in HA does not support relay-secondaries.

NSO Built-in HA differs 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, no retry attempts are made.
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/reconnect-interval</code>
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 <code>/high-availability/settings/reconnect-interval</code>
true	true	none	Assume none role.
false	false	-	Assume none role.

## Actions

NSO Built-in 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 <code>/high-availability/status/be-secondary-result</code>
local-node-id	Identify the which of the nodes in <code>/high-availability/ha-node</code> (if any) corresponds to the local NSO instance
enable	Enable NSO built in HA and optionally assume a ha role according to <code>/high-availability/settings/start-up/parameters</code>
disable	Disable NSO built in HA and assume a ha role none

## Status Check

The current state of NSO Built-in 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 built-in 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.

**Table 6. Tools Dependencies**

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

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 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
```

- 2 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
```

## 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”](#).

## 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 7. 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 (`devel.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 and its location provided to HCC as configuration data.

### 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 8. 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.

Parameters	Type	Description
gobgp-bin-dir	string	Directory containing gobgp and gobgpd binaries.
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 9. 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 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
```

## 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:fefa: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. See the 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	64514	BGP-enabled router
	192.168.31.2		
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. See the `examples.ncs/development-guide/high-availability/hcc/README`

# 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;
  }

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

  revision 2022-05-20 {
    description
      "Use bias-free language.";
  }

  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 {
        key node-id;

        leaf node-id {
            type leafref {
                path "/ncs:high-availability/ncs:ha-node/ncs:id";
            }
            description "Unique NCS node ID";
            mandatory true;
        }

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

        leaf gobgp-bin-dir {
            type string;
            tailf:info "Directory containing gobgp/gobgpd binaries";
            mandatory true;
            description
                "The directory where 'gobgp' and 'gobgpd' binary executables
                have been installed separately.";
        }

        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.";
    }

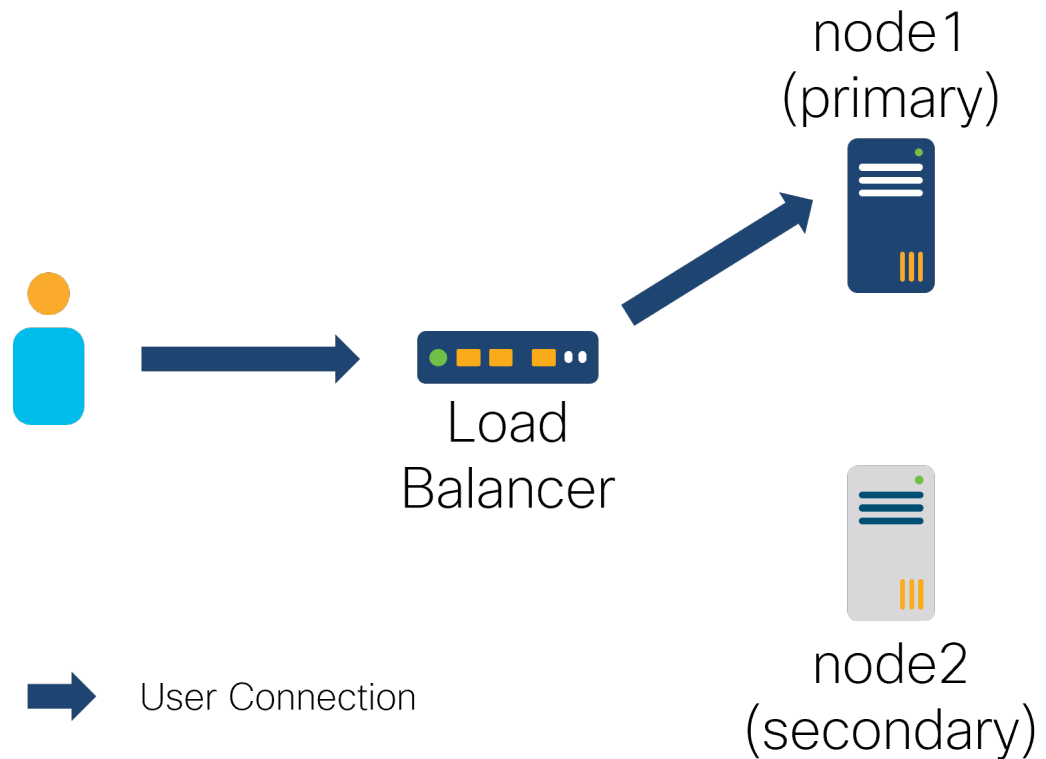
    list neighbor {
        key "address";
        description "BGP neighbor list";
        leaf address {
            type inet:ip-address;
            mandatory true;
            description "BGP neighbor IP address";
        }
        leaf as {
            type inet:as-number;
            mandatory true;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

```

## Setup with an External Load Balancer

## Administration Guide

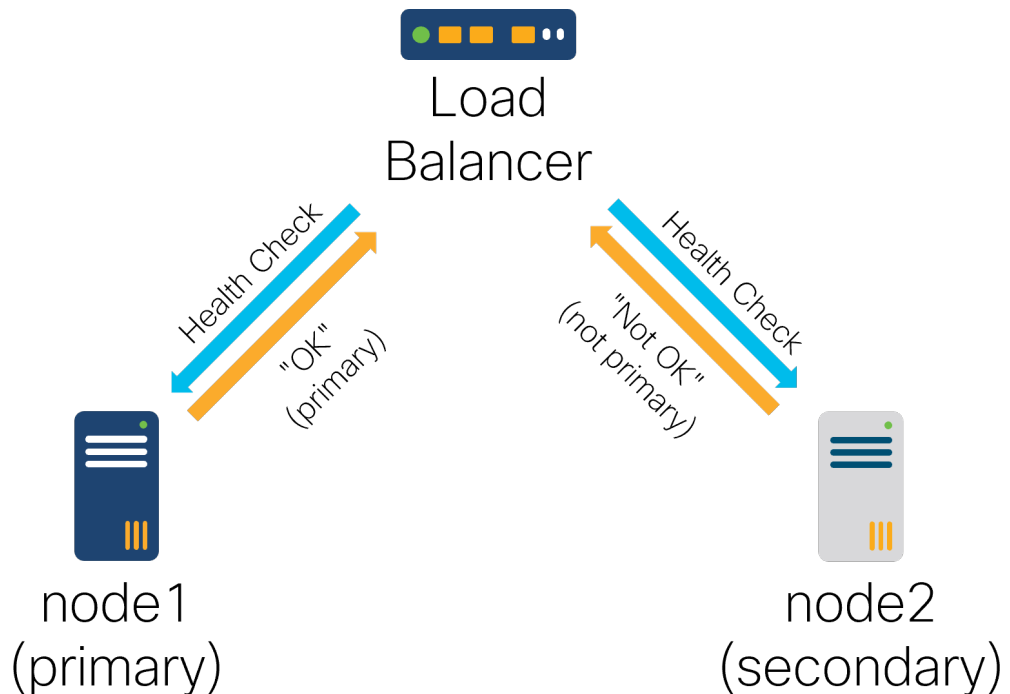
Figure 12. Load balancer routes connections to the appropriate NSO node



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. It can be configured to be done asynchronously (best performance) or synchronously in step with the transaction (most secure). 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.

Operational data is always replicated on all secondaries similar to how configuration data is replicated. Operational data is always replicated asynchronously, regardless of the `/ncs-config/cdb/operational/replication` setting.



## CHAPTER 8

# Rollbacks

---

- [Introduction, page 95](#)
- [Configuration, page 95](#)

## 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 97](#)
- [Structure - data models, page 97](#)
- [AAA related items in ncs.conf, page 98](#)
- [Authentication, page 99](#)
- [Restricting the IPC port, page 111](#)
- [Group Membership, page 111](#)
- [Authorization, page 112](#)
- [The AAA cache, page 125](#)
- [Populating AAA using CDB, page 125](#)
- [Hiding the AAA tree, page 125](#)

## 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, the SSH keys are not necessary.

`/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
```

If we always authenticate users through PAM it may make sense to set this configurable to `false`. 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 `/aaa/authentication/users` and, in the case of "external auth" also `/nacm/groups` (for NACM) or `/aaa/authentication/groups` (for legacy `tailf-aaa`) empty. NSO can authenticate users using PAM (Pluggable Authentication Modules). PAM is an integral part of most Unix-like systems.

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 also assign a user to a set of groups.

```
/ncs-config/aaa/default-group
```

PAM authentication is thoroughly described later in this chapter.

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.

```
/ncs-config/aaa/external-authentication
```

NSO can authenticate users using an external executable. This is further described later in the [the section called "External authentication"](#) section.

```
/ncs-config/aaa/external-validation
```

NSO can authenticate users by validation of tokens using an external executable. This is very similar to what is further described later in the [the section called "External token validation"](#) section. The difference is that a token, instead of a username and password, is input and a username and, optionally, a token is output. This is currently only supported for RESTCONF.

```
/ncs-config/aaa/external-challenge
```

NSO has support for multi factor 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 later in the [the section called "External multi factor authentication"](#) section.

## 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>$1$feedbabe$nG1MY1ZpQ0bzenyFOQI3L1</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, and external 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 $USER\n$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 $USER\n$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 $USER\n$token $info\n"
```

```
"accept_token_warning $groups $uid $gid $supplementary_gids $HOME $USER\n$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, why it must be base64 encoded

## 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.

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 currently only supported for Single Sign-On (see Chapter 4, *Single Sign-On in Web UI*).

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.

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 resource requiring authentication, redirecting to `/sso`), 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.

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

```
"[;;;R0VUIC9zc28vc2FtbC9sb2dpbi8gSFRUUC8xLjE=;;;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.

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.

## 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

This is a choice between three possible leafs that are used for matching, in addition to the module-name:

rpc-name

The name of a rpc operation, or "\*" to match any rpc.

notification-name

The name of a notification, or "\*" to match any notification.

path

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.

- 1 Tagpaths that are not containing any keys. For example / ncs/live-device/live-status.
- 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.
- 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.

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.

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.

## 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, the post-actions for nano-services are run in a user session without any access checks.

## 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>
  </rule>

```

```

        <access-operations>exec</access-operations>
        <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.

NSO ships with a decent initialization document for the AAA database. The file is called `aaa_init.xml` and is by default copied to the CDB directory by the NSO install scripts. The file defines two users, `admin` and `oper` with passwords set to `admin` and `oper` respectively.

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 9, *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 127](#)
- [Single Instance Upgrade, page 129](#)
- [Recover from Failed Upgrade, page 130](#)
- [NSO HA Version Upgrade, page 131](#)
- [Package Upgrade, page 134](#)
- [Patch Management, page 137](#)

## 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 5.6.1 to 5.6.2. 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 5.6.1 to 5.7 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 Update the `/opt/ncs/current` symbolic link.
- 5 If required, update the `ncs.conf` configuration file.
- 6 Update the packages in `/var/opt/ncs/packages/` if recompilation is needed.
- 7 Start the NSO server process, instructing it to reload the packages.

The following steps suppose that you are upgrading to the 5.7 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-5.7.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: .
```

Next, you update the symbolic link for the currently selected version to point to the newly installed one, 5.7 in this case.

```
# cd /opt/ncs
# rm -f current
# ln -s ncs-5.7 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-5.7-*; 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 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 14. 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 15. 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
}

# 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-5.7@2022-01-21T10:34:42.backup.gz created
successfully
# ls /opt/ncs/packages
ncs-5.7-router-nc-1.0 ncs-5.7-router-nc-1.0.2
# ncs_cli -C
admin@ncs# software packages install package router-nc-1.0.2 replace-existing
installed ncs-5.7-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 found in the load paths on 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.

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 16. 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

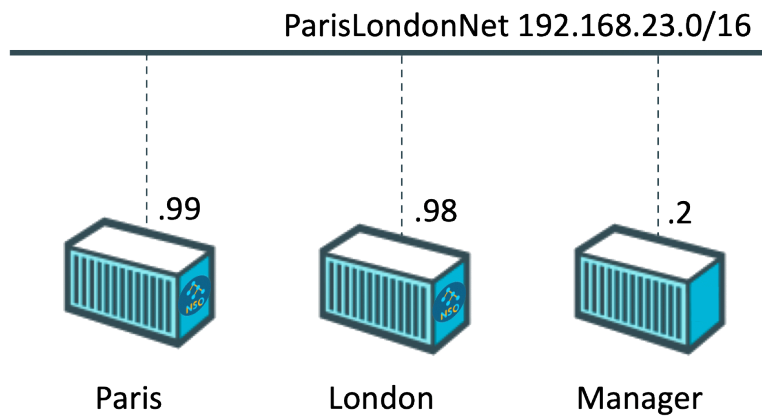
# Deployment Example

This chapter shows a series of examples in the typical order of deployment. A reference to the container-based example for the HCC layer-2 upgrade deployment scenario described here can be found in the NSO example set under `examples.ncs/development-guide/high-availability/hcc`. We will be describing a typical deployment and cover the following topics:

- Installation of NSO on all hosts
- Initial configuration of NSO on all hosts
- Upgrade of NSO on all hosts
- Upgrade of NSO packages/NEDs on all hosts
- Monitoring the installation
- Metric - counters, gauges and rate of change gauges
- Troubleshooting, backups, and disaster recovery
- Security considerations

We will be using a high availability setup as an example deployment. See [Chapter 7, High Availability](#), for HA details. The deployment consists of two hosts, a *paris* node and a *london* node, managed by a "manager" node. The *paris* and *london* nodes are an NSO HA pair.

**Figure 17. The Deployment Network**



Thus the two NSO hosts, *paris* and *london*, make up one HA pair, one *primary* and one *secondary*. We will describe the HA setup in detail later in this chapter.

Optionally the services on the *paris* and *london* nodes can be split up into a layered cluster with resource and customer-facing service (RFS and CFS) nodes when the amount of managed devices and instantiated services increase beyond what can fit on a single NSO host.

Avoid the complexity introduced by clustering if the expected number of managed devices and services is less than 20k. Instead equip the NSO hosts with sufficient RAM. Installation, performance, bug search, observation, and maintenance become harder with layered services clustering. See Chapter 1, *LSA Overview* in *Layered Service Architecture*, for more information on the Layered Service Architecture (LSA) design approach.

HA, on the other hand, is usually not optional for a deployment. Data resides in CDB, a RAM database with a disk-based journal for persistence. One possibility to run NSO without the HA component could be to use a fault-tolerant filesystem, such as CEPH. Provisioning data then survive a disk crash on the NSO host, but failover would require manual intervention. As we shall see, the NSO built-in HA can be set up not to require manual intervention.

In this chapter, we will describe a HA setup, and you will have to decide for your deployment whether HA is needed.

- [Initial NSO Installation, page 140](#)
- [Initial NSO Configuration, page 143](#)
- [Log Management, page 148](#)
- [Monitoring the Installation, page 150](#)
- [Metric - Counters, Gauges and Rate of Change Gauges, page 150](#)
- [Security Considerations, page 151](#)

## Initial NSO Installation

We will perform an NSO system installation on two NSO container nodes. To read more about the system installation, see the section called “System Install Steps” in *Getting Started*.

In this container-based example, a Dockerfile enable us to easily install NSO on multiple hosts, here containers. For installations on multiple physical or virtual machine hosts, `nct` can be a helpful tool. `nct` is shipped together with NSO, documented by `nct(1)` in *Manual Pages*, and will not be described here. Instead, we will use a shell script to setup our containers, a Dockerfile, and run the demo with a shell script, and, as an alternative, a simple Python script that implements SSH and RESTCONF clients.

- We set up an *admin* user on the two NSO hosts with only `sudo` rights to run the `ip` command. The operations towards the two hosts do not require root privileges. Only the Tail-f HCC server, when setting Layer 2 VIP routes using the `ip` command, requires root privileges.

The SSH client uses a pre-shared key, while the RESTCONF client uses Token authentication.

NSO users authenticate through Linux PAM.

- For this example, we create two packages using `ncs-make-package`, `dummy` and `inert`. A third package, Tail-f HCC, provides VIPs that point to the current HA *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 [Chapter 13, Running NSO in Containers](#).

For this example, we use a minimal Debian Linux distribution for hosting NSO with the following added Debian packages:

- NSO's basic dependency requirements are fulfilled by adding the Java Runtime Environment (JRE), OpenSSH server, and make Debian packages.
- The `setcap` program from the `libcap2-bin` package is needed when installing NSO for a non-root user.
- The OpenSSH server is used for shell access and secure copy to the NSO Linux host, which we will use for NSO version upgrade purposes, while we use the NSO built-in SSH server for CLI and NETCONF access to NSO.
- We add the Python Debian package for running our `inert` NSO service package.
- As no application runs Java, we do not need to install JDK or Ant. JRE is sufficient.
- To fulfill the Tail-f HCC server dependencies, we add the `iproute2` utilities, `awk`, `arping`, and `sudo` Debian packages. See the section called “Dependencies” the HCC chapter for details on HCC dependencies.
- We add the `nano` command/package and the `arp` command from the `net-tools` Debian package for demo purposes.

We perform the steps in the list below as root. Docker will build container images, i.e., create the installation, as root. This enables us to not even set up a password for user access or root access.

The admin user will only need root access for running the `ip` when Tail-f HCC adds the Layer 2 VIP address to an interface on the *primary* node.

The initialization steps performed with *root* privileges for the *paris* and *london* nodes that make up the HA group:

- The required services and authentication needs to be configured taking security requirements into account. We are here using Linux PAM, which is recommended for authenticating users. However, it is possible to store users in the NSO CDB database and use local authentication only or in combination with PAM.

We create the `ncsadmin` and `ncsoper` Linux user groups, create and add the `admin` and `oper` Linux users to their respective groups, and perform a system installation of NSO that runs NSO as the `admin` user.

The `admin` user is granted access to running the `ip` from the `vipctl` script as root using the `sudo` command required by the Tail-f 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.

For the read-only `oper` user, we set up password authentication for use with NSO only, intended for WebUI access.

SSH key and token authentication are used for NSO CLI, NETCONF, and RESTCONF authentication.

Password authentication will still be denied for all users to the Linux SSH shell, while the `oper` user is denied access to the Linux shell altogether.

The root user is not authorized to access anything through the NSO northbound interfaces, only through the Linux shell.

- The NSO installer, HCC package, application YANG models, scripts for generating and authenticating RESTCONF tokens, Makefile, and a script for running the demo are all copied to the *paris* and *london* containers.

*admin* user permissions are set for the NSO directories + files created by the system install, for the root, admin, and oper home directories, and for the application files.

- The configuration for encrypted strings is generated during installation. The keys are stored in the file `/etc/ncs/ncs.crypto_keys` and should be copied from one of the hosts to the other host(s) in the HA group.

This is done by default as we use Docker containers where the NSO installation is cached and shared by the *paris* and *london* container nodes. In YANG parlance, this is all YANG data modeled with the types `tailf:des3-cbc-encrypted-string`, `tailf:aes-cfb-128-encrypted-string` or `tailf:aes-256-cfb-128-encrypted-string`.



#### Note

The `ncs.crypto_keys` file is highly sensitive. The file contains the encryption keys for all CDB data that is encrypted on disk. This often include passwords for various entities, such as login credentials to managed devices.

- The token used with the built-in high availability is generated and distributed from the *paris* node to the other HA group nodes, i.e., the *london*, so it matches between the nodes.



#### 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.

- 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, except for executing the generate-token RPC action for RESTCONF authentication use.

The *oper* user cannot access the Linux shell, while the *root* user has zero authorization with NSO.

The *admin* user can access both the Linux shell and NSO CLI using public key authentication.

New keys for all users are distributed to the HA group nodes and the manager node when the HA group 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 servers.

- Host keys are generated and shared between the NSO built-in SSH and OpenSSH servers for authenticating the server to the client as we want to avoid using the host keys that come with NSO, and in this case, the container build where keys are stored in the build cache.

The HA group nodes share SSH host keys under `/etc/ssh/ssh_host_ed25519_key` so that the SSH client(s), here the manager, do not need to keep track of which node the VIP address currently points to.

The host keys, just like the keys used for client authentication, are generated each time the HA group nodes are initialized. The host keys are distributed to the manager and nodes in the HA group before the NSO built-in SSH and OpenSSH servers are started on the nodes.

- As we run NSO in containers, we have set the environment variables pointing to the system install directories in the Dockerfile.

The environment variables have been copied to a `.pam_environment` file for the `root` and `admin` users to set the required environment variables when those users log in via SSH. We, therefore, use the `ncs` command to start NSO and do not use the `/etc/init.d/ncs` and `/etc/profile.d` scripts.

As part of the NSO system install, the Debian/Ubuntu start script was installed, and can be customized if we would like to use it to start NSO. The available NSO `ncs` script variants are can be found under `/opt/ncs/current/src/ncs/package-skeletons/etc`. Perhaps the scripts provide what you need, can be used as a starting point, or will, as for this demo, only be used for stopping NSO.

- The OpenSSH `sshd` daemon is started before we are done with the initialization part in `root` context.

## Initial NSO Configuration

To complete the initialization on the *paris* and *london* nodes before the manager takes over, running as the *admin* user:

- The initial NSO configuration, `ncs.conf`, needs to be updated and in sync (identical) on the nodes.
- The initial AAA configuration needs to be updated and in sync (identical) on the nodes.
- The initial built-in HA and VIP configuration needs to be updated and in sync (identical) on the nodes.
- Global settings may need to be updated.
- Packages are installed.
- We set the NSO smart licensing token.

## The `ncs.conf` Configuration

- The NSO IPC port 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 port are considered trusted, i.e., no authentication is required, and we rely on 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”](#) for more details.

- We edit the default `ncs.conf` under `/ncs-config/aaa/pam` to enable PAM to authenticate users as recommended. All remote access to NSO must now be done using host privileges.

Depending on your Linux distro, you may have to change `/ncs-config/aaa/pam/service`. The default value is `common-auth`. Check the file `/etc/pam.d/common-auth` and make sure it fits your needs.

Alternatively, or as a complement to the PAM authentication, it is possible to store users in the NSO CDB database or externally authenticated. See [the section called “Authentication”](#) for details.

- We enable RESTCONF token authentication under `/ncs-config/aaa/external-validation` to use a `token_auth.sh` script that we uploaded earlier together with a `generate_token.sh` script.

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 `upgrade-12/app/yang/token.yang` in the reference demo example.

- The NSO web server HTTPS interface is enabled under `/ncs-config/webui`.



#### Note

The SSL certificates that get generated by NSO 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 used not 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.

- We enable the NSO SSH CLI login under `/ncs-config/cli/ssh/enabled`.

The NSO CLI style is set to C-style, the CLI prompt is modified to include the host name 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>
```

- The NSO HA is enabled under `/ncs-config/ha/enabled`.

Depending on your provisioning applications, you may want to turn `/ncs-config/rollback/enabled` off.

Rollbacks do not work that well with reactive-fastmap applications. If your application is a 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

Set up AAA using `aaa_init.xml`. As we saw in the previous sections, in addition to the Linux `root` user, we created the `admin` and `oper` users and the `ncsadmin` and `ncsoper` Linux user groups.

We enabled NSO to have PAM authenticate the users using pre-shared SSH keys without a passphrase for NSO CLI and NETCONF login, added password authentication for the `oper` user intended for NSO WebUI login, and token authentication for RESTCONF login.

The default AAA initialization file shipped with NSO resides under `/var/opt/ncs/cdb/aaa_init.xml`. If we are not happy with that, this is a good point in time to modify the initialization data for AAA.

The NSO daemon is still not running, and we have no existing CDB files. The default AAA configuration in the `aaa_init.xml` is restrictive and ok, so that we will keep it for this demo with only a minor modification.

Reviewing the `aaa_init.xml` file, we see that two groups are referred to in the NACM rule list, the `ncsadmin` and `ncsoper` groups that we created Linux user groups for in a previous section.

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 authorization.

- By 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.
- By not adding the `root` user to any group results in zero access as no NACM rule will match, and the default is to deny access.
- The one addition we make is to allow the `oper` user to execute the **generate-token** RPC for RESTCONF authentication using the token instead of basic user password authentication.

The recommended practice for clients that need to store passwords for authentication, here the manager station node, is to store them in a read-only file.

```
<rule-list>
  <name>oper</name>
  <group>ncsoper</group>
  <rule>
    <name>generate-token</name>
    <rpc-name>generate-token</rpc-name>
    <action>permit</action>
  </rule>
  ...
```

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.

Henceforth we will log 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.

## The High-Availability and VIP Configuration

This example sets up one HA pair, where we are using the built-in HA and the HCC package to manage virtual IP addresses. See the [the section called “NSO built-in HA”](#) and [the section called “Tail-f HCC Package”](#) for details.

Next, we will show a simple standard configuration of the built-in HA and HCC package and focus on issues when managing and upgrading an HA cluster.

The built-in HA and HCC packages provide us with three features:

- All CDB data becomes replicated from the *primary* to the *secondary* node.
- If the *primary* fails, the *secondary* takes over and starts to act as *primary*. I.e., the NSO built-in HA is configured to handle failover automatically.
- At failover, *tailf-hcc* sets up a virtual alias IP address on the *primary* node only and uses gratuitous ARP packets to update all nodes in the network with the new mapping to the *primary* node.

Nodes in other networks can be updated using the HCC layer-3 BGP functionality or a load balancer. See the NSO example set under `examples.ncs/development-guide/high-availability/hcc` for a reference to upgrading an HCC layer 2 enabled system.

Following the built-in HA documentation, we have the same HA configuration on the *paris* and *london* nodes.

We add the HA and HCC configuration to a `ha_init.xml` file loaded by NSO from the CDB directory when NSO is started.

We use the token that was shared between the HA group nodes earlier.

```
<config xmlns="http://tail-f.com/ns/config/1.0">
  <high-availability xmlns="http://tail-f.com/ns/ncs">
    <token>SHARED SECRET TOKEN HERE</token>
    <ha-node>
      <id>paris</id>
      <address>192.168.23.99</address>
      <nominal-role>primary</nominal-role>
    </ha-node>
    <ha-node>
      <id>london</id>
      <address>192.168.23.98</address>
      <nominal-role>secondary</nominal-role>
      <failover-primary>true</failover-primary>
    </ha-node>
    <settings>
      <start-up>
        <assume-nominal-role>true</assume-nominal-role>
        <join-ha>true</join-ha>
      </start-up>
      <enable-failover>true</enable-failover>
      <reconnect-interval>5</reconnect-interval>
      <reconnect-attempts>3</reconnect-attempts>
    </settings>
  </high-availability>
  <hcc xmlns="http://cisco.com/pkg/tailf-hcc">
    <enabled>true</enabled>
    <vip-address>192.168.23.122</vip-address>
  </hcc>
</config>
```

The *paris* node is given the *primary* role while the *london* node initializes to a *secondary* role.

Automatic failover from a *primary* node failure is enabled. We must take action for *secondary* node failures as the *primary* node will assume the *none* role when the *secondary* node connection is lost.

NSO generates *ha-primary-down* and *ha-secondary-down* alarms to allow the application to detect and take action. For example, if the link to the *secondary* node is lost and the *primary* node assumes the role

*none*, we can set the node back to *primary* and have the *secondary* node to reconnect once the issue is resolved.

The Tail-f HCC package is enabled with a VIP address and uses Layer 2 functionality. HCC always sets the VIP address on the *primary* node's matching interface and broadcasts a gratuitous ARP to announce the VIP to MAC mapping to the entire network. Here the manager node is located on the same network as the *paris* and *london* nodes.

After NSO started on both nodes and the initial configuration loaded, the last piece of the HA puzzle enables the node's built-in HA. To do so, we can, for example, execute the C-style CLI command **high-availability enable** on the two service nodes. Or, as we have each node do in this example, over MAAPI using the **ncs\_cmd** tool

```
$ ncs_cmd -u admin -g ncsadmin -o -c 'maction "/high-availability/enable"'
```

## 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. In particular, some of the timeouts.

- Device timeouts. NSO has connect, read, and write timeouts for traffic between NSO and the managed devices. The default value is 20 seconds for all. Some 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, best practice is to change the timeout per service from the service code using the Java `ServiceContext.setTimeout` method or the Python `data_set_timeout` method.

There are quite a few different global settings for NSO. The two mentioned above usually need to be changed.

## Initial Package Setup

- Next, we need to add the application packages. We generate the demo applications and RESTCONF authentication packages for this demo on both nodes.

While we usually copy the application packages to the NSO hosts, we here, for demo purposes, generate the dummy 1.0 + 1.1, inert 1.0, and token 1.0 packages from the YANG models and scripts we copied earlier to the *paris* and *london* nodes.

The application packages are added to the local `package-store` directory on both nodes before we initially copy the ones we use to the `${NCS_RUN_DIR}/packages` directory.

```
admin@paris:/app/package-store$ ncs-make-package --service-skeleton template \
--dest token-1.0 --no-test --root-container tokens token
admin@paris:/app/package-store$ cp ../yang/token.yang \
token-1.0/src/yang/token.yang
admin@paris:/app/package-store$ make -C token-1.0/src clean all
admin@paris:/app/package-store$ tar cvfz token-1.0.tar.gz token-1.0
...
```

We now start NSO on both nodes from the *admin* user and run the **/high-availability/enable** action to enable the HA group nodes with their nominal roles where *paris* is *primary* and *london* *secondary*.

```
admin@paris:/app/package-store$ ncs --cd ${NCS_RUN_DIR} \
```

```
--heart -c ${NCS_CONFIG_DIR}/ncs.conf
admin@paris:/app/package-store$ ncs_cmd -u admin -g ncsadmin -o \
-c 'maction "/high-availability/enable"'
admin@london:/app/package-store$ ncs --cd ${NCS_RUN_DIR} --heart \
-c ${NCS_CONFIG_DIR}/ncs.conf
admin@london:/app/package-store$ ncs_cmd -u admin -g ncsadmin -o \
-c 'maction "/high-availability/enable"'
```

**Note**

When installing new packages in run-time, after starting NSO, we can use the **software packages** commands.

For example, if we, instead of copying the token-1.0 package above to the `${NCS_RUN_DIR}/packages` directory before starting NSO, installed the token-1.0 package after starting NSO using the NSO CLI:

```
admin@nso-paris# software packages fetch package-from-file \
/path/to/package-store/token-1.0.tar.gz
admin@nso-paris# software packages install package token-1.0
admin@nso-paris# software packages list
package {
    name token-1.0.tar.gz
    loaded
}
...
```

- The packages that are actually running will reside under `/var/opt/ncs/state/packages-in-use.cur`

## Cisco Smart Licensing

NSO uses Cisco Smart Licensing, described in detail in [Chapter 3, Cisco Smart Licensing](#). After we have registered your NSO instance(s), and received a token, by following step 1-6 as described in the Create a License Registration Token section of [Chapter 3, Cisco Smart Licensing](#), we need to enter a token from our Cisco Smart Software Manager account on each host. We can use the same token for all instances and script entering the token as part of the initial NSO configuration or from the management station:

```
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.

## Verifying the Initial NSO Configuration

For ways to verify the initial configuration using the NSO CLI or RESTCONF interfaces, as mentioned earlier in this chapter, a reference to the container-based example for the layer-2 upgrade deployment scenario described here can be found in the NSO example set under `examples.ncs/development-guide/high-availability/hcc`.

## Log Management

You already covered some of the logging settings that can be set in `ncs.conf`. All `ncs.conf` settings are described in the man page for `ncs.conf`.

```
$ man ncs.conf
.....
```

## Log Rotate

The NSO system install that you have performed on your two hosts also installs good defaults for logrotate. Inspect `/etc/logrotate.d/ncs` and ensure that the settings are what you want. Note: 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

The *upgrade-l2* example, see reference from `examples.ncs/development-guide/high-availability/hcc`, integrates with **rsyslog** to log the ncs, developer, audit, netconf, snmp, and webui-access logs to syslog with facility set to *daemon* in `ncs.conf`. **rsyslogd** on the *london* and *paris* nodes 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/daemon.log` syslog.

## NED Logs

NED logs are a crucial tool for debugging NSO installations. These logs are very verbose and are for debugging only. Do not have these logs enabled in production.

Note that everything, including potentially sensitive data, is logged. No filtering is done. The NED trace logs are controlled through the CLI under: `/device/global-settings/trace`. It's 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, you want to use the *raw* setting.
- For all ConfD-based NETCONF devices, you want to use the *pretty* setting. This is because ConfD sends the NETCONF XML unformatted, pretty means that you get the XML formatted.
- For Juniper devices, you want to use the *raw* setting. Juniper sometimes sends 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, you want either *pretty* or *raw*.
- For SNMP-based devices, you want 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` `/var/log/ncs/ncs-java-vm.log`. The level of logging from Python code is controlled per Python 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* 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. Using the NSO RESTCONF interface is ideal for this. It is also recommended to set up a special read-only Linux user without shell access for this, like the *oper* user we set up earlier in this chapter. The **nct check** command can be used as a reference on what should be monitored. See `nct-check(1)` in *Manual Pages* for details.

## Alarms

The 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 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 *ha-secondary-down*, require operator invention. Thus, a monitoring tool should also *GET* the NSO alarm list.

```
$ curl -ik -H "X-Auth-Token: TsZTNwJZoYWBYPuOaMC6l4lCyXl+oDaasYqQZqqok=" \
https://paris:8888/restconf/data/ietf-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 changes 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 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. This is because they have full access to the system.
  - Users without shell access that are members of the *ncsadmin* Linux group have full access to the network. They have access to the NSO SSH shell and can execute some RESTCONF calls etc. However, they cannot manipulate backups and perform system upgrades. If you have provisioning systems north of NSO, it is recommended to assign a user of this type for those operations.
  - Users without shell access that are members of *ncsoper* Linux group have read-only access to the network. They can access the NSO SSH shell, execute arbitrary RESTCONF calls etc. However, they cannot 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.

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 port, configured through `/ncs-config/ncs-ipc-address`. For security, this port is typically limited to local connections and defaults to `127.0.0.1:4569`. The purpose of the port is to multiplex several different access methods to NSO.

The main security related point to make here is that *no AAA checks at all* are done on this port. If you have access to the port, 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 port and *tells* NSO who you are, NSO assumes that authentication is already performed. There is even a documented flag `--noaaa`, which tells NSO to skip all *NACM* rule checks for this session.

You must protect the port 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/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/ipc_access` file for example by changing group access to the file.

```
$ 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
.....
```





## CHAPTER 12

# Administration

---

- [User Management, page 153](#)
- [Packages, page 154](#)
- [Configuring NSO, page 157](#)
- [Monitoring NSO, page 157](#)
- [Backup and Restore, page 157](#)

## User Management

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

```
admin@ncs(config)# show full-configuration aaa authentication users user
aaa authentication users user admin
  uid      1000
  gid      1000
  password $1$GNwimSPV$E82za8AaDxukAi8Ya8eSR.
  ssh_keydir /var/ncs/homes/admin/.ssh
  homedir   /var/ncs/homes/admin
!
aaa authentication users user oper
  uid      1000
  gid      1000
  password $1$yOstEhXy$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  $4$wIo7Yd068FRwhYYI0d4IDw==
  !
  uimap oper
    remote-name      oper
    remote-password  $4$zp4zerM68FRwhYYI0d4IDw==
  !
!

```

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

```

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

```

## Packages

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

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

```

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

```

```

component cisco-ios
  ned cli ned-id cisco-ios
  ned cli java-class-name com.tailf.packages.ned.ios.IOSNedCli
  ned device vendor Cisco
NAME      VALUE
-----
show-tag  interface

oper-status up

```

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

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

```

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

```

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

## Adding and upgrading a package

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

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

```
$ ncs-netsim stop
```

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

```

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

```

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

```

admin@ncs# show packages packages package
cisco-ios ... admin@ncs# packages reload

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

```

```
>>> System upgrade has completed successfully.
reload-result {
  package cisco-ios
  result true
}
reload-result {
  package cisco-nx
  result true
}
```

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

## Simulating the new device

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

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

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

## Adding the new devices to 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

# Running NSO in Containers

- [Introduction, page 159](#)
- [Getting Started, page 159](#)
- [Administration, page 161](#)

## Introduction

NSO can be deployed to run in your production environment using a container, such as Docker. Cisco offers a pre-built, Red Hat UBI-based NSO image that you can readily use without needing to build the image environment. The [Red Hat UBI](#) is an [Open Container Initiative](#) (OCI)-compliant image that is freely distributable and independent of platform and technical dependencies.



### Note

This guide uses Docker as an example to use the NSO image, but you may choose another container runtime.

Running NSO in a container offers a number of 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.

## Getting Started

The Red Hat UBI-based NSO image is a pre-built production-ready image based on NSO System Install in *Getting Started* that you can run according to your deployment needs. Use the pre-built image as the base image in the container file (e.g., Dockerfile) and mount your own packages on top, such as NEDs and service packages, to run a final image for your production environment.

Consult the Installation guide in *Getting Started* for information concerning installing NSO on a Docker host, building NSO packages, and more.

## System Requirements

To run the image, make sure that your system meets the following requirements:

- An x86\_64 system running Linux (recommended) or macOS.
- A container platform. [Docker](#) is the recommended platform for running NSO.

**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 images when moving between a Linux machine and Docker on Mac as they both essentially run Docker on Linux.

## Running the Image

The following steps serve as general guidelines to run the NSO image. The CLI examples are applicable specifically to Docker. If you are running a non-Docker container, you basically need to fetch the NSO image, load/run the image with packages and networking, and finally log in to NSO CLI to run commands.

### Step 1 - Fetch the Image

Start by downloading the NSO base image:

- Step 1** Go to Cisco's official [Software Download](#) site.
- Step 2** Search for the product "Network Services Orchestrator" and select the desired version from the list.
- Step 3** The image is delivered as a signed package (e.g., `nso-6.1.container-image-prod.linux.x86_64.signed.bin`). Download it to your local disk.

**Note**

The signed archive file has the pattern `nso-VERSION.container-image-prod.OS.ARCH.signed.bin`, where, VERSION is the image's NSO version, OS is the operating system, and ARCH is the CPU architecture (only x86\_64 is supported currently).

- Step 4** Extract the image and other files from the signed package.

```
sh nso-6.1.container-image-prod.linux.x86_64.signed.bin
```

### Step 2 - Run the Image

Next, load the image and run it:

- Step 1** Start the container engine.
- Step 2** Navigate to the directory where you extracted the base image and load it. This will restore the image and its tag.
- ```
docker load -i nso-6.1.container-image-prod.linux.x86_64.tar.gz
```
- Step 3** Start a container from the image. Supply additional arguments to mount the packages/the `ncs.conf` as separate volumes (`-v flag`), and publish ports for networking (`-p flag`) as needed. The container starts NSO using the `/run-ncs.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 /data/nso:/nso -v /data/nso-logs:/log
-v /data/packages:/var/opt/ncs/packages/ --net=host
-e ADMIN_USERNAME=admin -e ADMIN_PASSWORD=admin cisco-nso-prod:6.1
```

Loading the packages by mounting the default load path `nso/run/packages` as a volume is the preferred way of loading the packages, but you can also load the packages by copying them manually into the `nso/run/packages`



directory. For optimal performance in your production environment, it is always recommended to mount the packages as a separate named volume instead of a bind mount 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 3 - Log in to NSO CLI

Finally, log in to the NSO CLI:

**Step 1**

Open an interactive shell on the running container.

```
docker exec -it cisco-nso bash
```

**Step 2**

Log in to NSO CLI.

```
/# ncs_cli -u admin
admin@ncs>
```

**Note**

You can also use `docker exec -it cisco-nso ncs_cli -u admin` to access the CLI from the host's terminal.

## Administration

This section covers the necessary administrative information.

### ncs.conf File Configuration and Preference

If the `ncs.conf` file is edited after startup, it can be reloaded using `MAAPI reload_config()`.  
Example: `$ ncs_cmd -c "reload"`.

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:

- The `ncs.conf` file specified in the Dockerfile (i.e., `ENV $NCS_CONFIG_DIR /etc/ncs/`) is used as the first preference.
- The second preference is to use the `ncs.conf` file mounted in the `/nso/etc/` run directory.
- 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.

## 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:6.1
```

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

Note that this only adds a user. If you are using a permanent volume for CDB, etc., and starting the NSO container multiple times with different `ADMIN_PASSWORD`, then the last run will effectively overwrite the older password. However, if you change `ADMIN_USERNAME` between invocations, then you will create multiple users. An admin user account created during the last run of NSO will not be removed just because `ADMIN_USERNAME` is set to a different value.



### 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.

## Exposed Ports

The following TCP ports are open by default.

- SSH: Port 22.
- HTTP: Port 80.
- HTTPS: Port 443.
- NETCONF: Port 830.
- NETCONF call-home: Port 4334.
- RESTCONF: Port 8888.

## Resolving Port Conflict

The container is configured to expose SSH on port 22. If an SSH server is run on the VM, there will be a conflict when configuring the NSO CLI to use the same port. To avoid port conflict, configure NSO to listen on a different port (e.g., 2024) in `ncs.conf` and then expose it with the **docker run** command using the **-p** or **--expose** flags, as **docker run -p 2024**. If you instead only want to expose the port to the localhost interface, run the command as **docker run -p 127.0.0.1:2024:2024**.



### Note

For logs, shared volumes are used. The `/log` folder inside the container contains the logs. Outside the container, you can access the logs in the `/data/nso-logs` directory, provided that the volume is configured.

## 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

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
```

### 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 of 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 and start a new one with the same persistent shared volume mounted but with a different command. Instead of running the `/run-nso.sh`, which is the normal command of the NSO container, run a command that keeps the container alive but also does not start NSO, for example `read DUMMY` (it is a bash builtin command so you still have to run bash). A full Docker command would look like this:

```
docker run -itd --name cisco-nso -v /data/nso:/nso -v /data/nso-logs:/log
cisco-nso-prod:6.1 bash -lc 'read DUMMY'
```

- Step 2** You now have the NSO container running but without NSO itself. Open a shell in the container with:

```
docker exec -it cisco-nso bash -l
```

- Step 3** Run the **ncs-backup restore** command, for example:

```
docker exec -it cisco-nso ncs-backup-restore
```

Or, if you want to automate the whole process slightly, you could do it all using **docker exec** and non-interactively:

```
docker exec -it cisco-nso bash -lc 'ncs-backup
restore /nso/run/backups/ncs-6.1.0@2023-03-07T14:41:02.backup.gz
--non-interactively'
```

**Step 4**

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, shut down your temporary container and start the regular NSO container again as usual.

## SSH Host Key

NSO looks for the SSH host key in the `/nso/ssh` directory. The filename differs depending on the configured host key algorithm. With Docker, NSO uses the ED25519 algorithm. If no SSH host key exists, one is generated. As the SSH key is stored in the `/nso` directory, which is typically a persistent shared volume in a production setup, it remains the same after restarts or upgrades of NSO.

## 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 production. 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"
```

## NSO Upgrade

This section 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 6.0 to 6.1.

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 `/data/nso` on the host machine running the container. This also makes all the CDB files available on the host in the `/data/nso/run/cdb` directory. At this point, the `/cdb` directory is empty.

```
docker run -itd --name cisco-nso -v /data/nso:/nso cisco-nso-prod:6.0
```

**Step 2**

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.

```
docker exec -it cisco-nso ncs backup
```

**Step 3**

Stop the NSO by issuing the following command, or by stopping the container itself which will run the **ncs stop** command automatically.

```
docker exec -it cisco-nso ncs stop
```

**Step 4**

Remove the old NSO

```
docker rm -f cisco-nso
```

**Step 5**

Start a new container and mount the `/nso` directory in the container to `/data/nso` on the host machine. 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 /data/nso:/nso cisco-nso-prod:6.1
```

---

At this point, you only have one container that is running the desired version 6.1 and you do not need to uninstall the old NSO.

## 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 production base image comes with a basic Docker 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.

---

