



NSO 4.4.2.3 Administration Guide

First Published: May 17, 2010

Last Modified: September 1, 2017

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: <http://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 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](#) 7

[Log messages and formats](#) 8

[Backup and restore](#) 16

[Backup](#) 16

[NSO Restore](#) 16

[Disaster management](#) 17

[NSO fails to start](#) 17

[NSO failure after startup](#) 18

[Transaction commit failure](#) 18

[Troubleshooting](#) 18

[Installation Problems](#) 19

[Problems Starting NSO](#) 19

[Problems Running Examples](#) 19

 General Troubleshooting Strategies 20

 CHAPTER 3

Cisco Smart Licensing 23

Introduction 23

Smart Accounts and Virtual Accounts 23

Request a Smart Account 23

Adding users to a Smart Account 25

Create a License Registration Token 26

Validation and Troubleshooting 29

Available Show Commands 29

Available Show Commands 29

 CHAPTER 4

NSO Alarms 31

Overview 31

Alarm type structure 31

Alarm type descriptions 32

 CHAPTER 5

NSO Packages 35

Package Overview 35

Loading Packages 35

Managing Packages 36

Package repositories 36

Actions 37

 CHAPTER 6

Advanced Topics 39

Locks 39

Global locks 39

Transaction locks 39

Northbound agents and global locks 40

External data providers 40

CDB 40

Lock impact on user sessions 41

IPC ports 41

Restricting access to the IPC port 42

Restart strategies for service manager 42

Security issues 42

Running NSO as a non privileged user	44
Using IPv6 on northbound interfaces	44

CHAPTER 7

NSO Clustering	47
NSO Clustering Introduction	47
Configuring and Running NSO in Cluster Mode	47
Initial Setup	49
Actions	50
Configuration Variations	51
Caveats	51

CHAPTER 8

High Availability	53
Introduction to NSO High Availability	53
HA framework requirements	55
Mode of operation	55
Security aspects	57
API	57
Ticks	57
Relay slaves	58
CDB replication	58

CHAPTER 9

The AAA infrastructure	61
The problem	61
Structure - data models	61
Data model contents	62
AAA related items in ncs.conf	62
Authentication	63
Public Key Login	64
Password Login	66
PAM	66
External authentication	67
Restricting the IPC port	68
Group Membership	69
Authorization	70
Command authorization	71
Rpc, notification, and data authorization	74

Authorization Examples	79
The AAA cache	81
Populating AAA using CDB	81
Hiding the AAA tree	81

CHAPTER 10

NSO Deployment	83
Introduction	83
Initial NSO installation	86
Initial NSO configuration - ncs.conf	87
Setting up AAA	89
Cisco Smart Licensing	90
Global settings and timeouts	90
Enabling SNMP	91
Loading the required NSO packages	92
Preparing the HA of the NSO installation	94
Handling tailf-hcc HA fallout	96
NSO slave host failure	96
NSO master host failure	96
Setting up the VIP or L3 anycast BGP support	97
Preparing the clustering of the NSO installation	98
Testing the cluster configuration	99
NSO system and packages upgrade	100
NSO major upgrade	100
NSO minor upgrade	102
Package upgrade	103
Patch management	104
Manual upgrade	104
Log management	106
Log rotate	106
NED logs	106
Java logs	106
Internal NSO log	107
Monitoring the installation	107
Alarms	107
Security considerations	107



CHAPTER 1

Introduction

Cisco Network Service Orchestrator enabled by Tail-f (NSO) version 4.4.2.3 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 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 also used to refer to the executable, the running daemon.



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 16](#)
- [Disaster management, page 17](#)
- [Troubleshooting, page 18](#)

Introduction

Cisco Network Service Orchestrator enabled by Tail-f (NSO) version 4.4.2.3 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 *NSO 4.4.2.3 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 Chapter 3, *NSO System Install* in *NSO Installation Guide*. 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: slave, master, relay-slave.
- *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:ESCVNFCatalogues
```

- *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 common 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 the **java-vm** settings in the CLI.
- *Python VMs*: NSO packages can be implemented in Python. The individual packages can be configured to run a VM each or share Python VM. Use the **show python-vm status current** to see current threads and **show python-vm status start** to see which threads where started at startup-time.

Logging

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

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

- *ncs.log* ncs daemon log. See [the section called “Log messages and formats”](#). Can be configured to syslog.
- *ncserr.log.1, ncserr.log.idx, ncserr.log.siz*: if the NSO daemon has a problem this contains debug information relevant for support. The content can be displayed with `"ncs --printlog ncserr.log"`.

- *audit.log* central audit log covering all northbound interfaces. See [the section called “Log messages and formats”](#) for formats. Can be configured to syslog.
- *localhost:8080.access*: all http requests to the daemon. This an access log for the embedded Web server. This file adheres to the Common Log Format, as defined by Apache and others. This log is not enabled by default and is not rotated, i.e. use logrotate(8).
- *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.
- *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.
- *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. To debug XPATH for a template, use the pipe-target debug in the CLI instead.
- *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.

syslog

NSO can syslog to a local or remote syslog server. The format can be BSD or IETF syslog format: RFC 5424. 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*.

Below follows an example of syslog configuration:

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

  <udp>
    <enabled>>false</enabled>
    <host>127.0.0.1</host>
    <port>895</port>
  </udp>

  <syslog-servers>
    <server>
      <host>127.0.0.2</host>
      <version>1</version>
    </server>
    <server>
      <host>127.0.0.3</host>
      <port>7900</port>
      <facility>local4</facility>
    </server>
  </syslog-servers>
</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	Format String	Comment
DAEMON_DIED	"Daemon ~s died"	An external database daemon closed its control socket.
DAEMON_TIMEOUT	"Daemon ~s timed out"	An external database daemon did not respond to a query.
NO_CALLPOINT	"no registration found for callpoint ~s of type=~s"	ConfD tried to populate an XML tree but no code had registered under the relevant callpoint.
CDB_DB_LOST	"CDB: lost DB, deleting old config"	CDB found it's data schema file but not it's data file. CDB recovers by starting from an empty database.
CDB_CONFIG_LOST	"CDB: lost config, deleting DB"	CDB found it's data files but no schema file. CDB recovers by starting from an empty database.
CDB_UPGRADE_FAILED	"CDB: Upgrade failed: ~s"	Automatic CDB upgrade failed. This means that the data model has been changed in a non-supported way.
CDB_INIT_LOAD	"CDB load: processing file: ~s"	CDB is processing an initialization file.
CDB_OP_INIT	"CDB: Operational DB re-initialized"	The operational DB was deleted and re-initialized (because of upgrade or corrupt file)
CDB_CLIENT_TIMEOUT	"CDB client (~s) timed out, waiting for ~s"	A CDB client failed to answer within the timeout period. The client will be disconnected.
INTERNAL_ERROR	"Internal error: ~s"	A ConfD internal error - should be reported to support@tail-f.com.
AAA_LOAD_FAIL	"Failed to load AAA: ~s"	Failed to load the AAA data, it could be that an external db is misbehaving or AAA is mounted/populated badly
EXTAUTH_BAD_RET	"External auth program (user=~s) ret bad output: ~s"	Authentication is external and the external program returned badly formatted data.

Symbol	Format String	Comment
BRIDGE_DIED	"confd_aaa_bridge died - ~s"	ConfD is configured to start the confd_aaa_bridge and the C program died.
PHASE0_STARTED	"ConfD phase0 started"	ConfD has just started its start phase 0.
PHASE1_STARTED	"ConfD phase1 started"	ConfD has just started its start phase 1.
STARTED	"ConfD started vsn: ~s"	ConfD has started.
UPGRADE_INIT_STARTED	"Upgrade init started"	In-service upgrade initialization has started.
UPGRADE_INIT_SUCCEEDED	"Upgrade init succeeded"	In-service upgrade initialization succeeded.
UPGRADE_PERFORMED	"Upgrade performed"	In-service upgrade has been performed (not committed yet).
UPGRADE_COMMITTED	"Upgrade committed"	In-service upgrade was committed.
UPGRADE_ABORTED	"Upgrade aborted"	In-service upgrade was aborted.
CONSULT_FILE	"Consulting daemon configuration file ~s"	ConfD is reading its configuration file.
STOPPING	"ConfD stopping (~s)"	ConfD is stopping (due to e.g. confd --stop).
RELOAD	"Reloading daemon configuration."	Reload of daemon configuration has been initiated.
BADCONFIG	"Bad configuration: ~s:~s: ~s"	confd.conf contained bad data.
WRITE_STATE_FILE_FAILED	"Writing state file failed: ~s: ~s (~s)"	Writing of a state file failed
READ_STATE_FILE_FAILED	"Reading state file failed: ~s: ~s (~s)"	Reading of a state file failed
SSH_SUBSYS_ERR	"ssh protocol subsystem - ~s"	Typically errors where the client doesn't properly send the \"subsystem\" command.
SESSION_LIMIT	"Session limit of type '~s' reached, rejected new session request"	Session limit reached, rejected new session request.
CONFIG_TRANSACTION_LIMIT	"Configuration transaction limit of type '~s' reached, rejected new transaction request"	Configuration transaction limit reached, rejected new transaction request.
ABORT_CAND_COMMIT	"Aborting candidate commit, request from user, reverting configuration."	Aborting candidate commit, request from user, reverting configuration.

Symbol	Format String	Comment
ABORT_CAND_COMMIT_TIMER	"Candidate commit timer expired, reverting configuration."	Candidate commit timer expired, reverting configuration.
ABORT_CAND_COMMIT_TERM	"Candidate commit session terminated, reverting configuration."	Candidate commit session terminated, reverting configuration.
ROLLBACK_REMOVE	"Found half created rollback0 file - removing and creating new"	Found half created rollback0 file - removing and creating new.
ROLLBACK_REPAIR	"Found half created rollback0 file - repairing"	Found half created rollback0 file - repairing.
ROLLBACK_FAIL_REPAIR	"Failed to repair rollback files."	Failed to repair rollback files.
ROLLBACK_FAIL_CREATE	"Error while creating rollback file: ~s: ~s"	Error while creating rollback file.
ROLLBACK_FAIL_RENAME	"Failed to rename rollback file ~s to ~s: ~s"	Failed to rename rollback file.
NS_LOAD_ERR	"Failed to process namespace ~s: ~s"	System tried to process a loaded namespace and failed.
NS_LOAD_ERR2	"Failed to process namespaces: ~s"	System tried to process a loaded namespace and failed.
FILE_LOAD_ERR	"Failed to load file ~s: ~s"	System tried to load a file in its load path and failed.
FILE_LOADING	"Loading file ~s"	System starts to load a file.
SKIP_FILE_LOADING	"Skipping file ~s: ~s"	System skips a file.
FILE_LOAD	"Loaded file ~s"	System loaded a file.
LISTENER_INFO	"~s to listen for ~s on ~s:~s"	ConfD starts or stops to listen for incoming connections.
NETCONF_HDR_ERR	"Got bad NETCONF TCP header"	The cleartext header indicating user and groups was badly formatted.
LIB_BAD_VSN	"Got library connect from wrong version (~s, expected ~s)"	An application connecting to ConfD used a library version that doesn't match the ConfD version (e.g. old version of the client library).
LIB_BAD_SIZES	"Got connect from library with insufficient keypath depth/keys support (~s/ ~s, needs ~s/~s)"	An application connecting to ConfD used a library version that can't handle the depth and number of keys used by the data model.

Symbol	Format String	Comment
LIB_NO_ACCESS	"Got library connect with failed access check: ~s"	Access check failure occurred when an application connected to ConfD.
SNMP_NOT_A_TRAP	"SNMP gateway: Non-trap received from ~s"	An UDP package was received on the trap receiving port, but it's not an SNMP trap.
SNMP_TRAP_V1	"SNMP gateway: V1 trap received from ~s"	An SNMP v1 trap was received on the trap receiving port, but forwarding v1 traps is not supported.
SNMP_TRAP_NOT_FORWARDED	"SNMP gateway: Can't forward trap from ~s; ~s"	An SNMP trap was to be forwarded, but couldn't be.
SNMP_TRAP_UNKNOWN_SENDER	"SNMP gateway: Not forwarding trap from ~s; the sender is not recognized"	An SNMP trap was to be forwarded, but the sender was not listed in confd.conf.
SNMP_TRAP_OPEN_PORT	"SNMP gateway: Can't open trap listening port ~s: ~s"	The port for listening to SNMP traps could not be opened.
SNMP_TRAP_NOT_RECOGNIZED	"SNMP gateway: Can't forward trap with OID ~s from ~s; There is no notification with this OID in the loaded models."	An SNMP trap was received on the trap receiving port, but its definition is not known
XPATH_EVAL_ERROR1	"XPath evaluation error: ~s for ~s"	An error occurred while evaluating an XPath expression.
XPATH_EVAL_ERROR2	"XPath evaluation error: '~s' resulted in ~s for ~s"	An error occurred while evaluating an XPath expression.
CANDIDATE_BAD_FILE_FORMAT	Bad format found in candidate db file ~s; resetting candidate"	The candidate database file has a bad format. The candidate database is reset to the empty database.
CANDIDATE_CORRUPT_FILE	"Corrupt candidate db file ~s; resetting candidate"	The candidate database file is corrupt and cannot be read. The candidate database is reset to the empty database.
MISSING_DES3CBC_SETTINGS	"DES3CBC keys were not found in confd.conf"	DES3CBC keys were not found in confd.conf
MISSING_AESCFB128_SETTINGS	"AESCFB128 keys were not found in confd.conf"	AESCFB128 keys were not found in confd.conf
SNMP_MIB_LOADING	"Loading MIB: ~s"	SNMP Agent loading a MIB file

Symbol	Format String	Comment
SNMP_CANT_LOAD_MIB	"Can't load MIB file: ~s"	The SNMP Agent failed to load a MIB file
SNMP_WRITE_STATE_FILE_FAILED	"Write state file failed: ~s: ~s"	Write SNMP agent state file failed
SNMP_READ_STATE_FILE_FAILED	"Read state file failed: ~s: ~s"	Read SNMP agent state file failed
SNMP_REQUIRES_CDB	"Can't start SNMP. CDB is not enabled"	The SNMP agent requires CDB to be enabled in order to be started.
FXS_MISMATCH	"Fxs mismatch, slave is not allowed"	A slave connected to a master where the fxs files are different
TOKEN_MISMATCH	"Token mismatch, slave is not allowed"	A slave connected to a master with a bad auth token
HA_SLAVE_KILLED	"Slave ~s killed due to no ticks"	A slave node didn't produce its ticks
HA_DUPLICATE_NODEID	"Nodeid ~s already exists"	A slave arrived with a node id which already exists
HA_FAILED_CONNECT	"Failed to connect to master: ~s"	An attempted library become slave call failed because the slave couldn't connect to the master
HA_BAD_VSN	"Incompatible HA version (~s, expected ~s), slave is not allowed"	A slave connected to a master with an incompatible HA protocol version
NETCONF	"~s"	NETCONF traffic log message
DEVEL_WEBUI	"~s"	Developer webui log message
DEVEL_AAA	"~s"	Developer aaa log message
DEVEL_CAPI	"~s"	Developer C api log message
DEVEL_CDB	"~s"	Developer CDB log message
DEVEL_CONFD	"~s"	Developer ConfD log message
DEVEL_SNMPGW	"~s"	Developer snmp GW log message
DEVEL_SNMPA	"~s"	Developer snmp agent log message
NOTIFICATION_REPLAY_STORE_FAILURE	"~s"	A failure occurred in the builtin notification replay store
EVENT_SOCKET_TIMEOUT	"Event notification subscriber with bitmask ~s timed out, waiting for ~s"	An event notification subscriber did not reply within the configured timeout period
EVENT_SOCKET_WRITE_BLOCK	"~s"	Write on an event socket blocked for too long time
COMMIT_UN_SYNCED_DEV	"Committed data towards device ~s which is out of sync"	Data was committed toward a device with bad or unknown sync state

Symbol	Format String	Comment
NCS_SNMP_INIT_ERR	"Failed to locate snmp_init.xml in loadpath ~s"	Failed to locate snmp_init.xml in loadpath
NCS_JAVA_VM_START	"Starting the NCS Java VM"	Starting the NCS Java VM
NCS_JAVA_VM_FAIL	"The NCS Java VM ~s"	The NCS Java VM failure/timeout
NCS_PACKAGE_SYNTAX_ERROR	"Failed to load NCS package: ~s; syntax error in package file"	Syntax error in package file
NCS_PACKAGE_DUPLICATE	"Failed to load duplicate NCS package ~s: (~s)"	Duplicate package found
NCS_PACKAGE_COPYING	"Copying NCS package from ~s to ~s"	A package is copied from the load path to private directory
NCS_PACKAGE_UPGRADE_ABORTED	"NCS package upgrade failed with reason '~s'"	The CDB upgrade was aborted implying that CDB is untouched. However the package state is changed
NCS_PACKAGE_BAD_NCS_VERSION	"Failed to load NCS package: ~s; requires NCS version ~s"	Bad NCS version for package
NCS_PACKAGE_BAD_DEPENDENCY	"Failed to load NCS package: ~s; required package ~s of version ~s is not present (found ~s)"	Bad NCS package dependency
NCS_PACKAGE_CIRCULAR_DEPENDENCY	"Failed to load NCS package: ~s; circular dependency found"	Circular NCS package dependency
CLI_CMD	"CLI '~s'"	User executed a CLI command.
CLI_DENIED	"CLI denied '~s'"	User was denied to execute a CLI command due to permissions.
BAD_LOCAL_PASS	"Provided bad password"	A locally configured user provided a bad password.
NO_SUCH_LOCAL_USER	"no such local user"	A non existing local user tried to login.
PAM_LOGIN_FAILED	"pam phase ~s failed to login through PAM: ~s"	A user failed to login through PAM.
PAM_NO_LOGIN	"failed to login through PAM: ~s"	A user failed to login through PAM
EXT_LOGIN	"Logged in over ~s using externalauth, member of groups: ~s~s"	An externally authenticated user logged in.

Symbol	Format String	Comment
EXT_NO_LOGIN	"failed to login using externalauth: ~s"	External authentication failed for a user.
GROUP_ASSIGN	"assigned to groups: ~s"	A user was assigned to a set of groups.
GROUP_NO_ASSIGN	"Not assigned to any groups - all access is denied"	A user was logged in but wasn't assigned to any groups at all.
MAAPI_LOGOUT	"Logged out from maapi ctx=~s (~s)"	A maapi user was logged out.
SSH_LOGIN	"logged in over ssh from ~s with authmeth:~s"	A user logged into ConfD's builtin ssh server.
SSH_LOGOUT	"Logged out ssh <~s> user"	A user was logged out from ConfD's builtin ssh server.
SSH_NO_LOGIN	"Failed to login over ssh: ~s"	A user failed to login to ConfD's builtin SSH server.
NOAAA_CLI_LOGIN	"logged in from the CLI with aaa disabled"	A user used the --noaaa flag to confd_cli
WEB_LOGIN	"logged in through Web UI from ~s"	A user logged in through the WebUI.
WEB_LOGOUT	"logged out from Web UI"	A Web UI user logged out.
WEB_CMD	"WebUI cmd '~s'"	User executed a Web UI command.
WEB_ACTION	"WebUI action '~s'"	User executed a Web UI action.
WEB_COMMIT	"WebUI commit ~s"	User performed Web UI commit.
SNMP_AUTHENTICATION_FAILED	"SNMP authentication failed: ~s"	An SNMP authentication failed.
LOGIN_REJECTED	"~s"	Authentication for a user was rejected by application callback.
COMMIT_INFO	"commit ~s"	Information about configuration changes committed to the running data store.
CLI_CMD_DONE	"CLI done"	CLI command finished successfully.
CLI_CMD_ABORTED	"CLI aborted"	CLI command aborted.
NCS_DEVICE_OUT_OF_SYNC	"NCS device-out-of-sync Device '~s' Info '~s'"	A check-sync action reported out-of-sync for a device
NCS_SERVICE_OUT_OF_SYNC	"NCS service-out-of-sync Service '~s' Info '~s'"	A check-sync action reported out-of-sync for a service
NCS_PYTHON_VM_START	"Starting the NCS Python VM"	Starting the NCS Python VM

Symbol	Format String	Comment
NCS_PYTHON_VM_FAIL	"The NCS Python VM ~s"	The NCS Python VM failure/timeout
NCS_SET_PLATFORM_DATA_ERROR	"NCS Device '~s' failed to set platform data Info '~s'"	The device failed to set the platform operational data at connect
NCS_SMART_LICENSING_START	"Starting the NCS Smart Licensing Java VM"	Starting the NCS Smart Licensing Java VM
NCS_SMART_LICENSING_FAIL	"The NCS Smart Licensing Java VM ~s"	The NCS Smart Licensing Java VM failure/timeout
NCS_SMART_LICENSING_GLOBAL_NOTIFICATION	"Smart Licensing Global Notification: ~s"	Smart Licensing Global Notification
NCS_SMART_LICENSING_ENTITLEMENT_NOTIFICATION	"Smart Licensing Entitlement Notification: ~s"	Smart Licensing Entitlement Notification
NCS_SMART_LICENSING_EVALUATION_COUNTDOWN	"Smart Licensing evaluation time remaining: ~s"	Smart Licensing evaluation time remaining
DEVEL_SLS	"~s"	Developer smartlicensing api log message
JSONRPC_REQUEST	"JSON-RPC: '~s' with JSON params ~s"	JSON-RPC method requested.
DEVEL_ECONFD	"~s"	Developer econfd api log message
CDB_FATAL_ERROR	"fatal error in CDB: ~s"	CDB encountered an unrecoverable error
LOGGING_STARTED	"Daemon logging started"	Logging subsystem started
LOGGING_SHUTDOWN	"Daemon logging terminating, reason: ~s"	Logging subsystem terminating
REOPEN_LOGS	"Logging subsystem, reopening log files"	Logging subsystem, reopening log files
OPEN_LOGFILE	"Logging subsystem, opening log file '~s' for ~s"	Indicate target file for certain type of logging
LOGGING_STARTED_TO	"Writing ~s log to ~s"	Write logs for a subsystem to a specific file
LOGGING_DEST_CHANGED	"Changing destination of ~s log to ~s"	The target logfile will change to another file
LOGGING_STATUS_CHANGED	"~s ~s log"	Notify a change of logging status (enabled/disabled) for a subsystem
ERRLOG_SIZE_CHANGED	"Changing size of error log (~s) to ~s (was ~s)"	Notify change of log size for error log

Symbol	Format String	Comment
CGI_REQUEST	"CGI: '~s' script with method ~s"	CGI script requested.
MMAP_SCHEMA_FAIL	"Failed to setup the shared memory schema"	Failed to setup the shared memory schema
KICKER_MISSING_SCHEMA	"Failed to load kicker schema"	Failed to load kicker schema
JSONRPC_REQUEST_IDLE_TIMEOUT	"Stopping session due to idle timeout: ~s"	JSON-RPC idle timeout.
JSONRPC_REQUEST_ABSOLUTE_TIMEOUT	"Stopping session due to absolute timeout: ~s"	JSON-RPC absolute timeout.
BAD_DEPENDENCY	"The dependency node '~s' for node '~s' in module '~s' does not exist"	A dependency was not found

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 *NSO 4.4.2.3 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 *NSO 4.4.2.3 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 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`.

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. This API can thus be used to manually reset the consistency state. Apart from this, the only way 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.

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.

- <http://www.lag.net/paramiko/>
- <http://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.

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.
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 <code>ncs.conf</code> . They are already enabled in all the examples.
Status	NSO will give you a comprehensive status report if you run <pre>\$ ncs --status</pre> <p>NSO status information is also available as operational data under <code>/ncs-state</code>.</p>
Check data provider	If you are implementing a data provider (for operational or configuration data), you can verify that it works for all possible data items using <pre>\$ ncs --check-callbacks</pre>
Debug dump	If you suspect you have experienced a bug in NSO, or NSO told you so, you can give Support a debug dump to help us diagnose the problem. It contains a lot of status information (including a full <code>ncs --status</code> report) and some internal state information. This information is only readable and comprehensible to the NSO development team, so send the dump to your support contact. A debug dump is created using <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>
Error log	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.
System dump	If NSO aborts due to failure to allocate memory (see the section called "Disaster management"), and you believe that this is due to a memory leak in NSO, creating one or more debug dumps as described above (before NSO aborts) will produce the most useful information for Support. If this is not possible, you can make NSO produce a system dump just before aborting. To do this, set the environment variable <code>\$NCS_DUMP</code> to a file

System call trace

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

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

Linux:

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

BSD:

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

Solaris:

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




CHAPTER 3

Cisco Smart Licensing

- [Introduction, page 23](#)
- [Smart Accounts and Virtual Accounts, page 23](#)
- [Validation and Troubleshooting, page 29](#)

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 User Guide](#).

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 Groudau, Benjamin Strickland, Burkhard Warning, James Ng, Jeffrey Smith, Joakim Grebeno, Marcus Bransell, mbransell@cisco.com, Cisco Systems, Inc., Virtual Account Administrator (1)

Create a License Registration Token

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

My Smart Account

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

Virtual Accounts

Virtual Account Name	Description
NSO	Tail-f

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

NSO

General | Users

* Name: NSO

Description: Tail-f

Current Default Virtual Account: DEFAULT

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

Save Reset

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

Smart Software Manager

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

Virtual Account: [NSO](#)

General | Licenses | Product Instances | Event Log

Virtual Account

Description: Tail-f

Default Virtual Account: No

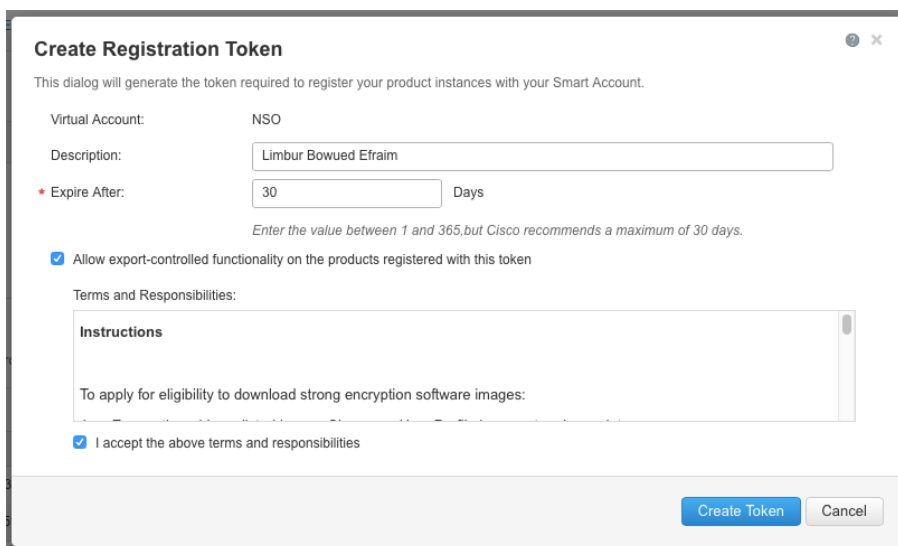
Product Instance Registration Tokens

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

[New Token...](#)

Token	Expiration Date	Description	Export-Controlled
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

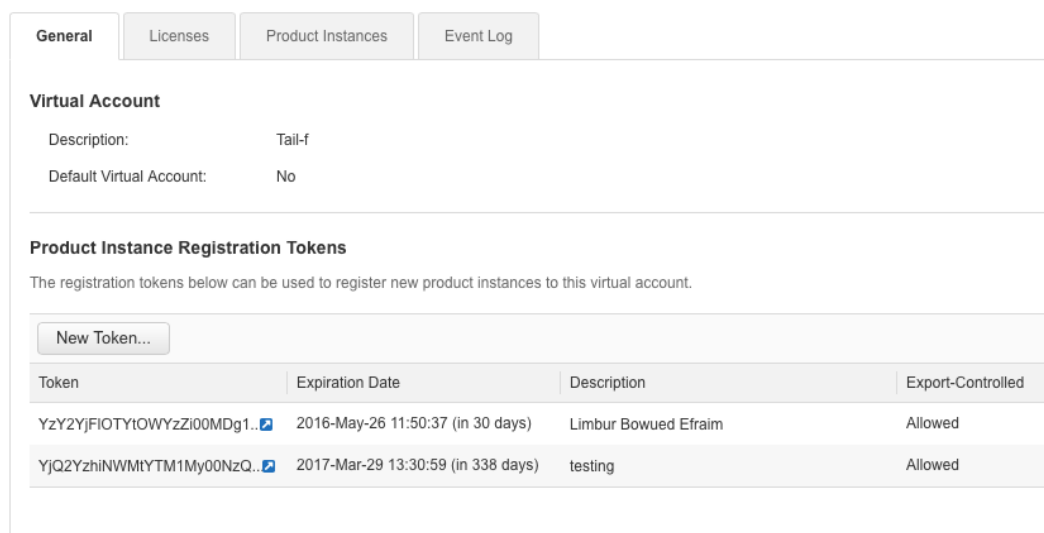
To apply for eligibility to download strong encryption software images:

☒ 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

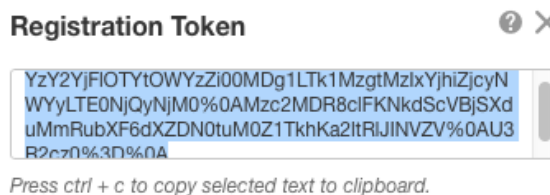
Product Instance Registration Tokens

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.
```

Validation and Troubleshooting

Available Show Commands

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

Available Show Commands

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



NSO Alarms

- [Overview, page 31](#)
- [Alarm type structure, page 31](#)
- [Alarm type descriptions, page 32](#)

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
  ncs-cluster-alarm
    cluster-subscriber-failure
  commit-through-queue-rollback-failed
  ncs-dev-manager-alarm
    ned-live-tree-connection-failure
    dev-manager-internal-error
    final-commit-error
    commit-through-queue-blocked
    bad-user-input
    abort-error
    revision-error
    missing-transaction-id
    configuration-error
    commit-through-queue-failed
    connection-failure
    out-of-sync
  ncs-snmpp-notification-receiver-alarm
    receiver-configuration-error
```

```

ncs-package-alarm
  package-load-failure
  package-operation-failure
ncs-service-manager-alarm
  service-activation-failure

```

Alarm type descriptions

- *alarm-type*: 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.
- *ncs-cluster-alarm*: Base type for all alarms related to cluster. This is never reported, sub-identities for the specific cluster alarms are used in the alarms.
- *cluster-subscriber-failure*: Failure to establish a notification subscription towards a remote node.
- *commit-through-queue-rollback-failed*: Automatic rollback of a commit-queue item failed.
- *ncs-dev-manager-alarm*: 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.
- *ned-live-tree-connection-failure*: NCS failed to connect to a managed device using one of the optional live-status-protocol NEDs. Verify the configuration of the optional NEDs. If the error occurs intermittently, increase connect-timeout.
- *dev-manager-internal-error*: An internal error in NCS device manager. It might happen due to unexpected device behaviour, or other unexpected condition. Report to the support team, provide NCS error log together with other available NCS logs.
- *final-commit-error*: A managed device validated a configuration change, but failed to commit. When this happens, NCS and the device are out of sync. Reconcile by comparing and sync-from or sync-to.
- *commit-through-queue-blocked*: 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.
- *bad-user-input*: Invalid input from user. NCS cannot recognize parameters needed to connect to device. Verify that the user supplied input are correct.
- *abort-error*: 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.
- *revision-error*: A managed device arrived with a known module, but too new revision. Upgrade the Device NED using the new YANG revision in order to use the new features in the device.
- *missing-transaction-id*: A device announced in its NETCONF hello message that it supports the transaction-id as defined in <http://tail-f.com/yang/netconf-monitoring>. However when NCS tries to read the 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.
- *configuration-error*: Invalid configuration of NCS managed device, NCS cannot recognize parameters needed to connect to device. Verify that the configuration parameters defined in tailf-ncs-devices.yang submodule are consistent for this device.
- *commit-through-queue-failed*: A queued commit failed. Resolve with rollback if possible.

- *connection-failure*: NCS failed to connect to a managed device before the timeout expired. Verify address, port, authentication, check that the device is up and running. If the error occurs intermittently, increase connect-timeout.
- *out-of-sync*: 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. Inspect the difference with compare-config, reconcile by invoking sync-from or sync-to.
- *ncs-snmp-notification-receiver-alarm*: The snmp-notification-receiver could not setup its configuration, either at startup or when reconfigured. Snmp notifications will now be missed. Check the error-message and change the configuration.
- *receiver-configuration-error*: An SNMP Notification Receiver configuration error prevented it from starting.
- *ncs-package-alarm*: Base type for all alarms related to packages. This is never reported, sub-identities for the specific package alarms are used in the alarms.
- *package-load-failure*: NCS failed load a package. Check the package for the reason.
- *package-operation-failure*: A package has some problem with its operation. Check the package for the reason.
- *ncs-service-manager-alarm*: 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.
- *service-activation-failure*: A service failed during re-deploy. Corrective action and another re-deploy is needed.



CHAPTER 5

NSO Packages

- [Package Overview, page 35](#)
- [Loading Packages, page 35](#)
- [Managing Packages, page 36](#)

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 is a controlled way to manage loading and versions of custom applications.

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. By default, the `reload` action will (when needed) wait up to 10 seconds for this to happen, and 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.

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 two different 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.

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.

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 dependant 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 2. 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 39](#)
- [IPC ports, page 41](#)
- [Restart strategies for service manager, page 42](#)
- [Security issues, page 42](#)
- [Running NSO as a non privileged user, page 44](#)
- [Using IPv6 on northbound interfaces, page 44](#)

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 9, *Package Development* in *NSO 4.4.2.3 Development* to be able to understand how the different locks fit into the picture.

Global locks

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

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

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

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

Transaction locks

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

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

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

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

Northbound agents and global locks

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

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

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

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

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

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

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

External data providers

Implementing the `lock()` and `unlock()` callbacks is not required of an external data provider. NSO will never try to initiate the `transLock()` state transition (see the transaction state diagram in Chapter 9, *Package Development in NSO 4.4.2.3 Development*) 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 an 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`.

IPC ports

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

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

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

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

There are two more instances of ports that will have to be modified, NETCONF and CLI over SSH. The netconf (SSH and TCP) ports that NCS 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 NCS 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 NCS 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.

Failing that, all programs will be executed as the user running the **confd** 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 NCS 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 5, *The NSO SNMP Agent in NSO 4.4.2.3 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.



NSO Clustering

- [NSO Clustering Introduction, page 47](#)
- [Configuring and Running NSO in Cluster Mode, page 47](#)
- [Caveats, page 51](#)

NSO Clustering Introduction

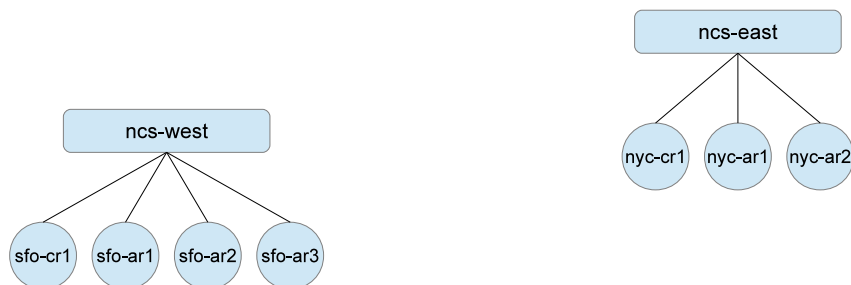
The NSO Clustering feature essentially makes it possible for a Device Manager on one NSO to connect and communicate with a Device Manager on another NSO. An NSO cluster consists of loosely coupled, yet tightly integrated set of co-operating NSO nodes.

When cluster mode is enabled it is possible to add devices using `/devices/device/remote-node` (instead of `address/port`). A device added in such a way does not store its configuration on that local NSO node, instead that node will query the remote node whenever configuration data is read. In fact, all operations towards that device (read/write of config/stats, as well as actions) are delegated to the remote NSO node in question.

Designing a cluster deployment requires planning. There can be many reasons to use clustering: scaling (handling more devices than a single node can handle), partitioning considerations (for example delegating Device Manager responsibility according to geographical or network topological considerations). Another reason can be to spread out services (services can be made to run on different cluster nodes)

Configuring and Running NSO in Cluster Mode

To illustrate how to work with an NSO cluster, a simple network is assumed. There are two NSO nodes (east and west), each handling three devices each. Like this:



Initial network

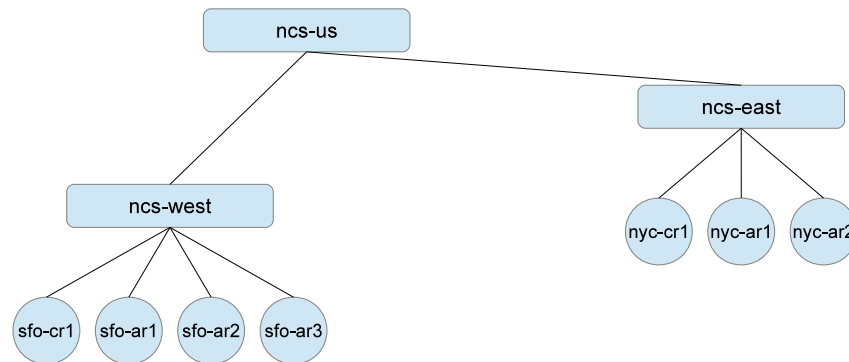
Logging in to these remote NSO:s we can examine the setup further:

```

admin@ncs-east# show running-config devices device | select address
devices device nyc-ar1
    address 10.10.34.17;
!
devices device nyc-ar2
    address 10.10.35.18;
!
devices device nyc-cr1
    address 10.20.1.3;
!

admin@ncs-west# show running-config devices device | select address
devices device sfo-ar1
    address 10.30.5.33;
!
devices device sfo-ar2
    address 10.30.9.42;
!
devices device sfo-ar3
    address 10.30.17.23;
!
devices device sfo-cr1
    address 10.20.2.5;
!
  
```

Now let us add an NSO which will be able to control all of the devices by using the cluster setup to communicate with ncs-west and ncs-east, illustrated like this:



Adding a cluster node, ncs-us

Initial Setup

Assuming we have the new NSO node, ncs-us, up and running we need to first add the two other nodes to its configuration. The very first step is to set up authentication configuration. NSO uses NETCONF over SSH to communicate to other nodes in the cluster, the simplest way to set up authentication is to have NSO pass on the credentials one has logged in with on to the next NSO.

```

admin@ncs-us(config)# cluster authgroup ncs-us ?
Possible completions:
  default-map - Remote authentication parameters for users not in umap
  umap        - Map NCS users to remote authentication parameters

admin@ncs-us(config)# cluster authgroup ncs-us default-map ?
Possible completions:
  remote-name      - Specify node user name
  remote-password  - Specify the remote password
  same-pass        - Use the local NCS user name as the remote user name
  same-user        - Use the local NCS user name as the remote user name

admin@ncs-us(config)# cluster authgroup ncs-us default-map same-user same-pass

admin@ncs-us(config)# cluster remote-node ncs-west
Value for 'authgroup' [default,ncs-us]: ncs-us

admin@ncs-us(config)# cluster remote-node ncs-east address 10.50.0.2 authgroup ncs-us

admin@ncs-us(config)# commit
Commit complete.

That is all that is required as far as initial configuration goes. The next step is to add the remote devices to ncs-us.

admin@ncs-us(config)# devices device nyc-ar1 remote-node ncs-east

```

```

admin@ncs-us(config)# devices device nyc-ar2 remote-node ncs-east
admin@ncs-us(config)# devices device nyc-cr1 remote-node ncs-east
admin@ncs-us(config)# devices device sfo-ar1 remote-node ncs-west
admin@ncs-us(config)# devices device sfo-ar2 remote-node ncs-west
admin@ncs-us(config)# devices device sfo-ar3 remote-node ncs-west
admin@ncs-us(config)# devices device sfo-cr1 remote-node ncs-west

admin@ncs-us(config)# commit
Commit complete.

admin@ncs-us(config)# exit

admin@ncs-us# show cluster remote-node
NAME      NAME
-----
ncs-east  nyc-ar1
          nyc-ar2
          nyc-cr1
ncs-west  sfo-ar1
          sfo-ar2
          sfo-ar3
          sfo-cr1

```

At this point the `ncs-us` is connected to the other NSO cluster node, and it is possible to for example inspect and edit the configuration of a remote device.

Actions

Actions that concerns a single device will be forwarded to the cluster node which is directly communicating with the device. Thus when running for example `sync-from` on `ncs-us`:

```

admin@ncs-us# devices device nyc-cr1 sync-from
result true

```

`ncs-us` will seamlessly forward the request to `ncs-east`.

In the case of actions that work on multiple devices, the local node will send out the request, in parallel, to the involved cluster nodes, capture the result and present it like it was performed on the local NSO node. For example:

```

admin@ncs-us# devices check-sync
sync-result {
    device nyc-ar1
    result in-sync
}
sync-result {
    device nyc-ar2
    result in-sync
}
sync-result {
    device nyc-cr1
    result in-sync
}
sync-result {
    device sfo-ar1
    result in-sync
}

```



```

    }
    sync-result {
        device sfo-ar2
        result in-sync
    }
    sync-result {
        device sfo-ar3
        result in-sync
    }
    sync-result {
        device sfo-cr1
        result in-sync
    }
}

```

Configuration Variations

There is no limitation on whether an NSO cluster node can have either local or remote devices, both is fine. It is also possible to have a subset of a remote nodes devices.

For example an NSO called `us-ar` could just have the `*-ar*` devices added. This setup is useful if there are services that only span a particular set of devices.

Another variation is the "side by side" configuration, where two or more cluster nodes have some devices local, and the rest of the devices configured as remote. This kind of setup allows all NSO cluster nodes to see all devices, but only require them to directly communicate with the local ones.

Caveats

In order to have a functional cluster there are a couple of things that need to be correctly set up, and there are some limitations.

- A node in a cluster must have the same (or a super-set) of the *Device NEDs* that the remote node has. (To ensure that the device YANG modules are loaded).
- When a device is configured as being on a remote-node, that remote cluster node must be accessible in order for the local NSO node to be able to do anything.
- There is no replication of configuration data or operational data whatsoever between nodes in a cluster. It is however possible to combine the NSO cluster feature with the NSO High Availability functionality (see [Chapter 8, High Availability](#)). A remote node can for example be a fully replicated HA pair. In this case it is important to configure the address of the remote node as the reachable address of the HA group (for example using a virtual/floating IP address or DNS redundancy).
- There is currently no way to forward NETCONF notifications, or NSO alarms from one cluster node to another.
- When using Commit Queues (see the section called “Commit Queue” in *NSO 4.4.2.3 User Guide*) together with clustering, each remote NSO has its own commit-queue. There is currently no way of monitoring, or controlling a commit queue on a remote cluster node.



CHAPTER 8

High Availability

- [Introduction to NSO High Availability, page 53](#)
- [HA framework requirements, page 55](#)
- [Mode of operation, page 55](#)
- [Security aspects, page 57](#)
- [API, page 57](#)
- [Ticks, page 57](#)
- [Relay slaves, page 58](#)
- [CDB replication, page 58](#)

Introduction to NSO High Availability

NSO support replication of the CDB configuration as well as of the operational data kept in CDB. The replication architecture is that of one active master and a number of passive slaves.

A group of NCS hosts consisting of a master, and one or more slaves, is referred to as an HA group (and sometimes as an HA cluster - however this is completely independent and separate from the clustering feature described in [Chapter 7, NSO Clustering](#)).

All configuration write operations must occur at the master and NSO will automatically distribute the configuration updates to the set of live slaves. 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 master, with the updates distributed to the live slaves, whereas non-replicated operational data can also be written on the slaves.

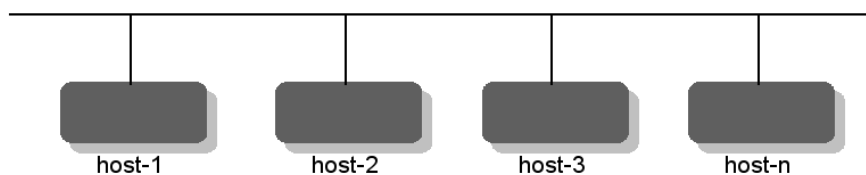
The *only* thing NSO does is to replicate the CDB data amongst the members in the HA group. It doesn't perform any of the otherwise High-Availability related tasks such as running election protocols in order to elect a new master. This is the task of a High-Availability Framework (HAFW) which must be in place. The HAFW must instruct NSO which nodes are up and down using methods from *Ha* class in the Java library.

Alternatively, if we do not wish to run NCS with a full scale HA framework, we can use the NCS package in `$NCS_DIR/examples.ncs/web-server-farm/ha/packages/manual-ha` to manually control a small HA pair of two (2) nodes and manually indicate which node is master and which is slave. If the master dies, the operator will have to manually tell the slave, that it is now master.

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.



Master - Slave configuration



One master - several slaves

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 3, “A data model divided into common and node specific subtrees”](#).

Example 3. 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;
        }
        ...
    }
}

```

HA framework requirements

NSO only replicates the CDB data. NSO must be told by the HAFW which node should be master and which nodes should be slaves.

The HA framework must also detect when nodes fail and instruct NSO accordingly. If the master node fails, the HAFW must elect one of the remaining slaves and appoint it the new master. The remaining slaves must also be informed by the HAFW about the new master situation. NSO will never take any actions regarding master/slave-ness by itself.

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>
  <tick-timeout>PT20S</tick-timeout>
</ha>

```

The IP address and the port above indicates which IP and which port should be used for the communication between the HA nodes. The `tick-timeout` is a duration indicating how often each slave must send a tick message to the master indicating liveness. If the master has not received a tick from a slave within 3 times the configured tick time, the slave is considered to be dead. Similarly, the master sends tick messages to all the slaves. If a slave has not received any tick messages from the master within the 3 times the timeout, the slave will consider the master dead and report accordingly.

A HA node can be in one of three states: NONE, SLAVE or MASTER. 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 slave or a master the HAFW must invoke either the methods `Ha.beSlave(master)` or `Ha.beMaster()`

When a NSO HA node starts, it always starts up in mode NONE. This is consistent with how NSO works without HA enabled. 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.

At some point, the HAFW will command some nodes to become slave nodes of a named master node. When this happens, each slave node tracks changes and (logically or physically) copies all the data from the master. Previous data at the slave 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 SLAVE 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 slave node. Once a node is returned either to the NONE state or to the MASTER state, write transactions can once again be initiated towards the node.

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

Each of the master and slave 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 slave nodes to the master, and before replication starts. The associated callbacks will however only be executed at the master. If e.g. the validation executing at the master 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 master, the node will start to listen for incoming slaves at the `ip:port` configured under `/ncs-config/ha`. The slaves TCP connect to the master and this socket is used by NSO to distribute the replicated data.

If the order is to be a slave, the node will contact the master and possibly copy the entire configuration from the master. This copy is not performed if the master and slave 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 master each time a configuration change is committed, and all nodes write the same transaction id into their copy of the committed configuration. If the master dies, and one of the remaining slaves is appointed new master, the other slaves must be told to connect to the new master. They will compare their last transaction id to the one from the newly appointed master. 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 master and the remaining slaves will still have the last transaction id generated by the old master - the new master will not generate a new transaction id until a new configuration change is committed. The same mechanism works if a slave 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 master, it is not possible for an agent to commit write operations at a slave node.

When an agent commits its CDB data, CDB will stream the committed data out to all registered slaves. If a slave dies during the commit, nothing will happen, the commit will succeed anyway. When and if the slave reconnects to the cluster, the slave will have to copy the entire configuration. All data on the HA sockets between NSO nodes only go in the direction from the master to the slaves. A slave which isn't reading its data will eventually lead to a situation with full TCP buffers at the master. In principle it is the responsibility of HAFW to discover this situation and notify the master NSO about the hanging slave. 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 master 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 slave 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 master or at any of the slaves. Once a slave has received the updates associated to a commit at the master, all CDB subscribers at the slave 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 slave(s) will have all subscriptions as configured in the system, but the subscription will be idle. All NETCONF notifications are

handled by the master, and once the notifications get written into stable storage (CDB) at the master, the list of received notifications will be replicated to all slaves.

Security aspects

We specify in `nsc.conf` which IP address the master should bind for incoming slaves. 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 slave node that connects to a master node negotiates with the master using a CHAP-2 like protocol, thus both the master and the slave 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 slave to a master 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-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-cfb-128-encrypted-string` type for that data. If the configuration data is of type `tailf:aes-cfb-128-encrypted-string` the encrypted data will be sent on the wire in update messages from the master to the slaves.

API

There are two APIs used by the HA framework to control the replication aspects of NCS. First there exists a synchronous API used to tell NCS what to do, secondly the application may create a notifications socket and subscribe to HA related events where NCS notifies the application on certain HA related events such as the loss of the master etc. The HA related notifications sent by NCS 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 `/nsc-cfg/ha/tick-timeout` is by default set to 20 seconds. This means that every 20 seconds each slave will send a tick message on the socket leading to the master. Similarly, the master will send a tick message every 20 seconds on every slave 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 master to the slaves. 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 slave or, if it is a slave that loose contact with the master, go into the initial `NONE` state.

If the HAFW can be really trusted, it is possible to set this timeout to `P0S`, i.e zero, in which case the entire dead-node-detection mechanism in NSO is disabled.

Relay slaves

The normal setup of a NSO HA cluster is to have all slaves connected directly to the master. 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 slaves at a location that is network-wise distant from the master, it can however be sub-optimal, since the replicated data will be sent to each remote slave individually over a potentially low-bandwidth network connection.

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

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

This setup requires special consideration by the HAFW. Instead of just telling each slave to connect to the master independently, it must setup the slaves that are intended to be relays, and tell them to become relays, before telling the "sub-slaves" to connect to the relay slaves. Consider the case of a master M and a slave S0 in one location, and two slaves 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 master.
- 2 Tell S0 and S1 to be slave with M as master.
- 3 Tell S1 to be relay.
- 4 Tell S2 to be slave with S1 as master.

Conversely, the handling of network outages and node failures must also take the relay slave setup into account. For example, if a relay slave loses contact with its master, it will transition to the NONE state just like any other slave, and it will then disconnect its "sub-slaves" which will cause those to transition to NONE too, since they lost contact with "their" master. Or if a relay slave dies in a way that is detected by its "sub-slaves", 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 slave of M instead (and when S1 comes back, perhaps tell S2 to be a relay and S1 to be a slave of S2).

Besides the use of `Ha.beRelay()`, the API is mostly unchanged when using relay slaves. The HA event notifications reporting the arrival or the death of a slave are still generated only by the "real" cluster master. If the `Ha.HaStatus()` method is used towards a relay slave, it will report the node state as `SLAVE_RELAY` rather than just `SLAVE`, and the array of nodes will have its master as the first element (same as for a "normal" slave), followed by its "sub-slaves" (if any).

CDB replication

When HA is enabled in `ncs.conf` CDB automatically replicates data written on the master to the connected slave nodes. Replication is done on a per-transaction basis to all the slaves 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 slave mode the northbound APIs are in read-only mode, that is the configuration can not be changed on a slave other than through replication updates from the master. It is still possible to read from for example NETCONF or CLI (if they are enabled) on a slave. 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 slaves similar to how configuration data is replicated. Operational data is always replicated asynchronously, regardless of the `/ncs-config/cdb/operational/replication` setting.



The AAA infrastructure

- [The problem, page 61](#)
- [Structure - data models, page 61](#)
- [AAA related items in ncs.conf, page 62](#)
- [Authentication, page 63](#)
- [Restricting the IPC port, page 68](#)
- [Group Membership, page 69](#)
- [Authorization, page 70](#)
- [The AAA cache, page 81](#)
- [Populating AAA using CDB, page 81](#)
- [Hiding the AAA tree, page 81](#)

The problem

This chapter describes how to use NSO's built-in authentication and authorization mechanisms. Users log into NSO through the CLI, NETCONF, 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.

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 6536](#)) as augmented by `tailf-acm.yang` is used for group assignment and authorization. For backwards compatibility, it is alternatively possible to use the older revision 2011-09-22 of `tailf-aaa.yang` for all of authentication, group assignment, and authorization. This legacy version of `tailf-`

aaa can be found in the `$NCS_DIR/src/ncs/aaa` directory, but its usage is not further described here. Detailed information about this can be found in versions of this document for NCS-3.3 and earlier.

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.

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 NCS 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.

If we always authenticate users through PAM it may make sense to set this configurable to `false`. If we disable local authentication

```
/ncs-config/aaa/pam
```

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
```

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

Authentication

Depending on northbound management protocol, when a user session is created in NCS, it may or may not be authenticated. If the session is not yet authenticated, NCS's AAA subsystem is used to perform authentication and authorization, as described below. If the session already has been authenticated, NCS'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 *NSO 4.4.2.3 Northbound APIs*.
- When NETCONF uses TCP, as described in the section called “NETCONF Transport Protocols” in *NSO 4.4.2.3 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 NCS 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 5, *The NSO SNMP Agent* in *NSO 4.4.2.3 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-publickey-authentication` and `/ncs-config/aaa/local-authentication/enabled`.

- 1 If `ssh-publickey-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-key-authentication` is set to `system`, `local-authentication` is enabled, and the SSH keys in `/aaa/authentication/users/user{$USER}/ssh_keydir` match the keys presented by the user, authentication succeeds.
- 3 Otherwise, if `ssh-key-authentication` is set to `system` and the user `/aaa/authentication/users/user{$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 and RSA 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$nG1MYlZpQ0bzenyFOQI3L1</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`

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 *NSO 4.4.2.3 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.

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 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`.

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".

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.

Restricting the IPC port

NCS listens for client connections on the NCS 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 NCS deployment ends up in either of two cases, untrusted users have, or have not shell access to the host(s) where NCS 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 <code>cli</code> , <code>webui</code> , or <code>*</code> 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 <code>cli</code> whereas for operations via the Web UI, the context will be the string <code>webui</code> .
command	<p>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 <code>request system reboot</code>. 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 <code>"*"</code> matches any command.</p> <p>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.</p>
access-operations	<p>The individual tokens can be POSIX extended regular expressions. Each regular expression is implicitly anchored, i.e. an <code>"^"</code> is prepended and a <code>"\$"</code> is appended to the regular expression.</p> <p><code>access-operations</code> is used to match the operation that NSO tries to perform. It must be one or both of the <code>"read"</code> and <code>"exec"</code> 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.</p>

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 leaves:

```

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 the rule applies to. The special value * matches all modules.
rpc-name / notification-name / path	This is a choice between three possible leafs that are used for matching: <div> <div>rpc-name</div> <div>The name of a rpc operation, or "*" to match any rpc.</div> <div>notification-name</div> <div>The name of a notification, or "*" to match any notification.</div> <div>path</div> <div>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. <ol style="list-style-type: none"> 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. </div> </div>

- 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.
access-operations	Similarly to command rules we can differentiate access depending on which agent is used to gain access.
	<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 leafs, 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 leafs 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 sub tree. 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.

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>
    <module-name>tailf-aaa</module-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.

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>
    <module-name>tailf-aaa</module-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.

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 *NSO 4.4.2.3 Manual Pages* manual page, or by triggering a subscription notification by using the "subscription lock" when updating the CDB operational data store, see Chapter 5, *CDB - The NSO Configuration Database* in *NSO 4.4.2.3 Development*.

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

NSO Deployment

- [Introduction, page 83](#)
- [Initial NSO installation, page 86](#)
- [Initial NSO configuration - ncs.conf, page 87](#)
- [Setting up AAA, page 89](#)
- [Cisco Smart Licensing, page 90](#)
- [Global settings and timeouts, page 90](#)
- [Enabling SNMP, page 91](#)
- [Loading the required NSO packages, page 92](#)
- [Preparing the HA of the NSO installation, page 94](#)
- [Handling tailf-hcc HA fallout, page 96](#)
- [Preparing the clustering of the NSO installation, page 98](#)
- [Testing the cluster configuration, page 99](#)
- [NSO system and packages upgrade, page 100](#)
- [Log management, page 106](#)
- [Monitoring the installation, page 107](#)
- [Security considerations, page 107](#)

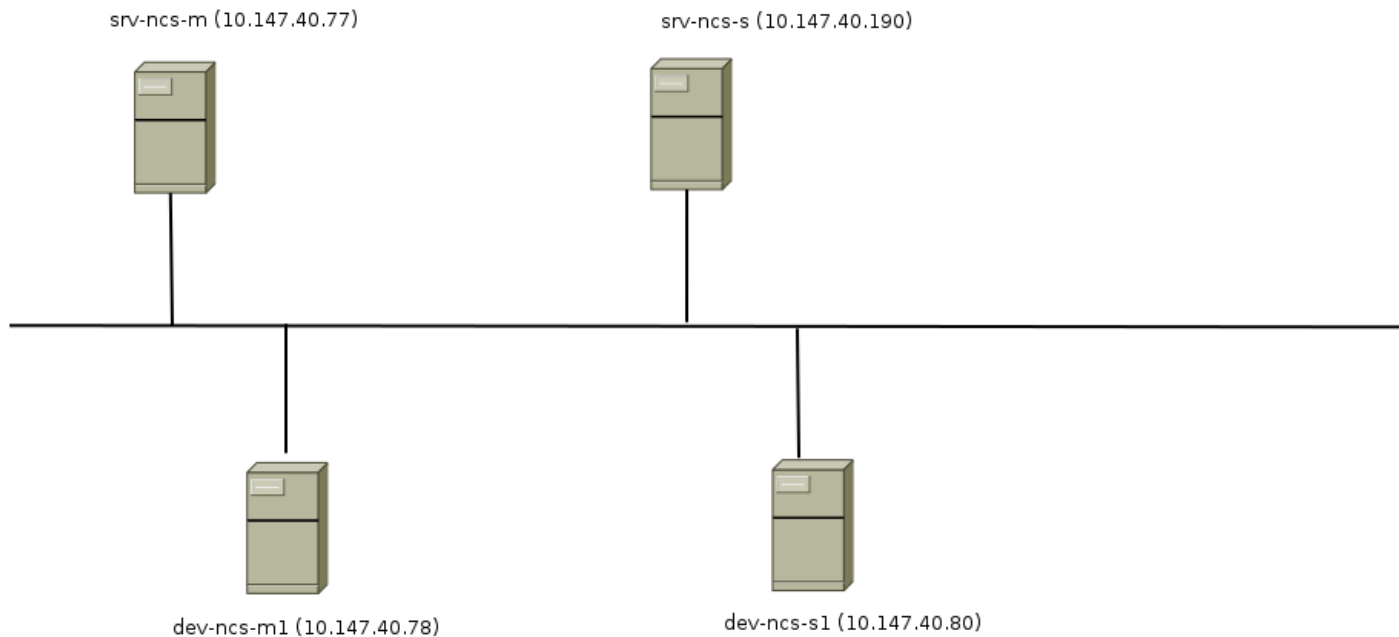
Introduction

This chapter is written as a series of examples. We'll be describing a typical large scale deployment where the following topics will be covered:

- 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
- Trouble shooting, backups and disaster recovery
- Security considerations

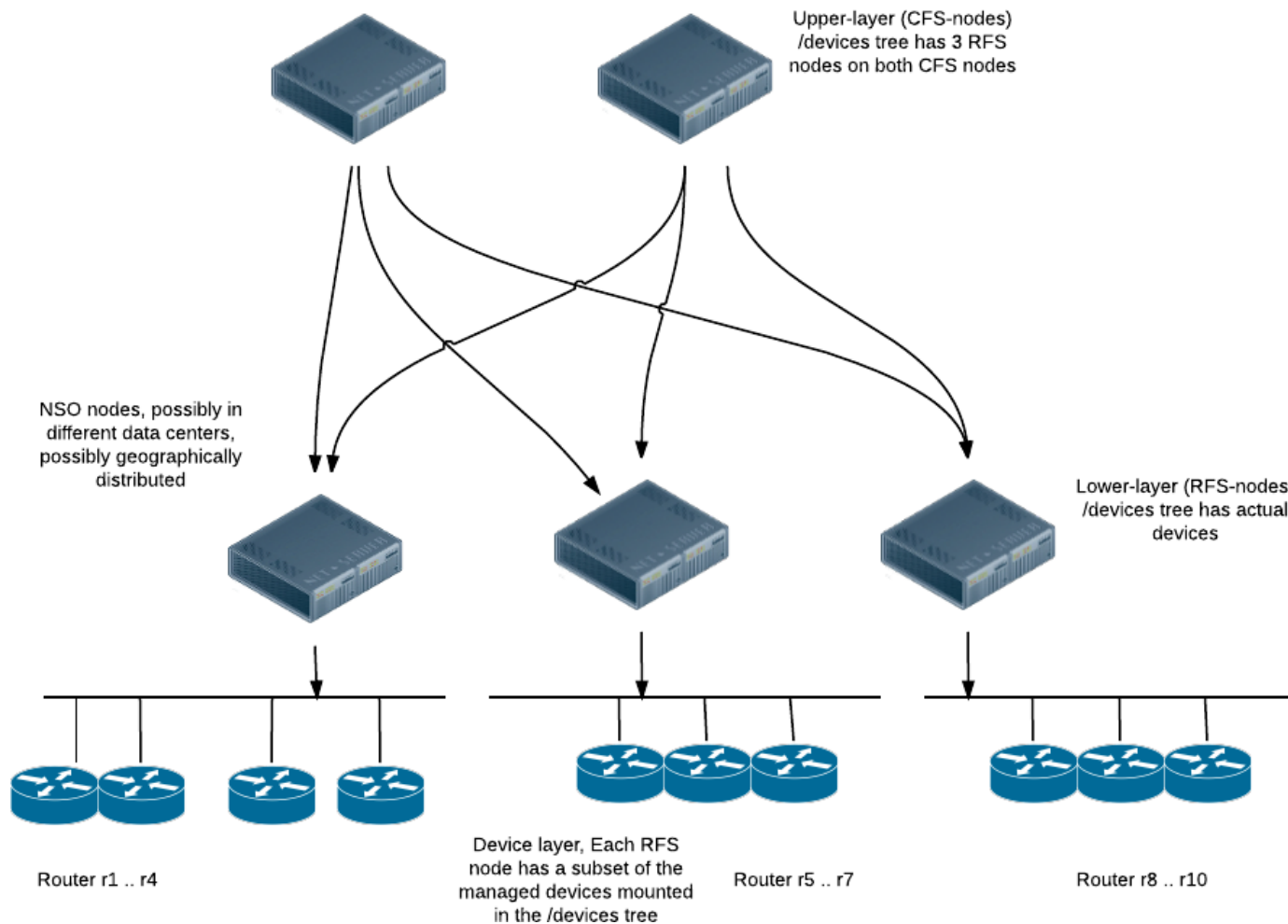
The example deployment consists of four hosts. A service node pair and a device node pair. We'll be using the NSO clustering technique from the service nodes to the device nodes. The two service nodes is an NSO HA-pair, and so are the two device nodes. The following picture shows our deployment.

Figure 4. The deployment network



This chapter equally well applies to a system installation with the "Layered Service Architecture, as well as a clustered installation. The picture above shows the IP addresses of a clustered installation, however, the nodes `srv-ncs` and `dev-ncs` might equally well have been LSA nodes as described in the "Layered Service Architecture book, or like the picture:

Figure 5. The deployment network



Thus the two NSO hosts `srv-ncs-m` and `srv-ncs-s` make up one HA pair, one active and one standby, and similarly for the two so called device nodes, `dev-ncs-m1` and `dev-ncs-s1`. The HA setup as well as the cluster setup will be thoroughly described later in this chapter.

The cluster part is really optional, it's only needed if you are expecting the amount of managed devices and/or the amount of instantiated services to be so large so that it doesn't fit in memory on a single NSO host. If for example the expected number of managed devices and services is less than 20k, it's recommended to *not* use clustering at all and instead equip the NSO hosts with sufficient RAM. Installation, performance, bug search, observation, maintenance, all become harder with clustering turned on.

HA on the other hand is usually not optional for customer deployment. Data resides in CDB, which is 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. This would mean that provisioning data survives a disk crash on the NSO host, but fail over would require manual intervention. As we shall see, the HA component we have *tailf-hcc* also requires some manual intervention, but only after an automatic fail over.

In this chapter we shall describe a complete HA/cluster setup though, you will have to decide for your deployment whether HA and/or clustering is required.

Initial NSO installation

We will perform an NSO system installation on 4 NSO hosts. NSO comes with a tool called `nct` which is ideal for the task at hand here. `nct` has its own documentation and will not be described here. `nct` is shipped together with NSO. The following prerequisites are needed for the `nct` operations.

- We need a user on the management station which has `sudo` rights on the four NSO hosts. Most of the `nct` operations that you'll execute towards the 4 NSO hosts require root privileges.
- Access to the NSO .bin install package as well as access to the NEDs and all packages you're planning to run. The packages shall be on the form of `tar.gz` packages.

We'll be needing an `NCT_HOSTSFILE`. In this example it looks like:

```
$ echo $NCT_HOSTFILE
/home/klacke/nct-hosts

$ cat $NCT_HOSTFILE

{"10.147.40.80", [{"name", "dev-ncs-s1"},
                 {"rest_pass", "MYPASS"},
                 {"ssh_pass", "MYPASS"},
                 {"netconf_pass", "MYPASS"},
                 {"rest_port", 8888}]}.

{"10.147.40.78", [{"name", "dev-ncs-m"},
                 {"rest_pass", "MYPASS"},
                 {"ssh_pass", "MYPASS"},
                 {"netconf_pass", "MYPASS"},
                 {"rest_port", 8888}]}.

{"10.147.40.190", [{"name", "srv-ncs-m"},
                  {"rest_pass", "MYPASS"},
                  {"ssh_pass", "MYPASS"},
                  {"netconf_pass", "MYPASS"},
                  {"rest_port", 8888}]}.

{"10.147.40.77", [{"name", "srv-ncs-s"},
                  {"rest_pass", "MYPASS"},
                  {"ssh_pass", "MYPASS"},
                  {"netconf_pass", "MYPASS"},
                  {"rest_port", 8888}]}.

$ ls -lat /home/klacke/nct-hosts
-rw----- 1 klacke staff 1015 Jan 22 13:12 /home/klacke/nct-hosts
```

The different passwords in `nct-hosts` file are all my regular Linux password on the target host.

We can use SSH keys, especially for normal SSH shell login, however, unfortunately, the `nct` tool doesn't work well with `ssh-agent`, thus the keys shouldn't have a pass phrase, if they do, we'll have to enter the pass phrase over and over again while using `nct`. Since `ssh-agent` doesn't work, and we'll be needing the password for the REST api access anyway, the recommended setup is to store the password for the target hosts in a read-only file. This is for ad-hoc `nct` usage.

This data is needed on the management station. That can be one of the 4 NSO hosts, but it can also be another host, e.g. an operator laptop. One convenient way to get easy access to the `nct` command is to do a "local install" of NSO on the management station. To test `nct`, and the SSH key setup you can do:

```
$ nct ssh-cmd -c 'sudo id'
```

```

SSH command to 10.147.40.80:22 [dev-ncs-s1]
SSH OK : 'ssh sudo id' returned: uid=0(root) gid=0(root) groups=0(root)

SSH command to 10.147.40.78:22 [dev-ncs-m1]
SSH OK : 'ssh sudo id' returned: uid=0(root) gid=0(root) groups=0(root)

SSH command to 10.147.40.190:22 [srv-ncs-m]
SSH OK : 'ssh sudo id' returned: uid=0(root) gid=0(root) groups=0(root)

SSH command to 10.147.40.77:22 [srv-ncs-s]
SSH OK : 'ssh sudo id' returned: uid=0(root) gid=0(root) groups=0(root)

```

Now you are ready to execute the NSO installer on all the 4 NSO hosts. This is done through the `nct` command `install`.

```

$ nct install --file ./nso-4.1.linux.x86_64.installer.bin --progress true
.....
Install NCS to 10.147.40.80:22
Installation started, see : /tmp/nso-4.1.linux.x86_64.installer.bin.log

Install NCS to 10.147.40.78:22
Installation started, see : /tmp/nso-4.1.linux.x86_64.installer.bin.log

Install NCS to 10.147.40.190:22
Installation started, see : /tmp/nso-4.1.linux.x86_64.installer.bin.log

Install NCS to 10.147.40.77:22
Installation started, see : /tmp/nso-4.1.linux.x86_64.installer.bin.log

```

If you for some reason want to undo everything and start over from scratch, the following command cleans up everything on all the NSO hosts.

```

$ nct ssh-cmd \
-c 'sudo /opt/ncs/current/bin/ncs-uninstall --non-interactive --all'

```

At this point NSO is properly installed on the NSO hosts. The default options were used for the NSO installer, thus files end up in the normal places on the NSO hosts. We have:

- Boot files in `/etc/init.d`, NSO configuration files in `/etc/ncs` and shell files under `/etc/profile.d`
- NSO run dir, with CDB database, packages directory, NSO state directory in `/var/opt/ncs`
- Log files in `/var/log/ncs`
- The releases structure in `/opt/ncs` with man pages for all NSO related commands under `/opt/ncs/current/man`

To read more about this, see man page `ncs-installer(1)`

Initial NSO configuration - ncs.conf

The first thing you must do is to decide on a few settings in `/etc/ncs/ncs.conf`. For starters, the crypto keys in `ncs.conf` (`/ncs-config/encrypted-strings`) must be the same in all 4 `ncs.conf` files. Furthermore, you must enable the services you want enabled. There are quite a few security considerations to consider here. The recommended AAA settings is to have NSO authenticate users via PAM although it is possible to have users in NSO CDB database.

For this example I choose to copy one of the generated `/etc/ncs/ncs.conf` files to the management station and edit it. See NSO man page `ncs.conf(1)` for all the settings of `ncs.conf`

- Enable the NSO ssh CLI login. `/ncs-config/cli/ssh/enabled`

- Modify the CLI prompt so that the hostname is part of the CLI prompt. `/ncs-config/cli/prompt`

```
<prompt1>\u@\H> </prompt1>
<prompt2>\u@\H% </prompt2>

<c-prompt1>\u@\H# </c-prompt1>
<c-prompt2>\u@\H(\m)# </c-prompt2>
```

- Enable the NSO https interface `/ncs-config/webui/`

The SSL certificates that get distributed with 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 real production environment, and the Web/REST interface is used for something which is not solely internal purposes it's a good idea to replace the self signed certificate with a properly signed certificate.

- Disable `/ncs-config/webui/cgi` unless needed.
- Enable the NSO netconf SSH interface `/ncs-config/netconf-northbound/`
- Enable the NSO ha in `ncs.conf`.

```
<ha>
  <enabled>true</enabled>
</ha>
```

- Centralized syslog settings. In real deployments, we typically want to enable remote syslog. The setting `/ncs-config/logs/syslog-config/udp` should be configured to send all logs to a centralized syslog server.
- PAM - the recommended authentication setting for NSO is to rely on Linux PAM. Thus all remote access to NSO must be done using real 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.
- Depending on the type of provisioning applications you have, you might want to turn `/ncs-config/rollback/enabled` off. Rollbacks don't work that well with reactive-fastmap applications. If your application is a classical NSO provisioning application, the recommendation is to enable rollbacks, otherwise not.

Now that you have a proper `ncs.conf` - the same config files can be used on all the 4 NSO hosts, we can copy the modified file to all hosts. To do this we use the `nct` command:

```
$ nct copy --file ncs.conf
$ nct ssh-cmd -c 'sudo mv /tmp/ncs.conf /etc/ncs'
$ nct ssh-cmd -c 'sudo chmod 600 /etc/ncs/ncs.conf'
```

Or use the builtin support for the `ncs.conf` file:

```
$ nct load-config --file ncs.conf --type ncs-conf
```

Note that the `ncs.conf` from this point on is highly sensitive. The file contains the encryption keys for all CDB data that is encrypted on disk. This usually contains passwords etc for various entities, such as login credentials to managed devices. In YANG parlance, this is all YANG data modeled with the types `tailf:des3-cbc-encrypted-string` or `tailf:aes-cfb-128-encrypted-string`

Setting up AAA

As we saw in the previous section, the REST HTTPS api is enabled. This API is used by a few of the crucial `nct` commands, thus if we want to use `nct`, we must enable password based REST login (through PAM)

The default AAA initialization file that gets shipped with NSO resides under `/var/opt/ncs/cdb/aaa_init.xml`. If we're 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 defaults are restrictive and fine though, so we'll keep them here.

Looking at the `aaa_init.xml` file we see that two groups are referred to in the NACM rule list, `ncsadmin` and `ncsoper`. The NSO authorization system is group based, thus for the rules to apply for a specific user, the user must be member of the right group. Authentication is performed by PAM, and authorization is performed by the NSO NACM rules. Adding myself to `ncsadmin` group will ensure that I get properly authorized.

```
$ nct ssh-cmd -c 'sudo addgroup ncsadmin'
$ nct ssh-cmd -c 'sudo adduser $USER ncsadmin'
```

Henceforth I will log into the different NSO hosts using my own login credentials. There are many advantages to this scheme, the main one being that all audit logs on the NSO hosts will show who did what and when. The common scheme of having a shared `admin` user with a shared password is not recommended.

To test the NSO logins, we must first start NSO:

```
$ nct ssh-cmd -c 'sudo /etc/init.d/ncs start'
```

Or use the `nct` command `nct start`:

```
$ nct start
```

At this point we should be able to curl login over REST, and also directly log in remotely to the NSO cli. On the admin host:

```
$ ssh -p 2024 srv-ncs-m
klacke connected from 10.147.40.94 using ssh on srv-ncs-m
klacke@srv-ncs-m> exit
Connection to srv-ncs-m closed.
```

Checking the NSO audit log on the NSO host `srv-ncs-m` we see at the end of `/var/log/ncs/audit.log`

```
<INFO> 5-Jan-2016::15:51:10.425 srv-ncs-m ncs[666]: audit user: klacke/0
logged in over ssh from 10.147.40.94 with authmeth:publickey
<INFO> 5-Jan-2016::15:51:10.442 srv-ncs-m ncs[666]: audit user:
klacke/21 assigned to groups: ncsadmin,sambashare,lpadmin,
klacke,plugdev,dip,sudo,cdrom,adm
<INFO> 5-Jan-2016::16:03:42.723 srv-ncs-m ncs[666]: audit user:
klacke/21 CLI 'exit'
<INFO> 5-Jan-2016::16:03:42.927 srv-ncs-m ncs[666]: audit user:
klacke/0 Logged out ssh <publickey> user
```

Especially the group assignment is worth mentioning here, we were assigned to the recently created `ncsadmin` group. Testing the REST api we get:

```
$ curl -u klacke:PASSW http://srv-ncs-m:8080/api -X GET
curl: (7) Failed to connect to srv-ncs-m port 8080: Connection refused
$ curl -k -u klacke:PASSW https://srv-ncs-m:8888/api -X GET
<api xmlns="http://tail-f.com/ns/rest" xmlns:y="http://tail-f.com/ns/rest">
  <version>0.5</version>
  <config/>
  <running/>
  <operational/>
  <operations/>
  <rollbacks/>
</api>
```

The `nct check` command is a good command to check all 4 NSO hosts in one go:

```
nct check --rest-pass PASSW --rest-port 8888 -c all
```

Cisco Smart Licensing

NSO uses Cisco Smart Licensing, described in detail in [Chapter 3, Cisco Smart Licensing](#). After you 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](#), you need to enter a token from your Cisco Smart Software Manager account on each host. You can use the same token for all instances. We can use the `nct cli-cmd` tool to do this on all NSO hosts:

```
$ nct cli-cmd --style cisco -c 'license smart register idtoken YzY2Yj...'
```



Note

The Cisco Smart Licensing CLI command is present only in the Cisco Style CLI, so make sure you use the `--style cisco` flag with `nct cli-cmd`

Global settings and timeouts

Depending on your installation, the size and speed of the managed devices, as well as the characteristics of your service applications - some of the 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 that goes from NSO to the managed devices. The default value is 20 seconds for all three. Some routers are slow to commit, some are sometimes slow to deliver it's full configuration. Adjust timeouts under `/devices/global-settings` accordingly.
- Service code timeouts. Some service applications can sometimes be slow. In order to minimize the chance of a service application timing out - adjusting `/java-vm/service-transaction-timeout` might be applicable - depending on the application.

There are quite a few different global settings to NSO, the two mentioned above usually needs to be changed. On the management station:

```
$ cat globs.xml
<config xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <global-settings>
      <connect-timeout>120</connect-timeout>
```



```

        <read-timeout>120</read-timeout>
        <write-timeout>120</write-timeout>
        <trace-dir>/var/log/ncs</trace-dir>
    </global-settings>
</devices>
<java-vm xmlns="http://tail-f.com/ns/ncs">
    <service-transaction-timeout>180</service-transaction-timeout>
</java-vm>
</config>
$ nct load-config --file globs.xml --type xml

```

Enabling SNMP

For real deployments we usually want to enable SNMP. Two reasons:

- When NSO alarms are created - SNMP traps automatically get created and sent - thus we typically want to enable SNMP and also set one or more trap targets
- Many organizations have SNMP based monitoring systems, in order to enable SNMP based system to monitor NSO we need SNMP enabled.

There is already a decent SNMP configuration in place, it just needs a few extra localizations: We need to enable SNMP, and decide:

- If and where-to send SNMP traps
- Which SNMP security model to choose.

At a minimum we could have:

```

klacke@srv-ncs-s% show snmp
agent {
    enabled;
    ip                0.0.0.0;
    udp-port          161;
    version {
        v1;
        v2c;
        v3;
    }
    engine-id {
        enterprise-number 32473;
        from-text         testing;
    }
    max-message-size 50000;
}
system {
    contact Klacke;
    name    nso;
    location Stockholm;
}
target test {
    ip        3.4.5.6;
    udp-port  162;
    tag       [ x ];
    timeout   1500;
    retries   3;
    v2c {
        sec-name test;
    }
}

```

```
community test {
    sec-name test;
}
notify test {
    tag x;
    type trap;
}
```

Loading the required NSO packages

We'll be using a couple of packages to illustrate the process of managing packages over a set of NSO nodes. The first prerequisite here is that all nodes *must* have the same version of all packages. If not, havoc will wreak. In particular HA will break, since a check is run during slave to master connect, ensuring that both nodes have exactly the same NSO packages loaded.

On our management station we have the following NSO packages.

```
$ ls -lt packages
total 15416
-rw-r--r-- 1 klacke klacke      8255 Jan  5 13:10 ncs-4.1-nso-util-1.0.tar.gz
-rw-r--r-- 1 klacke klacke 14399526 Jan  5 13:09 ncs-4.1-cisco-ios-4.0.2.tar.gz
-rw-r--r-- 1 klacke klacke 1369969 Jan  5 13:07 ncs-4.1-tailf-hcc-4.0.1.tar.gz
```

Package management in an NSO system install is a three-stage process.

- First, all versions of all packages, all reside in `/opt/ncs/packages` Since this is the initial install, we'll only have a single version of our 3 example packages.
- The version of each package we *want* to use will reside as a symlink in `/var/opt/ncs/packages/`
- And finally, the package which is actually running, will reside under `/var/opt/ncs/state/packages-in-use.cur`

The tool here is `nct packages`, it can be used to upload and install our packages in stages. The `nct packages` command work over the REST api, thus in the following examples I have added `{rest_user, "klacke"}` and also `{rest_port, 8888}` to my `$NCT_HOSTSFILE`. We upload all our packages as:

```
$ for p in packages/*; do
    nct packages --file $p -c fetch --rest-pass PASSW
done
Fetch Package at 10.147.40.80:8888
OK
.....
```

Verifying on one of the NSO hosts:

```
$ ls /opt/ncs/packages/
ncs-4.1-cisco-ios-4.0.2.tar.gz  ncs-4.1-tailf-hcc-4.0.1.tar.gz
ncs-4.1-nso-util-1.0.tar.gz
```

Verifying with the `nct` command

```
$ nct packages --rest-pass PASSW list
Package Info at 10.147.40.80:8888
  ncs-4.1-cisco-ios-4.0.2 (installable)
  ncs-4.1-nso-util-1.0 (installable)
  ncs-4.1-tailf-hcc-4.0.1 (installable)
Package Info at 10.147.40.78:8888
  ncs-4.1-cisco-ios-4.0.2 (installable)
```

```

ncs-4.1-nso-util-1.0 (installable)
ncs-4.1-tailf-hcc-4.0.1 (installable)
Package Info at 10.147.40.190:8888
ncs-4.1-cisco-ios-4.0.2 (installable)
ncs-4.1-nso-util-1.0 (installable)
ncs-4.1-tailf-hcc-4.0.1 (installable)
Package Info at 10.147.40.77:8888
ncs-4.1-cisco-ios-4.0.2 (installable)
ncs-4.1-nso-util-1.0 (installable)
ncs-4.1-tailf-hcc-4.0.1 (installable)

```

Next step is to install the packages. As stated above, package management in NSO is a three-stage process, we have now covered step one. The packages reside on the NSO hosts. Step two is to *install* the 3 packages. This is also done through the `nct` command as:

```

$ nct packages --package ncs-4.1-cisco-ios-4.0.2 --rest-pass PASSW -c install
$ nct packages --package ncs-4.1-nso-util-1.0 --rest-pass PASSW -c install
$ nct packages --package ncs-4.1-tailf-hcc-4.0.1 --rest-pass PASSW -c install

```

This command will setup the symbolic links from `/var/opt/ncs/packages` to `/opt/ncs/packages`. NSO is still running with the previous set of packages. Actually, even a restart of NSO will run with the previous set of packages. The packages that get loaded at startup time reside under `/var/opt/ncs/state/packages-in-use.cur`.

To force a single node to restart using the set of *installed* packages under `/var/opt/ncs/packages` we can do:

```
/etc/init.d/ncs restart-with-package-reload
```

This is a full NCS restart, depending on the amount of data in CDB and also depending on which data models are actually updated, it's usually faster to have the NSO node reload the data models and do the schema upgrade while running. The NSO CLI has support for this using the CLI command.

```

$ ncs_cli

klacke connected from 10.147.40.113 using ssh on srv-ncs-m
klacke@srv-ncs-m> request packages reload

```

Here however, we wish to do the data model upgrade on all 4 NSO hosts, the `nct` tool can do this as:

```

$ nct packages --rest-pass PASSW -c reload
Reload Packages at 10.147.40.80:8888
  cisco-ios      true
  nso-util       true
  tailf-hcc      true
Reload Packages at 10.147.40.78:8888
  cisco-ios      true
  nso-util       true
  tailf-hcc      true
Reload Packages at 10.147.40.190:8888
  cisco-ios      true
  nso-util       true
  tailf-hcc      true
Reload Packages at 10.147.40.77:8888
  cisco-ios      true
  nso-util       true
  tailf-hcc      true

```

To verify that all packages are indeed loaded and also running we can do the following in the CLI:

```
$ ncs_cli
```

```

klacke@srv-ncs-m> show status packages package oper-status
package cisco-ios {
    oper-status {
        up;
    }
}
package nso-util {
    oper-status {
        up;
    }
}
package tailf-hcc {
    oper-status {
        up;
    }
}

```

We can use the `nct` tool to do it on all NSO hosts

```
$ nct cli-cmd -c 'show status packages package oper-status'
```

This section covered initial loading of NSO packages, in a later section we will also cover upgrade of existing packages.

Preparing the HA of the NSO installation

In this example we will be running with two HA-pairs, the two service nodes will make up one HA-pair and the two device nodes will make up another HA-pair. We will use the `tailf-hcc` package as a HA framework. The package itself is well documented thus that will not be described here. Instead we'll just show a simple standard configuration of `tailf-hcc` and we'll focus on issues when managing and upgrading an HA cluster.

One simple alternative to the `tailf-hcc` package is to use completely manual HA, i.e HA entirely without automatic failover. An example of code that accomplish this can be found in the NSO example collection under `examples.ncs/web-server-farm/ha/packages/manual-ha`

I have also modified the `$NCT_HOSTSFILE` to have a few groups so that we can do `nct` commands to groups of NSO hosts.

If we plan to use VIP fail over, a prerequisite is the `arping` command and the `ip` command

```

$ nct ssh-cmd -c 'sudo aptitude -y install arping'
$ nct ssh-cmd -c 'sudo aptitude -y install iproute2'

```

The `tailf-hcc` package gives us two things and only that:

- All CDB data becomes replicated from the master to the slave
- If the master fails, the slave takes over and starts to act as master. I.e the package automatically handles one fail over. At fail over, the `tailf-hcc` either brings up a Virtual alias IP address using gratuitous ARP or by means of Quagga/BGP announce a better route to an anycast IP address.

Thus we become resilient to NSO host failures. However it's important to realize that the `tailf-hcc` is fairly primitive once a fail over has occurred. We shall run through a couple of failure scenarios in this section.

Following the `tailf-hcc` documentation we have the same HA configuration on both `srv-ncs-m` and `srv-ncs-s`. The tool to use in order to push identical config to two nodes, is `nct load-config`. We prepare the configuration as XML data on the management station:

```
$ dep cat srv-ha.xml
<ha xmlns="http://tail-f.com/pkg/tailf-hcc">
  <token>xyz</token>
  <interval>4</interval>
  <failure-limit>10</failure-limit>
  <member>
    <name>srv-ncs-m</name>
    <address>10.147.40.190</address>
    <default-ha-role>master</default-ha-role>
  </member>
  <member>
    <name>srv-ncs-s</name>
    <address>10.147.40.77</address>
    <default-ha-role>slave</default-ha-role>
    <failover-master>true</failover-master>
  </member>
</ha>
$ nct load-config --file srv-ha.xml --type xml --group srv

Node 10.147.40.190 [srv-ncs-m]
load-config result : successfully loaded srv-ha.xml with ncs_load

Node 10.147.40.77 [srv-ncs-s]
load-config result : successfully loaded srv-ha.xml with ncs_load
```

The last piece of the puzzle here is now to activate HA. The configuration is now there on both the service nodes. We use the `nct ha` command to basically just execute the CLI command `request ha commands activate` on the two service nodes.

```
$ nct ha --group srv --action activate --rest-pass PASSW
```

To verify the HA status we do:

```
$ nct ha --group srv --action status --rest-pass PASSW

HA Node 10.147.40.190:8888 [srv-ncs-m]
srv-ncs-m[master] connected srv-ncs-s[slave]

HA Node 10.147.40.77:8888 [srv-ncs-s]
srv-ncs-s[slave] connected srv-ncs-m[master]
```

To verify the whole setup, we can now also run the `nct check` command, now that HA is operational

```
$ nct check --group srv --rest-pass PASSW --netconf-user klacke all

ALL Check to 10.147.40.190:22 [srv-ncs-m]
SSH OK : 'ssh uname' returned: Linux
SSH+SUDO OK
DISK-USAGE FileSys=/dev/sda1 (/var,/opt) Use=37%
REST OK
NETCONF OK
NCS-VSN : 4.1
HA : mode=master, node-id=srv-ncs-m, connected-slave=srv-ncs-s

ALL Check to 10.147.40.77:22 [srv-ncs-s]
SSH OK : 'ssh uname' returned: Linux
SSH+SUDO OK
DISK-USAGE FileSys=/dev/sda1 (/var,/opt) Use=37%
REST OK
NETCONF OK
NCS-VSN : 4.1
HA : mode=slave, node-id=srv-ncs-s, master-node-id=srv-ncs-m
```

Handling tailf-hcc HA fallout

As previously indicated, the *tailf-hcc* is not especially sophisticated. Here follows a list of error scenarios after which the operator must act.

NSO slave host failure

If the *srv-ncs-s* host reboots, NSO will start from the */etc* boot scripts. The HA component cannot automatically decide what to do though. It will await an explicit operator command. After reboot, we will see:

```
klacke@srv-ncs-s> show status ncs-state ha
mode none;
[ok][2016-01-07 18:36:41]
klacke@srv-ncs-s> show status ha
member srv-ncs-m {
    current-ha-role unknown;
}
member srv-ncs-s {
    current-ha-role unknown;
}
```

On the designated slave node, and the preferred master node *srv-ncs-m* will show:

```
klacke@srv-ncs-m> show status ha
member srv-ncs-m {
    current-ha-role master;
}
member srv-ncs-s {
    current-ha-role unknown;
}
```

To remedy this, the operator must once again *activate* HA. It suffices to do it on *srv-ncs-s*, but we can in this case safely do it on both nodes, even doing it using the `nct ha` command. Re-activating HA on *srv-ncs-s* will ensure that

- All data from *srv-ncs-m* is copied to *srv-ncs-s*.
- Ensure that all future configuration changes (that have to go through *srv-ncs-m* are replicated.

NSO master host failure

This is the interesting fail over scenario. Powering off the *srv-ncs-m* master host, we see the following on *srv-ncs-s*:

- An alarm gets created on the designated slave

```
alarm-list {
    number-of-alarms 1;
    last-changed      2016-01-11T12:48:45.143+00:00;
    alarm ncs node-failure /ha/member[name='srv-ncs-m'] "" {
        is-cleared      false;
        last-status-change      2016-01-11T12:48:45.143+00:00;
        last-perceived-severity critical;
        last-alarm-text      "HA connection lost. 'srv-ncs-s' transitioning to HA MASTER.
                               When the problem has been fixed, role-override the old MASTER
                               to prevent config loss, then role-revert all nodes.
                               This will clear the alarm.";
```

- Fail over occurred:

```
klacke@srv-ncs-s> show status ha
member srv-ncs-m {
    current-ha-role unknown;
}
member srv-ncs-s {
    current-ha-role master;
}
```

This is a critical moment, HA has failed over. When the original master *srv-ncs-m* restarts, the operator **MUST** manually decide what to do. Restarting *srv-ncs-m* we get:

```
klacke@srv-ncs-m> show status ha
member srv-ncs-m {
    current-ha-role unknown;
}
member srv-ncs-s {
    current-ha-role unknown;
}
```

If we now activate the original master, it will resume its former master role. Since *srv-ncs-s* already is master, this will be a mistake. Instead we must

```
klacke@srv-ncs-m> request ha commands role-override role slave
status override
[ok][2016-01-11 14:28:27]
klacke@srv-ncs-m> request ha commands activate
status activated
[ok][2016-01-11 14:28:42]
klacke@srv-ncs-m> show status ha
member srv-ncs-m {
    current-ha-role slave;
}
member srv-ncs-s {
    current-ha-role master;
}
```

This means that all config from *srv-ncs-s* will be copied back to *srv-ncs-m*. Once HA is once again established, we can easily go back to original situation by executing:

```
klacke@srv-ncs-m> request ha commands role-revert
```

on both nodes. This is recommended in order to have the running situation as normal as possible.

Note: This is indeed a critical operation. It's actually possible to lose all or some data here. For example, assume that the original master *srv-ncs-m* was down for a period of time, the following sequence of events/commands will lose data.

- 1 *srv-ncs-m* goes down at time *t0*
- 2 Node *srv-ncs-s* continues to process provisioning request until time *t1* when it goes down.
- 3 Node *srv-ncs-s* goes down
- 4 The original master *srv-ncs-m* comes up and the operator activates *srv-ncs-m* manually. At which time it can start to process provisioning requests.

The above sequence of events/commands loses all provisioning requests between *t0* and *t1*

Setting up the VIP or L3 anycast BGP support

The final part of configuring HA is enabling either IP layer 2 VIP (Virtual IP) support or IP layer 3 BGP anycast fail over. Here we will describe layer 2 VIP configuration, details about anycast setup can be found in the *tailf-hcc* documentation.

We modify the HA configuration so that it looks as:

```
klacke@srv-ncs-m% show ha
token      xyz;
interval   4;
failure-limit 10;
vip {
    address 10.147.41.253;
}
member srv-ncs-m {
    address      10.147.40.190;
    default-ha-role master;
    vip-interface eth0;
}
member srv-ncs-s {
    address      10.147.40.77;
    default-ha-role slave;
    failover-master true;
    vip-interface eth0;
}
```

Whenever a node is master, it will also bring up a VIP.

```
$ ifconfig eth0:ncsvip
eth0:ncsvip Link encap:Ethernet HWaddr 08:00:27:0c:c3:48
inet addr:10.147.41.253 Bcast:10.147.41.255 Mask:255.255.254.0
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
```

The purpose of the VIP is to have northbound systems, northbound provisioning systems, activate NSO services through the VIP which will always be reachable as long as one NSO system is still up.

Referring to previous discussion above on manual activation, if the operator reactivates HA on the designated master after a master reboot, we will end up with two masters, both activating the VIP using gratuitous ARP. This must be avoided at all costs - thus HA activation must be done with care. It cannot be automated.

Preparing the clustering of the NSO installation

NSO clustering is a technique whereby we can use multiple NSO nodes for the data. Note: if all our data fits on one node, i.e. all service data and all device data, we recommend not using clustering. Clustering is non-trivial to configure, harder to search for errors and also has some performance issues. The NSO clustering has nothing to do with HA, it is solely a means of using multiple machines when our dataset is too big to fit in RAM one machine.

Communication between service node and device nodes is over SSH and the NETCONF interface. Thus the first thing we must do is to decide which user/password to use when service node establish the SSH connection to the device node. One good solution is to create a specific user solely for this purpose.

```
$ sudo adduser netconfncs --ingroup ncsadmin --disabled-login
$ sudo passwd netconfncs
...
$ id netconfncs
uid=1001(netconfncs) gid=1001(ncsadmin) groups=1001(ncsadmin)
```

At service node level we need some cluster configuration

```
klacke@srv-ncs-m% show cluster
authgroup devnodes {
    default-map {
```



```

        remote-name      netconfncs;
        remote-password $4$T8hh78koPrja9Hggowrl2A==;
    }
}

```

Our complete cluster configuration looks as:

```

klacke@srv-ncs-m> show configuration cluster
global-settings {
    caching {
        full-device-config {
            max-devices 100;
        }
    }
}
remote-node dev1 {
    address 10.147.41.254;
    port 2022;
    ssh {
        host-key-verification none;
    }
    authgroup devnodes;
    username netconfncs;
}
authgroup devnodes {
    default-map {
        remote-name      netconfncs;
        remote-password $4$T8hh78koPrja9Hggowrl2A==;
    }
}

```

Three important observations on this configuration:

- The address of the remote node is the VIP of the two device nodes.
- We turned off ssh host verification. If we need this, we must also make sure that the SSH keys of the two device nodes *dev-ncs-m1* and *dev-ncs-s1* are identical. Otherwise we'll not be able to connect over SSH after a fail over.
- We have caching turned on - we almost always want this. Especially fastmap based services almost require caching to be turned on. Unless caching is used, fastmap based services get very poor performance.

Testing the cluster configuration

Testing the cluster configuration involves testing a whole chain of credentials. To do this we must add at least one managed device. Managed devices must be added to both the device nodes and then also to the service nodes, but in a slightly different manner.

Either we hook up to a real managed device, or we can use a netsim device running on our management station. On the management station we can do:

```

$ ncs-netsim create-network packages/cisco-ios 1 c
DEVICE c0 CREATED
$ ncs-netsim start
DEVICE c0 OK STARTED
$ ncs-netsim list
ncs-netsim list for /home/klacke/dep/netsim

name=c0 netconf=12022 snmp=11022 ipc=5010 cli=10022
dir=/home/klacke/dep/netsim/c/c0

```

To hook up this device to our cluster we need to add this device to `/devices/device` on the device node(s) and on the service node(s). On the device node acting as HA master, *dev-ncs-m1*, we add:

```
klacke@dev-ncs-m1> show configuration devices
authgroups {
  group default {
    default-map {
      remote-name      admin;
      remote-password  $4$QM145chGWN5h3ggowr12A==;
    }
  }
}
device c0 {
  address  10.147.40.94;
  port     10022;
  authgroup default;
  device-type {
    cli {
      ned-id cisco-ios;
    }
  }
  state {
    admin-state unlocked;
  }
}
```

The netsim device *c0* has username *admin* with password *admin*. On the service node *srv-ncs-m*, we need to add the device as a *pointer* to the virtual node we named *dev1* in the cluster configuration at the service node, the *remote-node* which has the VIP of the device node HA-pair as address. On the service node we have:

```
klacke@srv-ncs-m% show devices
device c0 {
  remote-node dev1;
}
```

At this point we can read the data on device *c0* all the way from the top level service node *srv-ncs-m*. The HA slave service node is read-only, and all data is available from there too.

When trouble shooting the cluster setup, it's a good idea to have NED trace turned on at the device nodes, it's also a good idea to have cluster tracing turned on at the service node. On the service node:

```
klacke@srv-ncs-m% set cluster remote-node dev1 trace pretty
```

and on the device node:

```
klacke@dev-ncs-m1% set devices global-settings trace raw
```

NSO system and packages upgrade

NSO major upgrade

Upgrading NSO in the cluster is done as a staged process. To do this we need to have good group information in the `$NCT_HOSTSFILE` so that we can easily run `nct` commands to the designated masters and the designated slaves. See man-page for `nct`. There are three different forms of upgrade, the simpler is when we only upgrade some packages but retain the same NSO release, or when we do an NSO minor upgrade. The harder is when we do a major upgrade of NSO, because that also entails upgrading all packages. Maybe the packages are the same version as before, but the packages **MUST** be compiled with the right NSO version. The stages of an NSO upgrade are:

- 1 Take a full backup on all 4 nodes.
- 2 Disconnect the HA pairs by deactivating HA on the designated slaves. This means that the VIP will still be up on the designated masters.
- 3 Set the designated masters into read-only mode. VIP is up, but we cannot allow any updates while the upgrade is in progress.
- 4 Upgrade the designated slaves.
- 5 Deactivate HA on the designated masters, thereby removing the VIPs.
- 6 Activate HA on the designated slaves, and do *role-override*, ensuring that the VIP comes back up as soon as possible. At this point the system is up and online again, however without HA.
- 7 Upgrade the designated masters
- 8 Force the master to become slaves to the original designated slaves, connecting HA again. Once HA is operational, do HA role-revert on all nodes.

This is an error prone process, and scripting the process is highly recommended. One major caveat here is that all packages we have **MUST** match the NSO release. If they don't the `nct upgrade` command fails. Thus, we must not just `nct install` the right NSO release but we must also make sure that we by means of `nct packages fetch` all the packages we're currently running, but compiled for the new (or old) NSO release.

A script to upgrade our example environment here can look like:

```
#!/bin/bash

set -ex

vsn=$1
restpass=$2
img=nso-${vsn}.linux.x86_64.installer.bin

args="--rest-pass ${restpass}"

function fetch() {
    grp=$1
    for p in packages-${vsn}/*.gz; do
        nct packages --group ${grp} --file $p -c fetch ${args}
    done
}

nct backup
nct ha --group master --action readonly ${args} --mode enable
nct ha --group slave --action deactivate ${args}
nct install --file ${img} --group slave
fetch slave
nct upgrade --ncs-vsn ${vsn} --backup false -c upgrade --group slave ${args}

nct ha --group master --action deactivate ${args}
nct ha --group slave --action role-override ${args} --role master
nct ha --group slave --action activate ${args}

nct install --file ${img} --group master
fetch master
nct upgrade --ncs-vsn ${vsn} --backup false -c upgrade --group master ${args}

nct ha --group master --action readonly ${args} --mode disable ${args}
nct ha --group master --action role-override ${args} --role slave
nct ha --group master --action activate ${args}
```

Once we have run the script, it's paramount that we manually check the outcome. The `nct check` is ideal.

```
$ nct check --rest-pass PASSW

ALL Check to 10.147.40.80:22 [dev-ncs-s1]
SSH OK : 'ssh uname' returned: Linux
SSH+SUDO OK
DISK-USAGE FileSys=/dev/sda1 (/var,/opt) Use=58%
REST OK
NETCONF OK
NCS-VSN : 4.0.3
HA : mode=master, node-id=dev-ncs-s1, connected-slave=dev-ncs-m1

ALL Check to 10.147.40.78:22 [dev-ncs-m1]
SSH OK : 'ssh uname' returned: Linux
SSH+SUDO OK
DISK-USAGE FileSys=/dev/sda1 (/var,/opt) Use=55%
REST OK
NETCONF OK
NCS-VSN : 4.0.3
HA : mode=slave, node-id=dev-ncs-m1, master-node-id=dev-ncs-s1

ALL Check to 10.147.40.190:22 [srv-ncs-m]
SSH OK : 'ssh uname' returned: Linux
SSH+SUDO OK
DISK-USAGE FileSys=/dev/sda1 (/var,/opt) Use=54%
REST OK
NETCONF OK
NCS-VSN : 4.0.3
HA : mode=slave, node-id=srv-ncs-m, master-node-id=srv-ncs-s

ALL Check to 10.147.40.77:22 [srv-ncs-s]
SSH OK : 'ssh uname' returned: Linux
SSH+SUDO OK
DISK-USAGE FileSys=/dev/sda1 (/var,/opt) Use=57%
REST OK
NETCONF OK
NCS-VSN : 4.0.3
HA : mode=master, node-id=srv-ncs-s, connected-slave=srv-ncs-m
```

The above script leaves the system with reversed HA roles. If this is not desired, we must also do

```
$ nct ha --rest-pass PASSW --action role-revert
```

It's very important that we don't do this without ensuring that the designated master truly have a complete latest copy of all the configuration data.

NSO minor upgrade

This is considerably easier than the major upgrade. A minor NSO upgrade is when the third number in the NSO release is changed, for example upgrading from NSO 4.1.1 to NSO 4.1.2. A major upgrade is when the first or the second number is changed, for example upgrading to 4.1.2 to 4.2 is a major upgrade.

With minor upgrades we can keep all the packages as they are. Packages (actually .fxs files) are guaranteed to be compatible within a major release. Thus the upgrade scenario for simple upgrades look similar to the major upgrade, with the exception of packages - that can be retained as is.

A script perform an NSO minor upgrade of our example environment here can look like:

```
#!/bin/bash
```

```

set -ex

vsn=$1
restpass=$2
img=nso-${vsn}.linux.x86_64.installer.bin

args="--rest-pass ${restpass}"

nct backup
nct ha --group master --action readonly ${args} --mode enable
nct ha --group slave --action deactivate ${args}
nct install --file ${img} --group slave
nct ssh-cmd --group slave \
    -c 'sudo /etc/init.d/ncs restart-with-package-reload'

nct ha --group master --action deactivate ${args}
nct ha --group slave --action role-override ${args} --role master
nct ha --group slave --action activate ${args}

nct install --file ${img} --group master
nct ssh-cmd --group master \
    -c 'sudo /etc/init.d/ncs restart-with-package-reload'

nct ha --group master --action readonly ${args} --mode disable ${args}
nct ha --group master --action role-override ${args} --role slave
nct ha --group master --action activate ${args}

```

Package upgrade

Similarly to upgrading NSO, upgrading packages is also a staged process. We have the exact same sequence of steps as in upgrading NSO. This must also be scripted.

Here is a small script to do this.

```

#!/bin/bash

set -ex

package=$1
fromver=$2
tover=$3
ncsver=$4
restpass=$5

file=ncs-${ncsver}-${package}-${tover}.tar.gz
frompack=ncs-${ncsver}-${package}-${fromver}
topack=ncs-${ncsver}-${package}-${tover}

args="--rest-pass $restpass "

function pkg() {
    grp=$1
    nct packages --group ${grp} --file ${file} -c fetch ${args}
    nct packages --group ${grp} --package ${frompack} -c deinstall ${args}
    nct packages --group ${grp} --package ${topack} -c install ${args}
    nct cli-cmd -c 'request packages reload' --group ${grp}
}

nct backup

```

```

nct ha --group master --action readonly ${args} --mode enable
nct ha --group slave --action deactivate ${args}
pkg slave

nct ha --group master --action deactivate ${args}
nct ha --group slave --action role-override ${args} --role master
nct ha --group slave --action activate ${args}
nct ha --group master --action readonly ${args} --mode disable
pkg master
nct ha --group master --action role-override ${args} --role slave
nct ha --group master --action activate ${args}

```

This script also leaves the system with reversed HA roles, same argument as in previous section apply.

The script can be expanded to handle multiple packages in one go. It's more efficient to upgrade several packages in one go than in several steps. Another important efficiency note here is the use of `nct cli-cmd -c 'request packages reload'`. There are two ways to load new data models into NSO. This is one, the other is to invoke:

```
$ sudo /etc/init.d/ncs restart-with-package-reload
```

The former is considerably more efficient than the latter. If the amount of data in CDB is huge, the time difference to upgrade can be considerable.

An important note to make on NED upgrades is that care must be exercised when upgrading NEDs in production environments. If there exists NSO service instances that have provisioned services towards the old NED, the services may become out of sync after a NED upgrade - followed by a *sync-from*. Worst case scenario is that once services have been provisioned, some NED cannot ever be updated without explicit cleanup code attached to the installation.

It all depends on the type of changes in the NED, if there are major structural changes in YANG model data that is used by service provisioning code, we cannot simply upgrade NEDs without also doing the accompanying changes to the service packages.

Patch management

NSO has the ability to install - during runtime - emergency patches. These get delivered on the form of .beam files. The `nct patch` command can be used to install such patches.

Manual upgrade

This section is included so that the reader can understand what is done under the hood by the above examples. It's important to understand this in order to be able to repair a system which for some reason is broken.

Here we have a system running NSO 4.0.3, and we wish to manually upgrade to 4.1. System is running with the 3 packages we have been using above:

```

klacke@srv-ncs-m> show status packages package package-version
package cisco-ios {
    package-version 4.0.3;
}
package nso-util {
    package-version 1.0;
}
package tailf-hcc {
    package-version 4.0.1;
}
[ok][2016-01-20 13:05:52]

```

```
klacke@srv-ncs-m> show status ncs-state version
version 4.0.3;
```

To manually upgrade this system we must:

- 1 Do a system install of the new NSO release. This must be done as root. Download the `nso-4.1.linux.x86_64.installer.bin` to the host and install it as:

```
# sh nso-4.1.linux.x86_64.installer.bin --system-install
```

After this, the 4.1 NSO code resides under `/opt/ncs/4.1`, however the symbolic link `/opt/ncs/current` must also be updated. As root:

```
# rm /opt/ncs/current
# ln -s /opt/ncs/ncs-4.1 /opt/ncs/current
```

- 2 Update all the packages. For all packages we're running, we must first ensure that we have the exact same version of each package, but compiled for the new NSO release. All packages, for all versions of NSO that we have used, reside under `/opt/ncs/packages`

```
# ls /opt/ncs/packages
ncs-4.0.3-cisco-ios-4.0.3.tar.gz  ncs-4.1-cisco-ios-4.0.2.tar.gz
ncs-4.0.3-nso-util-1.0.tar.gz    ncs-4.1-nso-util-1.0.tar.gz
ncs-4.0.3-tailf-hcc-4.0.1.tar.gz ncs-4.1-tailf-hcc-4.0.1.tar.gz
```

Thus just download the right packages and put them all in `/opt/ncs/packages`. Next step is to manually rearrange all the symlinks in `/var/opt/ncs/packages` Initially we have links:

```
ncs-4.0.3-cisco-ios-4.0.3.tar.gz -> /opt/ncs/packages/ncs-4.0.3-cisco-ios-4.0.3.tar.gz
ncs-4.0.3-nso-util-1.0.tar.gz -> /opt/ncs/packages/ncs-4.0.3-nso-util-1.0.tar.gz
ncs-4.0.3-tailf-hcc-4.0.1.tar.gz -> /opt/ncs/packages/ncs-4.0.3-tailf-hcc-4.0.1.tar.gz
```

Instead we want:

```
# cd /var/opt/ncs/packages/
# rm -f *
# for p in /opt/ncs/packages/*4.1*; do ln -s $p; done
```

resulting in links:

```
ncs-4.1-cisco-ios-4.0.2.tar.gz -> /opt/ncs/packages/ncs-4.1-cisco-ios-4.0.2.tar.gz
ncs-4.1-nso-util-1.0.tar.gz -> /opt/ncs/packages/ncs-4.1-nso-util-1.0.tar.gz
ncs-4.1-tailf-hcc-4.0.1.tar.gz -> /opt/ncs/packages/ncs-4.1-tailf-hcc-4.0.1.tar.gz
```

- 3 Final step is to restart NSO, and tell it to reload packages. Again as root:

```
# /etc/init.d/ncs restart-with-package-reload
Stopping ncs: Starting ncs: .
```

and then also check the result:

```
$ ncs_cli
klacke@srv-ncs-m> show status packages package oper-status
package cisco-ios {
  oper-status {
    up;
  }
}
package nso-util {
  oper-status {
    up;
  }
}
package tailf-hcc {
```

```

        oper-status {
            up;
        }
    }
}
[ok][2016-01-20 13:33:59]
klacke@srv-ncs-m> show status ncs-state version
version 4.1;

```

System is upgraded, and all packages loaded fine.

Log management

We already covered some of the logging settings that are possible to 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 we have performed on our 4 hosts also install 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

NED logs

A crucial tool for debugging NSO installations are NED logs. These logs are very verbose and are for debugging only. Do not have these logs enabled in production. The NED trace logs are controlled through in 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 3 different levels of trace, and for various historic reasons we usually want different settings depending on device type.

- For all CLI NEDs we want to use the *raw* setting.
- For all ConfD based NETCONF devices we want to use the *pretty* setting. ConfD sends the NETCONF XML unformatted, pretty means that we get the XML formatted.
- For Juniper devices, we want to use the *raw* setting, Juniper sends sometimes broken XML that cannot be properly formatted, however their XML payload is already indented and formatted.
- For generic NED devices - depending on the level of trace support in the NED itself, we want either *pretty* or *raw*.
- For SNMP based devices, we want the *pretty* setting.

Thus it's usually not good enough to just control the NED trace from `/devices/global-settings/trace`

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 on a per Java package basis. For example if, we want to increase the level of logging on e.g the *tailf-hcc* code, we need to look into the code and find out the name of the corresponding Java package. Unpacking the *tailf-hcc* tar.gz package, we see in file `tailf-hcc/src/java/src/com/tailf/ns/tailfHcc/TcmApp.java` that package is called *com.tailf.ns.tailfHcc*. We can then do:


```
klacke@srv-ncs-s% show java-vm java-logging
logger com.tailf.ns.tailfHcc {
    level level-all;
}
```

Internal NSO log

The internal NSO log resides at `/var/log/ncs/ncserr.*`. The log is written in a binary format, to view the log run command:

```
$ ncs --printlog /var/log/ncs/ncserr.log
```

to view the internal error log.

The command `nct get-logs` grabs all logs from all hosts. This is good when collecting data from the system.

Monitoring the installation

All large scale deployments employ monitoring systems. There are plenty good tools to choose. Open source and commercial. Examples are *Cacti* and *Nagios*. All good monitoring tools has the ability to script (using various protocols) what should be monitored. Using the NSO REST api is ideal for this. We also recommend setting up a special read-only Linux user without shell access for this. The command `nct check` summarizes well what should be monitored.

Alarms

The REST api can be used to view the NSO alarm table. NSO alarms are not events, whenever an NSO alarm is created - an SNMP trap is also sent (assuming we have configured a proper SNMP target) All alarms require operator invention. Thus a monitoring tool should also *GET* the NSO alarm table.

```
curl -k -u klacke:PASSW https://srv-ncs-m:8888/api/operational/alarms/alarm-list -X GET
```

Whenever there are new alarms, an operator **MUST** take a look.

Security considerations

The AAA setup that we have described so far in this deployment document are the recommended AAA setup. To reiterate:

- Have all users that need access to NSO in PAM, this may then be through `/etc/passwd` or whatever. Do not store any users in CDB.
- Given the default *NACM* authorization rules we should have three different types of users on the system
 - Users with shell access that are members of `ncsadmin` Linux group. These users are considered fully trusted. They have full access to the system as well as the entire network.
 - Users without shell access that are members of `ncsadmin` Linux group. These users have full access to the network. They can SSH to the NSO SSH shell, they can execute arbitrary REST calls etc. They cannot manipulate backups and perform system upgrades. If we have provisioning systems north of NSO, it's recommended to assign a user of this type for those operations.
 - Users without shell access that are members of `ncsoper` Linux group. These users have read-only access to the network. They can SSH to the NSO SSH shell, they can execute arbitrary REST calls etc. They cannot manipulate backups and perform system upgrades.

If you have more fine grained authorization requirements than read-write all and read all, additional Linux groups can be created and the *NACM* rules can be updated accordingly. Since the *NACM* are data model specific, We'll do an example here. Assume we have a service that stores all data under `/mv:myvpn`. These services - once instantiated - manipulates the network. We want two new sets of users - apart from the *ncsoper* and *ncsadmin* users we already have. We want one set of users that can read everything under `/mv:myvpn` and one set of users that can read-write everything there. They're not allowed to see anything else in the system as a whole. To accomplish this we recommend:

- Create two new Linux groups. One called *vpnread* and one called *vpnwrite*.
- Modify `/nacm` by adding to all 4 nodes:

```
$ cat nacm.xml
<nacm xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-acm">
  <groups>
    <group>
      <name>vpnread</name>
    </group>
    <group>
      <name>vpnwrite</name>
    </group>
  </groups>
  <rule-list>
    <name>vpnwrite</name>
    <group>vpnwrite</group>
    <rule>
      <name>rw</name>
      <module-name>myvpn</module-name>
      <path>/myvpn</path>
      <access-operations>create read update delete</access-operations>
      <action>permit</action>
    </rule>
    <cmdrule xmlns="http://tail-f.com/yang/acm">
      <name>any-command</name>
      <action>permit</action>
    </cmdrule>
  </rule-list>
  <rule-list>
    <name>vpnread</name>
    <group>vpnread</group>
    <rule>
      <name>ro</name>
      <module-name>myvpn</module-name>
      <path>/myvpn</path>
      <access-operations>read</access-operations>
      <action>permit</action>
    </rule>
    <cmdrule xmlns="http://tail-f.com/yang/acm">
      <name>any-command</name>
      <action>permit</action>
    </cmdrule>
  </rule-list>
</nacm>

$ nct load-config --file nacm.xml --type xml
```

The above command will merge the data in `nacm.xml` on top of the already existing NACM data in CDB.

For a detailed discussion of the configuration of authorization rules via NACM, see [Chapter 9, The AAA infrastructure](#), in particular the section called “Authorization”.

A considerably more complex scenario is when you need/want to have users with shell access to the host, but those users are either untrusted, or shouldn't have any access to NSO at all. NSO listens to a port called `/ncs-config/ncs-ipc-address`, typically on localhost. By default this is `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 there are no AAA checks done on that port at all. If you have access to the port, you also have complete access to all of NSO. To drive this point home, when we invoke the command `ncs_cli`, that is a small C program that connects to the port and *tells* NSO who you are - assuming that authentication is already performed. There is even a documented flag `--noaaa` which tells NSO to skip all NACM rules checks for this session.

To cover the scenario with untrusted users with SHELL access, we must thus protect the port. This is done through the use of a file in the Linux file system. At install time, the file `/etc/ncs/ncs_ipc_access` gets created and populated with random data. Enable `/ncs-config/ncs-ipc-access-check/` enabled in `ncs.conf` and ensure that trusted users can read the `/etc/ncs/ncs_ipc_access` file for example by changing group access to the file.

```
$ cat /etc/ncs/ncs_ipc_access
cat: /etc/ncs/ncs_ipc_access: Permission denied
$ sudo chown root:ncsadmin /etc/ncs/ncs_ipc_access
$ sudo chmod g+r /etc/ncs/ncs_ipc_access
$ ls -lat /etc/ncs/ncs_ipc_access
$ cat /etc/ncs/ncs_ipc_access
.....
```

