Northbound APIs

Release: NSO 6.2.5

Published: May 17, 2010

Last Modified: May 14, 2024

Americas Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 527-0883

CONTENTS

CHAPTER 1

Introduction

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

This chapter describes the various northbound programmatic APIs in NSO NETCONF, REST, and SNMP. These APIs are used by external systems that need to communicate with NSO, such as portals, OSS, or BSS systems.

NSO has two northbound interfaces intended for human usage, the CLI and the WebUI. These interfaces are described in Chapter 5, *NSO CLI* in *User Guide* and Chapter 13, *Web User Interface* in *User Guide* respectively.

There are also programmatic Java, Python, and Erlang APIs intended to be used by applications integrated with NSO itself. See the section called “Running Application Code” in *Development Guide* for more information about these APIs.

Integrating an External System with NSO

There are two APIs to choose from when an external system should communicate with NSO: NETCONF and REST. Which one to choose is mostly a subjective matter. REST may at first sight appear to be simpler to use, but is not as feature-rich as NETCONF. By using a NETCONF client library such as the open source Java library JNC or Python library ncclient, the integration task is significantly reduced.

Both NETCONF and REST provide functions for manipulating the configuration (including creating services) and reading operational state from NSO. NETCONF provides more powerful filtering functions than REST.

NETCONF and SNMP can be used to receive alarms as notifications from NSO. NETCONF provides a reliable mechanism to receive notifications over SSH, whereas SNMP notifications are sent over UDP.

Regardless of the protocol you choose for integration, please keep in mind all of them communicate with the NSO server over network sockets, which may be unreliable. Additionally, write transactions in NSO can fail if they conflict with another, concurrent transaction. As a best practice, the client implementation should be able to gracefully handle such errors and be prepared to retry requests. For details on the NSO concurrency, please refer to Chapter 24, *NSO Concurrency Model* in *Development Guide*.

CHAPTER 2

The NSO NETCONF Server

Introduction

This chapter describes the north bound NETCONF implementation in NSO. As of this writing, the server supports the following specifications:

- RFC 4741 - NETCONF Configuration Protocol
- RFC 4742 - Using the NETCONF Configuration Protocol over Secure Shell (SSH)
- RFC 5277 - NETCONF Event Notifications
- RFC 5717 - Partial Lock Remote Procedure Call (RPC) for NETCONF

- RFC 6020 - YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)
- RFC 6021 - Common YANG Data Types
- RFC 6022 - YANG Module for NETCONF Monitoring
- RFC 6241 - Network Configuration Protocol (NETCONF)
- RFC 6242 - Using the NETCONF Configuration Protocol over Secure Shell (SSH)
- RFC 6243 - With-defaults capability for NETCONF
- RFC 6470 - NETCONF Base Notifications
- RFC 6536 - NETCONF Access Control Model
- RFC 6991 - Common YANG Data Types
- RFC 7895 - YANG Module Library
- RFC 7950 - The YANG 1.1 Data Modeling Language
- RFC 8071 - NETCONF Call Home and RESTCONF Call Home
- RFC 8342 - Network Management Datastore Architecture (NMDA)
- RFC 8525 - YANG Library
- RFC 8528 - YANG Schema Mount
- RFC 8526 - NETCONF Extensions to Support the Network Management Datastore Architecture
- RFC 8639 - Subscription to YANG Notifications
- RFC 8640 - Dynamic Subscription to YANG Events and Datastores over NETCONF
- RFC 8641 - Subscription to YANG Notifications for Datastore Updates

Note For the <delete-config> operation specified in RFC 4741 / RFC 6241, only <url> with scheme "file" is supported for the <target> parameter - i.e. no data stores can be deleted. The concept of deleting a data store is not well defined, and at odds with the transaction-based configuration management of NSO. To delete the entire *contents* of a data store, with full transactional support, a <copy-config> with an empty <config/> element for the <source> parameter can be used.

Note For the <partial-lock> operation, RFC 5717, section 2.4.1 says that if a node in the scope of the lock is deleted by the session owning the lock, it is removed from the scope of the lock. In NSO this is not true; the deleted node is kept in the scope of the lock.

NSO NETCONF north bound API can be used by arbitrary NETCONF clients. A simple Python based NETCONF client called `netconf-console` is shipped as source code in the distribution. See the section called “Using netconf-console” for details. Other NETCONF clients will work too, as long as they adhere to the NETCONF protocol. If you need a Java client, the open source client JNC can be used.

When integrating NSO into larger OSS/NMS environments, the NETCONF API is a good choice of integration point.

Protocol Capabilities

The NETCONF server in NSO supports the following capabilities in both NETCONF 1.0 (RFC 4741) and NETCONF 1.1 (RFC 6241).

`:writable-running`

This capability is always advertised.

`:candidate`

Not supported by NSO.

`:confirmed-commit`

Not supported by NSO.

`:rollback-on-error`

This capability allows the client to set the `<error-option>` parameter to `rollback-on-error`. The other permitted values are `stop-on-error` (default) and `continue-on-error`. Note that the meaning of the word "error" in this context is not defined in the specification. Instead, the meaning of this word must be defined by the data model. Also note that if `stop-on-error` or `continue-on-error` is triggered by the server, it means that some parts of the edit operation succeeded, and some parts didn't. The error `partial-operation` must be returned in this case. `partial-operation` is obsolete and SHOULD NOT be returned by a server. If some other error occurs (i.e. an error not covered by the meaning of "error" above), the server generates an appropriate error message, and the data store is unaffected by the operation.

The NSO server never allows partial configuration changes, since it might result in inconsistent configurations, and recovery from such a state can be very difficult for a client. This means that regardless of the value of the `<error-option>` parameter, NSO will always behave as if it had the value `rollback-on-error`. So in NSO, the meaning of the word "error" in `stop-on-error` and `continue-on-error`, is something which never can happen.

It is possible to configure the NETCONF server to generate an `operation-not-supported` error if the client asks for the `error-option` `continue-on-error`. See ncs.conf(5) in *Manual Pages* .

`:validate`

NSO supports both version 1.0 and 1.1 of this capability.

`:startup`

Not supported by NSO.

`:url`

The URL schemes supported are *`file`*, *`ftp`*, and *`sftp`* (SSH File Transfer Protocol).

There is no standard URL syntax for the *`sftp`* scheme, but NSO supports the syntax used by `curl`:

```
sftp://<user>:<password>@<host>/<path>
```

Note that user name and password must be given for *`sftp`* URLs.

NSO does not support `validate` from a url.

`:xpath`

The NETCONF server supports XPath according to the W3C XPath 1.0 specification (https://www.w3.org/TR/xpath).

The following list of optional standard capabilities are also supported:

`:notification`

NSO implements the `urn:ietf:params:netconf:capability:notification:1.0` capability, including support for the optional replay feature.

See the section called “Notification Capability” for details.

`:with-defaults`

NSO implements the `urn:ietf:params:netconf:capability:with-defaults:1.0` capability, which is used by the server to inform the client how default values are handled by the server, and by the client to control whether defaults values should be generated to replies or not.

If the capability is enabled, NSO also implements the `urn:ietf:params:netconf:capability:with-operational-defaults:1.0` capability, which targets the operational state datastore while the `:with-defaults` capability targets configuration datastores.

`:yang-library:1.0`

NSO implements the `urn:ietf:params:netconf:capability:yang-library:1.0` capability, which informs the client that server implements the YANG module library RFC 7895, and informs the client about the current `module-set-id`.

`:yang-library:1.1`

NSO implements the `urn:ietf:params:netconf:capability:yang-library:1.1` capability, which informs the client that server implements the YANG library RFC 8525, and informs the client about the current `content-id`.

Protocol YANG Modules

In addition to the protocol capabilities listed above, NSO also implements a set of YANG modules that are closely related to the protocol.

`ietf-netconf-nmda`

This module from RFC 8526 defines the NMDA extension to NETCONF. It defines the following features:

`origin`

Indicates that the server supports the origin annotation. It is not advertised by default.

The support for `origin` can be enabled in `ncs.conf` (see ncs.conf(5) in *Manual Pages*). If it is enabled, the `origin` feature is advertised.

`with-defaults`

Advertised if the server supports the `:with-defaults` capability, which NSO does.

`ietf-subscribed-notifications`

This module from RFC 8639 defines operations, configuration data nodes, and operational state data nodes related to notification subscriptions. It defines the following features:

`configured`

Indicates that the server supports configured subscriptions. This feature is not advertised.

`dscp`

Indicates that the server supports the ability to set the Differentiated Services Code Point (DSCP) value in outgoing packets. This feature is not advertised.

`encode-json`

Indicates that the server support JSON encoding of notifications. This is not applicable to NETCONF, and this feature is not advertised.

`encode-xml`

Indicates that the server support XML encoding of notifications. This feature is advertised by NSO.

`interface-designation`

Indicates that a configured subscription can be configured to send notifications over a specific interface. This feature is not advertised.

`qos`

Indicates that a publisher supports absolute dependencies of one subscription's traffic over another as well as weighted bandwidth sharing between subscriptions. This feature is not advertised.

`replay`

Indicates that historical event record replay is supported. This feature is advertised by NSO.

`subtree`

Indicates that the server supports subtree filtering of notifications. This feature is advertised by NSO.

`supports-vrf`

Indicates that a configured subscription can be configured to send notifications from a specific VRF. This feature is not advertised.

`xpath`

Indicates that the server supports XPath filtering of notifications. This feature is advertised by NSO.

In addition to this, NSO does not support pre-configuration or monitoring of subtree filters, and thus advertises a deviation module that deviates `/filters/stream-filter/filter-spec/stream-subtree-filter` and `/subscriptions/subscription/target/stream/stream-filter/within-subscription/filter-spec/stream-subtree-filter` as "not-supported".

There is basic support for monitoring subscriptions via the `/subscriptions` container. Currently it is possible to view dynamic subscriptions' attributes: `subscription-id`, `stream`, `encoding`, `receiver`, `stop-time`, and `stream-xpath-filter`. Unsupported attributes are: `stream-subtree-filter`, `receiver/sent-event-records`, `receiver/excluded-event-records`, and `receiver/state`.

`ietf-yang-push`

This module from RFC 8641 extends operations, data nodes, and operational state defined in ietf-subscribed-notifications; and also introduces continuous and customizable notification subscriptions for updates from running and operational datastores. It defines the same features as ietf-subscribed-notifications and also the following feature:

`on-change`

Indicates that on-change triggered notifications are supported. This feature is advertised by NSO but only supported on the running datastore.

In addition to this, NSO does not support pre-configuration or monitoring of subtree filters, and thus advertises a deviation module that deviates `/filters/selection-filter/filter-spec/datastore-subtree-filter` and `/subscriptions/subscription/target/datastore/selection-filter/within-subscription/filter-spec/datastore-subtree-filter` as "not-supported".

The monitoring of subscriptions via the `subscriptions` container does currently not support the attributes: `periodic/period`, `periodic/state`, `on-change/dampening-period`, `on-change/sync-on-start`, `on-change/excluded-change`.

Advertising Capabilities and YANG Modules

All enabled NETCONF capabilities are advertised in the hello message that the server sends to the client.

A YANG module is supported by the NETCONF server if its fxs file is found in NSO's loadPath, and if the fxs file is exported to NETCONF.

The following YANG modules are built-in, which means that their fxs files need not be present in the loadPath. If they are found in the loadPath they are skipped.

- `ietf-netconf`

- `ietf-netconf-with-defaults`
- `ietf-yang-library`
- `ietf-yang-types`
- `ietf-inet-types`
- `ietf-restconf`
- `ietf-datastores`
- `ietf-yang-patch`

All built-in modules are always supported by the server.

All YANG version 1 modules supported by the server are advertised in the hello message, according to the rules defined in RFC 6020.

All YANG version 1 and version 1.1 modules supported by the server are advertised in the YANG library.

If a YANG module (any version) is supported by the server, and its .yang or .yin file is found in the fxs file or in the loadPath, then the module is also advertised in the `schema` list defined in `ietf-netconf-monitoring`, made available for download with the RPC operation `get-schema`, and if RESTCONF is enabled, also advertised in the `schema` leaf in `ietf-yang-library`. See the section called “Monitoring of the NETCONF Server”.

Advertising Device YANG Modules

NSO uses YANG Schema Mount to mount the data models for the devices. There are two mount points, one for the configuration (in `/devices/device/config`), and one for operational state data (in `/devices/device/live-status`). As defined in YANG Schema Mount, a client can read the `module` list from the YANG library in each of these mount points to learn which YANG models each device supports via NSO.

For example, to get the YANG library data for device "x0", we can do:

```
$ netconf-console --get -x '/devices/device[name="x0"]/config/yang-library'
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
  <data>
    <devices xmlns="http://tail-f.com/ns/ncs">
      <device>
        <name>x0</name>
        <config>
          <yang-library xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library">
            <module-set>
              <name>common</name>
              <module>
                <name>a</name>
                <namespace>urn:a</namespace>
              </module>
              <module>
                <name>b</name>
                <namespace>urn:b</namespace>
              </module>
            </module-set>
            <schema>
              <name>common</name>
              <module-set>common</module-set>
            </schema>
            <datastore>
              <name xmlns:ds="urn:ietf:params:xml:ns:yang:ietf-datastores">\
                 ds:running\
              </name>
              <schema>common</schema>
            </datastore>
            <datastore>
              <name xmlns:ds="urn:ietf:params:xml:ns:yang:ietf-datastores">\
                ds:intended\
              </name>
              <schema>common</schema>
            </datastore>
            <datastore>
              <name xmlns:ds="urn:ietf:params:xml:ns:yang:ietf-datastores">\
                ds:operational\
              </name>
              <schema>common</schema>
            </datastore>
            <content-id>f0071b28c1e586f2e8609da036379a58</content-id>
          </yang-library>
        </config>
      </device>
    </devices>
  </data>
</rpc-reply>
```

The set of modules reported for a device is the set of modules that NSO knows, i.e., the set of modules compiled for the specific device type. This means that all devices of the same device type will report the same set of modules. Also note that the device may support other modules that are not known to NSO. Such modules are not reported here.

NETCONF Transport Protocols

The NETCONF server natively supports the mandatory SSH transport, i.e., SSH is supported without the need for an external SSH daemon (such as sshd). It also supports integration with OpenSSH.

Using OpenSSH

NSO is delivered with a program **netconf-subsys** which is an OpenSSH *subsystem* program. It is invoked by the OpenSSH daemon after successful authentication. It functions as a relay between the ssh daemon and NSO; it reads data from the ssh daemon from standard input, and writes the data to NSO over a loopback socket, and vice versa. This program is delivered as source code in `$NCS_DIR/src/ncs/netconf/netconf-subsys.c`. It can be modified to fit the needs of the application. For example, it could be modified to read the group names for a user from an external LDAP server.

When using OpenSSH, the users are authenticated by OpenSSH, i.e. the user names are not stored in NSO. To use OpenSSH, compile the **netconf-subsys** program, and put the executable in e.g. `/usr/local/bin`. Then add the following line to the ssh daemon's config file, `sshd_config`:

```
Subsystem     netconf   /usr/local/bin/netconf-subsys
```

The connection from **netconf-subsys** to NSO can be arranged in one of two different ways:

1 Make sure NSO is configured to listen to TCP traffic on localhost, port 2023, and disable SSH in `ncs.conf` (see ncs.conf(5) in *Manual Pages*). (Re)start sshd and NSO. Or:

2 Compile **netconf-subsys** to use a connection to the IPC port instead of the NETCONF TCP transport (see the `netconf-subsys.c` source for details), and disable both TCP and SSH in `ncs.conf`. (Re)start sshd and NSO.

  This method may be preferable, since it makes it possible to use the IPC Access Check (see the section called “Restricting access to the IPC port” in *Administration Guide*) to restrict the unauthenticated access to NSO that is needed by **netconf-subsys**.

By default the **netconf-subsys** program sends the names of the UNIX groups the authenticated user belongs to. To test this, make sure that NSO is configured to give access to the group(s) the user belongs to. Easiest for test is to give access to all groups.

Configuration of the NETCONF Server

NSO itself is configured through a configuration file called `ncs.conf`. For a description of the parameters in this file, please see the ncs.conf(5) in *Manual Pages* man page.

Error Handling

When NSO processes `<get>`, `<get-config>`, and `<copy-config>` requests, the resulting data set can be very large. To avoid buffering huge amounts of data, NSO streams the reply to the client as it traverses the data tree and calls data provider functions to retrieve the data.

If a data provider fails to return the data it is supposed to return, NSO can take one of two actions. Either it simply closes the NETCONF transport (default), or it can reply with an *inline rpc error* and continue to process the next data element. This behavior can be controlled with the `/ncs-config/netconf/rpc-errors` configuration parameter (see ncs.conf(5) in *Manual Pages*).

An inline error is always generated as a child element to the parent of the faulty element. For example, if an error occurs when retrieving the leaf element "mac-address" of an "interface" the error might be:

```
<interface>
  <name>atm1</name>
  <rpc-error xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <error-type>application</error-type>
    <error-tag>operation-failed</error-tag>
    <error-severity>error</error-severity>
    <error-message xml:lang="en">Failed to talk to hardware</error-message>
    <error-info>
      <bad-element>mac-address</bad-element>
    </error-info>
  </rpc-error>
  ...
</interface>
```

If a `get_next` call fails in the processing of a list, a reply might look like this:

```
<interface>
  <!-- successfully retrieved list entry -->
  <name>eth0</name>
  <mtu>1500</mtu>
  <!-- more leafs here -->
</interface>
<rpc-error xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <error-type>application</error-type>
  <error-tag>operation-failed</error-tag>
  <error-severity>error</error-severity>
  <error-message xml:lang="en">Failed to talk to hardware</error-message>
  <error-info>
    <bad-element>interface</bad-element>
  </error-info>
</rpc-error>
```

Using netconf-console

The `netconf-console` program is a simple NETCONF client. It is delivered as Python source code and can be used as-is or modified.

When NSO has been started, we can use `netconf-console` to query the configuration of the NETCONF Access Control groups:

```
$ netconf-console --get-config -x /nacm/groups
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
  <data>
    <nacm xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-acm">
      <groups>
        <group>
          <name>admin</name>
          <user-name>admin</user-name>
          <user-name>private</user-name>
        </group>
        <group>
          <name>oper</name>
          <user-name>oper</user-name>
          <user-name>public</user-name>
        </group>
      </groups>
    </nacm>
  </data>
</rpc-reply>
```

With the `-x` flag an XPath expression can be specified, in order to retrieve only data matching that expression. This is a very convenient way to extract portions of the configuration from the shell or from shell scripts.

Monitoring of the NETCONF Server

RFC 6022 - YANG Module for NETCONF Monitoring defines a YANG module, `ietf-netconf-monitoring`, for monitoring of the NETCONF server. It contains statistics objects such as number of RPCs received, status objects such as user sessions, and an operation to retrieve data models from the NETCONF server.

This data model defines an RPC operation, `get-schema`, which is used to retrieve YANG modules from the NETCONF server. NSO will report the YANG modules for all fxs files that are reported as capabilities, and for which the corresponding YANG or YIN file is stored in the fxs file or found in the loadPath. If a file is found in the loadPath, it has priority over a file stored in the fxs file. Note that by default, the module and its submodules are stored in the fxs file by the compiler.

If the YANG (or YIN files) are copied into the loadPath, they can be stored as is or compressed with gzip. The filename extension MUST be ".yang", ".yin", ".yang.gz", or ".yin.gz".

Also available is a Tail-f specific data model, `tailf-netconf-monitoring`, which augments `ietf-netconf-monitoring` with additional data about files available for usage with the `<copy-config>` command with a *file* `<url>` source or target. `/ncs-config/netconf-north-bound/capabilities/url/enabled` and `/ncs-config/netconf-north-bound/capabilities/url/file/enabled` must both be set to true. If rollbacks are enabled, those files are listed as well, and they can be loaded using `<copy-config>`.

This data model also adds data about which notification streams are present in the system, and data about sessions that subscribe to the streams.

Notification Capability

This section describes how NETCONF notifications are implemented within NSO, and how the applications generates these events.

Central to NETCONF notifications is the concept of a *stream*. The stream serves two purposes. It works like a high-level filtering mechanism for the client. For example, if the client subscribes to notifications on the `security` stream, it can expect to get security related notifications only. Second, each stream may have its own log mechanism. For example by keeping all debug notifications in a `debug` stream, they can be logged separately from the `security` stream.

Built-in Notification Streams

NSO has built-in support for the well-known stream `NETCONF`, defined in RFC 5277 and RFC 8639. NSO supports the notifications defined in RFC 6470 - NETCONF Base Notifications on this stream. If the application needs to send any additional notifications on this stream, it can do so.

NSO can be configured to listen to notifications from devices, and send those notifications to northbound NETCONF clients. The stream `device-notifications` is used for this purpose. In order to enable this, the stream `device-notifications` must be configured in `ncs.conf`, and additionally, subscriptions must be created in `/ncs:devices/device/notifications`.

Defining Notification Streams

It is up to the application to define which streams it supports. In NSO, this is done in `ncs.conf` (see ncs.conf(5) in *Manual Pages*). Each stream must be listed, and whether it supports replay or not. The following example enables the built-in stream `device-notifications` with replay support, and an additional, application-specific stream `debug` without replay support:

```
<notifications>
  <event-streams>
    <stream>
      <name>device-notifications</name>
      <description>Notifications received from devices</description>
      <replay-support>true</replay-support>
      <builtin-replay-store>
        <enabled>true</enabled>
        <dir>/var/log</dir>
        <max-size>S10M</max-size>
        <max-files>50</max-files>
      </builtin-replay-store>
    </stream>
    <stream>
      <name>debug</name>
      <description>Debug notifications</description>
      <replay-support>false</replay-support>
    </stream>
  </event-streams>
</notifications>
```

The well-known stream `NETCONF` does not have to be listed, but if it isn't listed, it will not support replay.

Automatic Replay

NSO has builtin support for logging of notifications, i.e., if replay support has been enabled for a stream, NSO automatically stores all notifications on disk ready to be replayed should a NETCONF client ask for logged notifications. In the `ncs.conf` fragment above the security stream has been setup to use the builtin notification log/replay store. The replay store uses a set of wrapping log files on disk (of a certain number and size) to store the security stream notifications.

The reason for using a wrap log is to improve replay performance whenever a NETCONF client asks for notifications in a certain time range. Any problems with log files not being properly closed due to hard power failures etc. is also kept to a minimum, i.e., automatically taken care of by NSO.

Subscribed Notifications

This section describes how Subscribed Notifications are implemented for NETCONF within NSO.

Subscribed Notifications is defined in RFC 8639 and the NETCONF transport binding is defined in RFC 8640. Subscribed Notifications build upon NETCONF notifications defined in RFC 5277 and have a number of key improvements:

- Multiple subscriptions on a single transport session
- Support for dynamic and configured subscriptions
- Modification of an existing subscription in progress
- Per-subscription operational counters
- Negotiation of subscription parameters (through the use of hints returned as part of declined subscription requests)
- Subscription state change notifications (e.g., publisher-driven suspension, parameter modification)
- Independence from transport

Compatibility with NETCONF notifications

Both NETCONF notifications and Subscribed Notifications can be used at the same time and are configured the same way in `ncs.conf`. However there are some differences and limitations.

For Subscribed Notifications, a new subscription is requested by invoking the RPC `establish-subscription`. For NETCONF notifications, the corresponding RPC is `create-subscription`.

A NETCONF session can only have either subscribers started with `create-subscription` or `establish-subscription` simultaneously.

- If a session has subscribers established with `establish-subscription` and receives a request to create subscriptions with `create-subscription`, an `<rpc-error>` is sent containing `<error-tag>` `operation-not-supported`.

  If a session has subscribers created with `create-subscription` and receives a request to establish subscriptions with `establish-subscription`, an `<rpc-error>` is sent containing `<error-tag>` `operation-not-supported`.

Dynamic subscriptions send all notifications on the transport session where they were established.

Monitoring subscriptions

Existing subscriptions and their configuration can be found in the `/subscriptions` container.

For example, viewing all established subscriptions, we can do:

```
$ netconf-console --get -x /subscriptions
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
  <data>
    <subscriptions xmlns="urn:ietf:params:xml:ns:yang:ietf-subscribed-notifications">
      subscription>
       <id>3</id>
       <stream-xpath-filter>/if:interfaces/interface[name='eth0']/enabled</stream-xpath-filte
       <stream>interface</stream>
       <stop-time>2030-10-04T14:00:00+02:00</stop-time>
       <encoding>encode-xml</encoding>
       <receivers>
         <receiver>
           <name>127.0.0.1:57432</name>
```

```
          <state>active</state>
        </receiver>
      </receivers>
     /subscription>
    </subsrcriptions>
  </data>
</rpc-reply>
```

Limitations

It is not possible to establish a subscription with a stored filter from `/filters`.

The support for monitoring subscriptions have basic functionality. It is possible to read `subscription-id`, `stream`, `stream-xpath-filter`, `replay-start-time`, `stop-time`, `encoding`, `receivers/receiver/name`, and `receivers/receiver/state`.

The leaf `stream-subtree-filter` is deviated as "not-supported", hence can not be read.

The unsupported leafs in the subscriptions container are the following: `stream-subtree-filter`, `receiver/sent-event-records`, and `receiver/excluded-event-records`.

YANG-Push

This section describes how YANG-Push is implemented for NETCONF within NSO.

YANG-Push is defined in RFC 8641 and the NETCONF transport binding is defined in RFC 8640. YANG-Push implementation in NSO introduces a subscription service that provides updates from a datastore. This implementation supports dynamic subscriptions on updates of datastore nodes. A subscribed receiver is provided with update notifications according to terms of the subscription. There are two types of notification messages defined to provide updates and these are used according to subscription terms.

- `push-update` notification is a complete, filtered update that reflects the data of the subscribed datastore. It is the type of notification that is used for `periodic` subscriptions. A `push-update` notification can also be used for the `on-change` subscriptions in case of a receiver asks for synchronization, either at the start of a new subscription or by sending a resync request for an established subscription.

  An example `push-update` notification:

```
<notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
  <eventTime>2020-06-10T10:00:00.00Z</eventTime>
  <push-update xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push">
    <id>1</id>
    <datastore-contents>
      <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
        <interface>
          <name>eth0</name>
          <oper-status>up</oper-status>
        </interface>
      </interfaces>
    </datastore-contents>
  </push-update>
</notification>
```

- `push-change-update` notification is the most common type of notification that is used for `on-change` subscriptions. It provides a set of filtered changes that happened on the subscribed datastore since the last update notification. The update records are constructed in a form of `YANG-Patch Media Type` that is defined in RFC 8072.

  An example `push-change-update` notification:

```
<notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
  <eventTime>2020-06-10T10:05:00.00Z</eventTime>
  <push-change-update
    xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push">
    <id>2</id>
    <datastore-changes>
      <yang-patch>
        <patch-id>s2-p4</patch-id>
        <edit>
          <edit-id>edit1</edit-id>
          <operation>merge</operation>
          <target>/ietf-interfaces:interfaces</target>
          <value>
            <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
              <interface>
                <name>eth0</name>
                <oper-status>down</oper-status>
              </interface>
            </interfaces>
          </value>
        </edit>
      </yang-patch>
    </datastore-changes>
  </push-change-update>
</notification>
```

Periodic Subscriptions

For periodic subscriptions, updates are triggered periodically according to specified time interval. Optionally a reference `anchor-time` can be provided for a specified `period`.

On-Change Subscriptions

For on-change subscriptions, updates are triggered whenever a change is detected on the subscribed information. In case of rapidly changing data, instead of receiving frequent notifications for every change, a receiver may specify a `dampening-period` to receive update notifications in a lower frequency. A receiver may request for synchronization at the start of a subscription by using `sync-on-start` option. And a receiver may filter out specific types of changes by providing a list of `excluded-change` parameters.

To provide updates for `on-change` subscriptions on `operational` datastore, data provider applications are required to implement push on-change callbacks. For more details, see the PUSH ON-CHANGE CALLBACKS in *Manual Pages* section of confd_lib_dp(3) in *Manual Pages* .

YANG-Push Operations

In addition to RPCs defined in subscribed notifications, YANG-Push defines `resync-subscription` RPC. Upon receipt of `resync-subscription`, if the subscription is an on-change triggered type, a `push-update` notification is sent to receiver according to the terms of subscription. Otherwise an appropriate error response is sent.

- `resync-subscription`

Monitoring of YANG-Push Subscriptions

YANG-Push subscriptions can be monitored in similar way to Subscribed Notifications through /subscriptions container. For more information see the section called “Monitoring subscriptions”.

YANG-Push filters differ from the filters of Subscribed Notifications and they are specified as `datastore-xpath-filter` and `datastore-subtree-filter`. The leaf `datastore-subtree-filter` is deviated as "not-supported", hence can not be monitored. And also YANG-Push specific update trigger parameters `periodic/period`, `periodic/anchor-time`, `on-change/dampening-period`, `on-change/sync-on-start` and `on-change/excluded-change` are not supported for monitoring.

Limitations

- `modify-subscriptions` operation does not support changing a subscriptions update trigger type from `periodic` to `on-change` or vice versa.
- `on-change` subscriptions do not work for changes that are made through the CDB-API.
- `on-change` subscriptions do not work on internal callpoints such as ncs-state, ncs-high-availability and live-status.

Actions Capability

Note This capability is deprecated, since actions are now supported in standard YANG 1.1. It is recommended to use standard YANG 1.1 for actions.

Overview

This capability introduces a new RPC operation that is used to invoke actions defined in the data model. When an action is invoked, the instance on which the action is invoked is explicitly identified by an hierarchy of configuration or state data.

Here is a simple example that invokes the action `sync-from` on the device `ce1`. It uses the **netconf-console** command:

```
$ cat ./sync-from-ce1.xml
<action xmlns="http://tail-f.com/ns/netconf/actions/1.0">
  <data>
    <devices xmlns="http://tail-f.com/ns/ncs">
      <device>
        <name>ce1</name>
        <sync-from/>
      </device>
    </devices>
  </data>
</action>
$ netconf-console --rpc sync-from-ce1.xml
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
  <data>
    <devices xmlns="http://tail-f.com/ns/ncs">
      <device>
        <name>ce1</name>
        <sync-from>
          <result>true</result>
        </sync-from>
      </device>
    </devices>
  </data>
</rpc-reply>
```

Capability Identifier

The actions capability is identified by the following capability string:

```
http://tail-f.com/ns/netconf/actions/1.0
```

Transactions Capability

Overview

This capability introduces four new RPC operations that are used to control a two-phase commit transaction on the NETCONF server. The normal <edit-config> operation is used to write data in the transaction, but the modifications are not applied until an explicit <commit-transaction> is sent.

This capability is formally defined in the YANG module "tailf-netconf-transactions". It is recommended that this module is enabled.

A typical sequence of operations looks like this:

```
C                           S
|                           |
|  capability exchange      |
|-------------------------->|
|<------------------------->|
|                           |
|   <start-transaction>     |
|-------------------------->|
|<--------------------------|
|         <ok/>             |
|                           |
|     <edit-config>         |
|-------------------------->|
|<--------------------------|
|         <ok/>             |
|                           |
|  <prepare-transaction>    |
|-------------------------->|
|<--------------------------|
|         <ok/>             |
|                           |
|   <commit-transaction>    |
|-------------------------->|
|<--------------------------|
|         <ok/>             |
|                           |
```

Dependencies

None.

Capability Identifier

The transactions capability is identified by the following capability string:

```
http://tail-f.com/ns/netconf/transactions/1.0
```

New Operation: <start-transaction>

Description

Starts a transaction towards a configuration datastore. There can be a single ongoing transaction per session at any time.

When a transaction has been started, the client can send any NETCONF operation, but any <edit-config> or <copy-config> operation sent from the client MUST specify the same <target> as the <start-transaction>, and any <get-config> MUST specify the same <source> as <start-transaction>.

If the server receives an <edit-config> or <copy-config> with another <target>, or a <get-config> with another <source>, an error MUST be returned with an <error-tag> set to "invalid-value".

The modifications sent in the <edit-config> operations are not immediately applied to the configuration datastore. Instead they are kept in the transaction state of the server. The transaction state is only applied when a <commit-transaction> is received.

The client sends a <prepare-transaction> when all modifications have been sent.

Parameters

target:	Name of the configuration datastore towards which the transaction is started.
with-inactive:	If this parameter is given, the transaction will handle the "inactive" and "active" attributes. If given, it MUST also be given in the <edit-config> and <get-config> invocations in the transaction.

Positive Response

If the device was able to satisfy the request, an <rpc-reply> is sent that contains an <ok> element.

Negative Response

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

If there is an ongoing transaction for this session already, an error MUST be returned with <error-app-tag> set to "bad-state".

Example

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <start-transaction xmlns="http://tail-f.com/ns/netconf/transactions/1.0">
    <target>
     <running/>
    </target>
  </start-transaction>
</rpc>

<rpc-reply message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

New Operation: <prepare-transaction>

Description

Prepares the transaction state for commit. The server may reject the prepare request for any reason, for example due to lack of resources or if the combined changes would result in an invalid configuration datastore.

After a successful <prepare-transaction>, the next transaction related rpc operation must be <commit-transaction> or <abort-transaction>. Note that an <edit-config> cannot be sent before the transaction is either committed or aborted.

Care must be taken by the server to make sure that if <prepare-transaction> succeeds then the <commit-transaction> SHOULD not fail, since this might result in an inconsistent distributed state. Thus, <prepare-transaction> should allocate any resources needed to make sure the <commit-transaction> will succeed.

Parameters

None.

Positive Response

If the device was able to satisfy the request, an <rpc-reply> is sent that contains an <ok> element.

Negative Response

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

If there is no ongoing transaction in this session, or if the ongoing transaction already has been prepared, an error MUST be returned with <error-app-tag> set to "bad-state".

Example

```
<rpc message-id="103"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <prepare-transaction
     xmlns="http://tail-f.com/ns/netconf/transactions/1.0"/>
</rpc>

<rpc-reply message-id="103"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

New Operation: <commit-transaction>

Description

Applies the changes made in the transaction to the configuration datastore. The transaction is closed after a <commit-transaction>.

Parameters

None.

Positive Response

If the device was able to satisfy the request, an <rpc-reply> is sent that contains an <ok> element.

Negative Response

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

If there is no ongoing transaction in this session, or if the ongoing transaction already has not been prepared, an error MUST be returned with <error-app-tag> set to "bad-state".

Example

```
<rpc message-id="104"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <commit-transaction
     xmlns="http://tail-f.com/ns/netconf/transactions/1.0"/>
</rpc>

<rpc-reply message-id="104"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

New Operation: <abort-transaction>

Description

Aborts the ongoing transaction, and all pending changes are discarded. <abort-transaction> can be given at any time during an ongoing transaction.

Parameters

None.

Positive Response

If the device was able to satisfy the request, an <rpc-reply> is sent that contains an <ok> element.

Negative Response

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

If there is no ongoing transaction in this session, an error MUST be returned with <error-app-tag> set to "bad-state".

Example

```
<rpc message-id="104"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <abort-transaction
     xmlns="http://tail-f.com/ns/netconf/transactions/1.0"/>
</rpc>

<rpc-reply message-id="104"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

Modifications to Existing Operations

The <edit-config> operation is modified so that if it is received during an ongoing transaction, the modifications are not immediately applied to the configuration target. Instead they are kept in the transaction state of the server. The transaction state is only applied when a <commit-transaction> is received.

Note that it doesn't matter if the <test-option> is 'set' or 'test-then-set' in the <edit-config>, since nothing is actually set when the <edit-config> is received.

Inactive Capability

Overview

This capability is used by the NETCONF server to indicate that it supports marking nodes as being inactive. A node that is marked as inactive exists in the data store, but is not used by the server. Any node can be marked as inactive.

In order to not confuse clients that do not understand this attribute, the client has to instruct the server to display and handle the inactive nodes. An inactive node is marked with an "inactive" XML attribute, and in order to make it active, the "active" XML attribute is used.

This capability is formally defined in the YANG module "tailf-netconf-inactive".

Dependencies

None.

Capability Identifier

The inactive capability is identified by the following capability string:

```
http://tail-f.com/ns/netconf/inactive/1.0
```

New Operations

None.

Modifications to Existing Operations

A new parameter, <with-inactive>, is added to the <get>, <get-config>, <edit-config>, <copy-config>, and <start-transaction> operations.

The <with-inactive> element is defined in the http://tail-f.com/ns/netconf/inactive/1.0 namespace, and takes no value.

If this parameter is present in <get>, <get-config>, or <copy-config>, the NETCONF server will mark inactive nodes with the "inactive" attribute.

If this parameter is present in <edit-config> or <copy-config>, the NETCONF server will treat inactive nodes as existing, so that an attempt to create a node which is inactive will fail, and an attempt to delete a node which is inactive will succeed. Further, the NETCONF server accepts the "inactive" and "active" attributes in the data hierarchy, in order to make nodes inactive or active, respectively.

If the parameter is present in <start-transaction>, it MUST also be present in any <edit-config>, <copy-config>, <get>, or <get-config> operations within the transaction. If it is not present in <start-transaction>, it MUST NOT be present in any <edit-config> operation within the transaction.

The "inactive" and "active" attributes are defined in the http://tail-f.com/ns/netconf/inactive/1.0 namespace. The "inactive" attribute's value is the string "inactive", and the "active" attribute's value is the string "active".

Example

This request creates an inactive interface:

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target>
      <running/>
    </target>
    <with-inactive
       xmlns="http://tail-f.com/ns/netconf/inactive/1.0"/>
    <config>
      <top xmlns="http://example.com/schema/1.2/config">
        <interface inactive="inactive">
          <name>Ethernet0/0</name>
          <mtu>1500</mtu>
        </interface>
      </top>
    </config>
  </edit-config>
</rpc>

<rpc-reply message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

This request shows the inactive interface:

```
<rpc message-id="102"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
      <running/>
    </source>
    <with-inactive
       xmlns="http://tail-f.com/ns/netconf/inactive/1.0"/>
  </get-config>
</rpc>

<rpc-reply message-id="102"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
    <top xmlns="http://example.com/schema/1.2/config">
      <interface inactive="inactive">
        <name>Ethernet0/0</name>
        <mtu>1500</mtu>
      </interface>
    </top>
  </data>
</rpc-reply>
```

This request shows that inactive data is not returned unless the client asks for it:

```
<rpc message-id="103"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
      <running/>
    </source>
  </get-config>
</rpc>
```

```
<rpc-reply message-id="103"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
  </data>
</rpc-reply>
```

This request activates the interface:

This request creates an inactive interface:

```
<rpc message-id="104"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target>
      <running/>
    </target>
    <with-inactive
       xmlns="http://tail-f.com/ns/netconf/inactive/1.0"/>
    <config>
      <top xmlns="http://example.com/schema/1.2/config">
        <interface active="active">
          <name>Ethernet0/0</name>
        </interface>
      </top>
    </config>
  </edit-config>
</rpc>

<rpc-reply message-id="104"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

Rollback Id Capability

Overview

This module extends existing operations with a with-rollback-id parameter which will, when set, extend the result with information about the rollback that was generated for the operation if any.

The rollback id returned is the id from within the rollback file which is stable with regards to new rollbacks being created.

Dependencies

None.

Capability Identifier

The transactions capability is identified by the following capability string:

```
http://tail-f.com/ns/netconf/with-rollback-id
```

Modifications to Existing Operations

This module adds a parameter 'with-rollback-id' to the following RPCs:

```
o  edit-config
```

```
   o  copy-config
   o  commit
   o  commit-transaction
```

If 'with-rollback-id' is given, rollbacks are enabled and the operation results in a rollback file being created the response will contain a rollback reference.

NETCONF extensions in NSO

The YANG module `tailf-netconf-ncs` augments some NETCONF operations with additional parameters to control the behavior in NSO over NETCONF. See that YANG module for all details. In this section the options are summarized.

To control the commit behaviour of NSO the following input parameters are available:

`no-revision-drop`	NSO will not run its data model revision algorithm, which requires all participating managed devices to have all parts of the data models for all data contained in this transaction. Thus, this flag forces NSO to never silently drop any data set operations towards a device.
`no-overwrite`	NSO will check that the data that should be modified has not changed on the device compared to NSO's view of the data.
`no-networking`	Do not send any data to the devices. This is a way to manipulate CDB in NSO without generating any southbound traffic.
`no-out-of-sync-check`	Continue with the transaction even if NSO detects that a device's configuration is out of sync.
`no-deploy`	Commit without invoking the service create method, i.e, write the service instance data without activating the service(s). The service(s) can later be re-deployed to write the changes of the service(s) to the network.
`reconcile/keep-non-service-config`	Reconcile the service data. All data which existed before the service was created will now be owned by the service. When the service is removed that data will also be removed. In technical terms the reference count will be decreased by one for everything which existed prior to the service. If manually configured data exists below in the configuration tree that data is kept.
`reconcile/discard-non-service-config`	Reconcile the service data but do not keep manually configured data that exists below in the configuration tree.
`use-lsa`	Force handling of the LSA nodes as such. This flag tells NSO to propagate applicable commit flags and actions to the LSA nodes without applying them on the upper NSO node itself. The commit flags affected are *`dry-run`*, *`no-networking`*, *`no-out-of-sync-check`*, *`no-overwrite`* and *`no-revision-drop`*.
`no-lsa`	Do not handle any of the LSA nodes as such. These nodes will be handled as any other device.
`commit-queue/async`	Commit the transaction data to the commit queue. The operation returns successfully if the transaction data has been successfully placed in the queue.
`commit-queue/sync/timeout`	Commit the transaction data to the commit queue. The operation does not return until the transaction data has been sent to all devices, or a timeout occurs. The timeout value specifies a maximum number of seconds to wait for the completion.

commit-queue/sync/infinity	Commit the transaction data to the commit queue. The operation does not return until the transaction data has been sent to all devices.
commit-queue/bypass	If `/devices/global-settings/commit-queue/enabled-by-default` is *true* the data in this transaction will bypass the commit queue. The data will be written directly to the devices.
commit-queue/atomic	Sets the atomic behaviour of the resulting queue item. Possible values are: *true* and *false*. If this is set to *false*, the devices contained in the resulting queue item can start executing if the same devices in other non-atomic queue items ahead of it in the queue are completed. If set to *true*, the atomic integrity of the queue item is preserved.
commit-queue/block-others	The resulting queue item will block subsequent queue items, which use any of the devices in this queue item, from being queued.
commit-queue/lock	Place a lock on the resulting queue item. The queue item will not be processed until it has been unlocked, see the actions **unlock** and **lock** in `/devices/commit-queue/queue-item`. No following queue items, using the same devices, will be allowed to execute as long as the lock is in place.
commit-queue/tag	The value is a user defined opaque tag. The tag is present in all notifications and events sent referencing the specific queue item.
commit-queue/error-option	The error option to use. Depending on the selected error option NSO will store the reverse of the original transaction to be able to undo the transaction changes and get back to the previous state. This data is stored in the `/devices/commit-queue/completed` tree from where it can be viewed and invoked with the **rollback** action. When invoked the data will be removed. Possible values are: *continue-on-error*, *rollback-on-error* and *stop-on-error*. The *continue-on-error* value means that the commit queue will continue on errors. No rollback data will be created. The *rollback-on-error* value means that the commit queue item will roll back on errors. The commit queue will place a lock with `block-others` on the devices and services in the failed queue item. The **rollback** action will then automatically be invoked when the queue item has finished its execution. The lock will be removed as part of the rollback. The *stop-on-error* means that the commit queue will place a lock with `block-others` on the devices and services in the failed queue item. The lock must then either manually be released when the error is fixed or the **rollback** action under `/devices/commit-queue/completed` be invoked. **Note** Read about error recovery in the section called “Commit Queue” in *User Guide* for a more detailed explanation.
trace-id	Use the provided trace id as part of the log messages emitted while processing. If no trace id is given, NSO is going to generate and assign a trace id to the processing.

These optional input parameters are augmented into the following NETCONF operations:

- `commit`

- `edit-config`
- `copy-config`
- `prepare-transaction`

The operation `prepare-transaction` is also augmented with an optional parameter *`dry-run`*, which can be used to show the effects that would have taken place, but not actually commit anything to the datastore or to the devices. *`dry-run`* takes an optional parameter *`outformat`*, which can be used to select in which format the result is returned. Possible formats are *xml* (default), *cli* and *native*. The optional *`reverse`* parameter can be used together with the *native* format to display the device commands for getting back to the current running state in the network if the commit is successfully executed. Beware that if any changes are done later on the same data the reverse device commands returned are invalid.

FASTMAP attributes such as backpointers and reference counters are typically internal to NSO and are not shown by default. The optional parameter *`with-service-meta-data`* can be used to include these in the NETCONF reply. The parameter is augmented into the following NETCONF operations:

- `get`
- `get-config`
- `get-data`

The Query API

The Query API consists of a number of RPC operations to start queries, fetch chunks of the result from a query, restart a query, and stop a query.

In the installed release there are two YANG files named `tailf-netconf-query.yang` and `tailf-common-query.yang` that defines these operations. An easy way to find the files is to run the following command from the top directory of release installation:

```
$ find . -name tailf-netconf-query.yang
```

The API consists of the following operations:

- `start-query`: Start a query and return a query handle.
- `fetch-query-result`: Use a query handle to repeatedly fetch chunks of the result.
- `immediate-query`: Start a query and return the entire result immediately.
- `reset-query`: (Re)set where the next fetched result will begin from.
- `stop-query`: Stop (and close) the query.

In the following examples, the following data model is used:

```
container x {
  list host {
    key number;
    leaf number {
      type int32;
    }
    leaf enabled {
      type boolean;
    }
    leaf name {
      type string;
    }
    leaf address {
      type inet:ip-address;
    }
  }
}
```

Here is an example of a `start-query` operation:

```
<start-query xmlns="http://tail-f.com/ns/netconf/query">
  <foreach>
    /x/host[enabled = 'true']
  </foreach>
  <select>
    <label>Host name</label>
    <expression>name</expression>
    <result-type>string</result-type>
  </select>
  <select>
    <expression>address</expression>
    <result-type>string</result-type>
  </select>
  <sort-by>name</sort-by>
  <limit>100</limit>
  <offset>1</offset>
</start-query>
```

An informal interpretation of this query is:

For each `/x/host` where `enabled` is true, select its `name`, and `address`, and return the result sorted by `name`, in chunks of 100 results at the time.

Let us discuss the various pieces of this request.

The actual XPath query to run is specified by the `foreach` element. In the example below will search for all `/x/host` nodes that has the `enabled` node set to `true`:

```
<foreach>
  /x/host[enabled = 'true']
</foreach>
```

Now we need to define what we want to have returned from the node set by using one or more `select` sections. What to actually return is defined by the XPath `expression`.

We must also choose how the result should be represented. Basically, it can be the actual value or the path leading to the value. This is specified per select chunk The possible result-types are: `string` , `path` , `leaf-value` and `inline`.

The difference between `string` and `leaf-value` is somewhat subtle. In the case of `string` the result will be processed by the XPath function `string()` (which if the result is a node-set will concatenate all the values). The `leaf-value` will return the value of the first node in the result. As long as the result is a leaf node, `string` and `leaf-value` will return the same result. In the example above, we are using `string` as shown below. At least one `result-type` must be specified.

The result-type `inline` makes it possible to return the full sub-tree of data in XML format. The data will be enclosed with a tag: `data`.

Finally we can specify an optional `label` for a convenient way of labeling the returned data. In the example we have the following:

```
<select>
  <label>Host name</label>
  <expression>name</expression>
  <result-type>string</result-type>
</select>
<select>
  <expression>address</expression>
  <result-type>string</result-type>
</select>
```

The returned result can be sorted. This is expressed as XPath expressions, which in most cases are very simple and refers to the found node set. In this example we sort the result by the content of the `name` node:

```
<sort-by>name</sort-by>
```

To limit the max amount of results in each chunk that `fetch-query-result` will return we can set the `limit` element. The default is to get all results in one chunk.

```
<limit>100</limit>
```

With the `offset` element we can specify at which node we should start to receive the result. The default is 1, i.e., the first node in the resulting node-set.

```
<offset>1</offset>
```

Now, if we continue by putting the operation above in a file `query.xml` we can send a request, using the command **netconf-console**, like this:

```
$ netconf-console --rpc query.xml
```

The result would look something like this:

```
<start-query-result>
  <query-handle>12345</query-handle>
</start-query-result>
```

The query handle (in this example "12345") must be used in all subsequent calls. To retrieve the result, we can now send:

```
<fetch-query-result xmlns="http://tail-f.com/ns/netconf/query">
  <query-handle>12345</query-handle>
</fetch-query-result>
```

Which will result in something like the following:

```
<query-result xmlns="http://tail-f.com/ns/netconf/query">
  <result>
    <select>
      <label>Host name</label>
      <value>One</value>
    </select>
    <select>
      <value>10.0.0.1</value>
    </select>
  </result>
  <result>
    <select>
      <label>Host name</label>
      <value>Three</value>
    </select>
    <select>
      <value>10.0.0.1</value>
    </select>
  </result>
</query-result>
```

If we try to get more data with the `fetch-query-result` we might get more `result` entries in return until no more data exists and we get an empty query result back:

```
<query-result xmlns="http://tail-f.com/ns/netconf/query">
</query-result>
```

If we want to send the query and get the entire result with only one request, we can do this by using `immediate-query`. This function takes similar arguments as `start-query` and returns the entire result analogous `fetch-query-result`. Note that it is not possible to paginate or set an offset start node for the result list; i.e. the options `limit` and `offset` are ignored.

An example request and response:

```
<immediate-query xmlns="http://tail-f.com/ns/netconf/query">
  <foreach>
    /x/host[enabled = 'true']
  </foreach>
  <select>
    <label>Host name</label>
    <expression>name</expression>
    <result-type>string</result-type>
  </select>
  <select>
    <expression>address</expression>
    <result-type>string</result-type>
  </select>
  <sort-by>name</sort-by>
  <timeout>600</timeout>
</immediate-query>
```

```
<query-result xmlns="http://tail-f.com/ns/netconf/query">
  <result>
    <select>
      <label>Host name</label>
      <value>One</value>
    </select>
    <select>
      <value>10.0.0.1</value>
    </select>
  </result>
  <result>
    <select>
      <label>Host name</label>
      <value>Three</value>
    </select>
    <select>
      <value>10.0.0.3</value>
    </select>
  </result>
</query-result>
```

If we want to go back in the "stream" of received data chunks and have them repeated, we can do that with the `reset-query` operation. In the example below we ask to get results from the 42:nd result entry:

```
<reset-query xmlns="http://tail-f.com/ns/netconf/query">
  <query-handle>12345</query-handle>
  <offset>42</offset>
</reset-query>
```

Finally, when we are done we stop the query:

```
<stop-query xmlns="http://tail-f.com/ns/netconf/query">
  <query-handle>12345</query-handle>
</stop-query>
```

Meta-data in Attributes

NSO supports three pieces of meta-data data nodes: tags, annotations, and inactive.

An annotation is a string which acts a comment. Any data node present in the configuration can get an annotation. An annotation does not affect the underlying configuration, but can be set by a user to comment what the configuration does.

An annotation is encoded as an XML attribute 'annotation' on any data node. To remove an annotation, set the 'annotation' attribute to an empty string.

Any configuration data node can have a set of tags. Tags are set by the user for data organization and filtering purposes. A tag does not affect the underlying configuration.

All tags on a data node are encoded as a space separated string in an XML attribute 'tags'. To remove all tags, set the 'tags' attribute to an empty string.

Annotation, tags, and inactive attributes can be present in `<edit-config>`, `<copy-config>`, `<get-config>`, and `<get>`. For example:

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target>
      <running/>
    </target>
    <config>
      <interfaces xmlns="http://example.com/ns/if">
        <interface annotation="this is the management interface"
                   tags=" important ethernet ">
          <name>eth0</name>
          ...
        </interface>
      </interfaces>
    </config>
  </edit-config>
</rpc>
```

Namespace for Additional Error Information

NSO adds an additional namespace which is used to define elements which are included in the `<error-info>` element. This namespace also describes which `<error-app-tag/>` elements the server might generate, as part of an `<rpc-error/>`.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://tail-f.com/ns/netconf/params/1.1"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xml:lang="en">

  <xs:annotation>
    <xs:documentation>
      Tail-f's namespace for additional error information.
      This namespace is used to define elements which are included
      in the 'error-info' element.

      The following are the app-tags used by the NETCONF agent:

        o  not-writable

          Means that an edit-config or copy-config operation was
          attempted on an element which is read-only
          (i.e. non-configuration data).

        o  missing-element-in-choice
```

```
          Like the standard error missing-element, but generated when
          one of a set of elements in a choice is missing.

        o  pending-changes

          Means that a lock operation was attempted on the candidate
          database, and the candidate database has uncommitted
          changes. This is not allowed according to the protocol
          specification.

        o  url-open-failed

          Means that the URL given was correct, but that it could not
          be opened. This can e.g. be due to a missing local file, or
          bad ftp credentials. An error message string is provided in
          the &lt;error-message&gt; element.

        o  url-write-failed

          Means that the URL given was opened, but write failed. This
          could e.g. be due to lack of disk space. An error message
          string is provided in the &lt;error-message&gt; element.

        o  bad-state

          Means that an rpc is received when the session is in a state
          which don't accept this rpc.  An example is
          &lt;prepare-transaction&gt; before &lt;start-transaction&gt;

  </xs:documentation>
</xs:annotation>

<xs:element name="bad-keyref">
  <xs:annotation>
    <xs:documentation>
      This element will be present in the 'error-info' container when
      'error-app-tag' is "instance-required".
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element name="bad-element" type="xs:string">
        <xs:annotation>
          <xs:documentation>
            Contains an absolute XPath expression pointing to the element
            which value refers to a non-existing instance.
          </xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="missing-element" type="xs:string">
        <xs:annotation>
          <xs:documentation>
            Contains an absolute XPath expression pointing to the missing
            element referred to by 'bad-element'.
          </xs:documentation>
        </xs:annotation>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="bad-instance-count">
```

```
    <xs:annotation>
      <xs:documentation>
        This element will be present in the 'error-info' container when
        'error-app-tag' is "too-few-elements" or "too-many-elements".
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="bad-element" type="xs:string">
          <xs:annotation>
            <xs:documentation>
              Contains an absolute XPath expression pointing to an
              element which exists in too few or too many instances.
            </xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:element name="instances" type="xs:unsignedInt">
          <xs:annotation>
            <xs:documentation>
              Contains the number of existing instances of the element
              referd to by 'bad-element'.
            </xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:choice>
          <xs:element name="min-instances" type="xs:unsignedInt">
            <xs:annotation>
              <xs:documentation>
                Contains the minimum number of instances that must
                exist in order for the configuration to be consistent.
                This element is present only if 'app-tag' is
                'too-few-elems'.
              </xs:documentation>
            </xs:annotation>
          </xs:element>
          <xs:element name="max-instances" type="xs:unsignedInt">
            <xs:annotation>
              <xs:documentation>
                Contains the maximum number of instances that can
                exist in order for the configuration to be consistent.
                This element is present only if 'app-tag' is
                'too-many-elems'.
              </xs:documentation>
            </xs:annotation>
          </xs:element>
        </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:attribute name="annotation" type="xs:string">
    <xs:annotation>
      <xs:documentation>
        This attribute can be present on any configuration data node.  It
        acts as a comment for the node.  The annotation does not affect the
        underlying configuration data.
      </xs:documentation>
    </xs:annotation>
  </xs:attribute>

  <xs:attribute name="tags" type="xs:string">
    <xs:annotation>
```

```
      <xs:documentation>
        This attribute can be present on any configuration data node.  It
        is a space separated string of tags for the node.  The tags of a
        node does not affect the underlying configuration data, but can
        be used by a user for data organization, and data filtering.
      </xs:documentation>
    </xs:annotation>
  </xs:attribute>

</xs:schema>
```

CHAPTER 3

The RESTCONF API

Introduction

RESTCONF is an HTTP based protocol as defined in RFC 8040. RESTCONF standardizes a mechanism to allow Web applications to access the configuration data, state data, data-model-specific Remote Procedure Call (RPC) operations, and event notifications within a networking device.

RESTCONF uses HTTP methods to provide Create, Read, Update, Delete (CRUD) operations on a conceptual datastore containing YANG-defined data, which is compatible with a server that implements NETCONF datastores as defined in RFC 6241

Configuration data and state data are exposed as resources that can be retrieved with the GET method. Resources representing configuration data can be modified with the DELETE, PATCH, POST, and PUT methods. Data is encoded with either XML (W3C.REC-xml-20081126) or JSON (RFC 7159)

This chapter describes the NSO implementation and extension to or deviation from RFC 8040 respectively.

As of this writing, the server supports the following specifications:

- RFC 6020 - YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)
- RFC 6021 - Common YANG Data Types
- RFC 6470 - NETCONF Base Notifications
- RFC 6536 - NETCONF Access Control Model
- RFC 6991 - Common YANG Data Types
- RFC 7950 - The YANG 1.1 Data Modeling Language
- RFC 7951 - JSON Encoding of Data Modeled with YANG
- RFC 7952 - Defining and Using metadata with YANG
- RFC 8040 - RESTCONF Protocol
- RFC 8072 - YANG Patch Media Type
- RFC 8341 - Network Configuration Access Control Model
- RFC 8525 - YANG Library
- RFC 8528 - YANG Schema Mount

Getting started

In order to enable RESTCONF in NSO, RESTCONF must be enabled in the `ncs.conf` configuration file. The web server configuration for RESTCONF is shared with the WebUI's config, but you may define a separate RESTCONF transport section. The WebUI does not have to be enabled for RESTCONF to work.

Here is a minimal example of what is needed in the `ncs.conf`.

Example 1. NSO configuration for RESTCONF

```
<restconf>
  <enabled>true</enabled>
</restconf>

<webui>
  <transport>
    <tcp>
      <enabled>true</enabled>
      <ip>0.0.0.0</ip>
      <port>8080</port>
    </tcp>
  </transport>
</webui>
```

If you want to run RESTCONF with a different transport configuration than what the WebUI is using, you can specify a separate RESTCONF transport section.

Example 2. NSO separate transport configuration for RESTCONF

```
<restconf>
  <enabled>true</enabled>
  <transport>
    <tcp>
      <enabled>true</enabled>
      <ip>0.0.0.0</ip>
      <port>8090</port>
    </tcp>
  </transport>
</restconf>

<webui>
  <enabled>false</enabled>
  <transport>
    <tcp>
      <enabled>true</enabled>
      <ip>0.0.0.0</ip>
      <port>8080</port>
    </tcp>
  </transport>
</webui>
```

It is now possible to do a RESTCONF requests towards NSO. Any HTTP client can be used, in the following examples *curl* will be used. The example below will show how a typical RESTCONF request could look like.

Example 3. A RESTCONF request using 'curl'

```
# Note that the command is wrapped in several lines in order to fit.
#
# The switch '-i' will include any HTTP reply headers in the output
# and the '-s' will suppress some superflous output.
#
# The '-u' switch specify the User:Password for login authentication.
#
# The '-H' switch will add a HTTP header to the request; in this case
# an 'Accept' header is added, requesting the preferred reply format.
#
# Finally, the complete URL to the wanted resource is specified,
# in this case the top of the configuration tree.
#
curl -is -u admin:admin \
-H "Accept: application/yang-data+xml" \
http://localhost:8080/restconf/data
```

In the rest of the document, in order to simplify the presentation, the example above will be expressed as:

Example 4. A RESTCONF request, simplified

```
GET /restconf/data
Accept: application/yang-data+xml

# Any reply with relevant headers will be displayed here!
HTTP/1.1 200 OK
```

Note the HTTP return code (200 OK) in the example, which will be displayed together with any relevant HTTP headers returned and a possible body of content.

Top-level GET request

Send a RESTCONF query to get a representation of the top-level resource, which is accessible through the path: */restconf*.

Example 5. A top-level RESTCONF request

```
GET /restconf
Accept: application/yang-data+xml

HTTP/1.1 200 OK
<restconf xmlns="urn:ietf:params:xml:ns:yang:ietf-restconf">
  <data/>
  <operations/>
  <yang-library-version>2019-01-04</yang-library-version>
</restconf>
```

As can be seen from the result, the server exposes three additional resources:

- *data* : this mandatory resource represents the combined configuration and state data resources that can be accessed by a client.
- *operations* : this optional resource is a container that provides access to the data-model-specific RPC operations supported by the server.
- *yang-library-version* : this mandatory leaf identifies the revision date of the "ietf-yang-library" YANG module that is implemented by this server. This resource exposes which YANG modules are in use by NSO system.

Get resources under the *data* resource

To fetch configuration, operational data, or both, from the server, a request to the *data* resource is made. In order to restrict the amount of returned data, the following example will prune the amount of output to only consist of the top most nodes. This is achieved by using the *depth* query argument as shown in the example below:

Example 6. Get the top most resources under the *data*

```
GET /restconf/data?depth=1
Accept: application/yang-data+xml

<data xmlns="urn:ietf:params:xml:ns:yang:ietf-restconf">
  <yang-library xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library"/>
  <modules-state xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library"/>
  <dhcp xmlns="http://yang-central.org/ns/example/dhcp"/>
  <nacm xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-acm"/>
  <netconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring"/>
  <restconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-restconf-monitoring"/>
  <aaa xmlns="http://tail-f.com/ns/aaa/1.1"/>
  <confd-state xmls="http://tail-f.com/yang/confd-monitoring"/>
  <last-logins xmlns="http://tail-f.com/yang/last-login"/>
</data>
```

Manipulating config data with RESTCONF

Let's assume we are interested in the *dhcp/subnet* resource in our configuration. In the following examples, assume it is defined by a corresponding Yang module that we have named *dhcp.yang*, looking like this:

Example 7. The *dhcp.yang* resource

```
> yanger -f tree examples.confd/restconf/basic/dhcp.yang
```

```
module: dhcp
  +--rw dhcp
  +--rw max-lease-time?       uint32
  +--rw default-lease-time?   uint32
  +--rw subnet* [net]
  |  +--rw net               inet:ip-prefix
  |  +--rw range!
  |  |  +--rw dynamic-bootp?   empty
  |  |  +--rw low              inet:ip-address
  |  |  +--rw high             inet:ip-address
  |  +--rw dhcp-options
  |  |  +--rw router*        inet:host
  |  |  +--rw domain-name?   inet:domain-name
  |  +--rw max-lease-time?   uint32
```

We can issue a HTTP GET request to retrieve the value content of the resource. In this case we find that there is no such data, which is indicated by the HTTP return code *204 No Content*.

Note also how we have prefixed the *dhcp:dhcp* resource. This is how RESTCONF handle namespaces, where the prefix is the YANG module name and the namespace is as defined by the *namespace* statement in the YANG module.

Example 8. Get the *dhcp/subnet* resource

```
GET /restconf/data/dhcp:dhcp/subnet

HTTP/1.1 204 No Content
```

We can now create the *dhcp/subnet* resource by sending a HTTP POST request + the data that we want to store. Note the *Content-Type* HTTP header, which indicates the format of the provided body. Two formats is supported: *XML* or *JSON*. In this example we are using XML, which is indicated by the Content-Type value: *application/yang-data+xml*.

Example 9. Create a new *dhcp/subnet* resource

```
POST /restconf/data/dhcp:dhcp
Content-Type: application/yang-data+xml

<subnet xmlns="http://yang-central.org/ns/example/dhcp"
          xmlns:dhcp="http://yang-central.org/ns/example/dhcp">
  <net>10.254.239.0/27</net>
  <range>
    <dynamic-bootp/>
    <low>10.254.239.10</low>
    <high>10.254.239.20</high>
  </range>
  <dhcp-options>
    <router>rtr-239-0-1.example.org</router>
    <router>rtr-239-0-2.example.org</router>
  </dhcp-options>
  <max-lease-time>1200</max-lease-time>
</subnet>

# If the resource is created, the server might respond as follows:

HTTP/1.1 201 Created
Location: http://localhost:8080/restconf/data/dhcp:dhcp/subnet=10.254.239.0%2F27
```

Note the HTTP return code (201 Created) indicating that the resource was successfully created. We also got a *Location* header, which always is returned in a reply to a successful creation of a resource, stating the resulting URI leading to the created resource.

If we now want to modify a part of our *dhcp/subnet* config, we can use the HTTP PATCH method, as shown below. Note that the URI used in the request need to be URL-encoded, such that the key value: *10.254.239.0/27* is URL-encoded as: *10.254.239.0%2F27*.

Also, note the difference of the PATCH URI compared to the earlier *POST* request. With the latter, since the resource does not yet exist, we POST to the parent resource (*dhcp:dhcp*), while with the PATCH request we address the (existing) resource (*10.254.239.0%2F27*).

Example 10. Modify a part of the *dhcp/subnet* resource

```
PATCH /restconf/data/dhcp:dhcp/subnet=10.254.239.0%2F27

<subnet>
  <max-lease-time>3333</max-lease-time>
</subnet>

# If our modification is successful, the server might respond as follows:

HTTP/1.1 204 No Content
```

We can also replace the subnet with some new configuration. To do this we make use of the *PUT* HTTP method as shown below. Since the operation was successful and no body was returned, we will get a *204 No Content* return code.

Example 11. Replace a *dhcp/subnet* resource

```
PUT /restconf/data/dhcp:dhcp/subnet=10.254.239.0%2F27
Content-Type: application/yang-data+xml

<subnet xmlns="http://yang-central.org/ns/example/dhcp"
          xmlns:dhcp="http://yang-central.org/ns/example/dhcp">
  <net>10.254.239.0/27</net>

  <!-- ...config left out here... -->

</subnet>

# At success, the server will respond as follows:

HTTP/1.1 204 No Content
```

To delete the subnet we make use of the *DELETE* HTTP method as shown below. Since the operation was successful and no body was returned, we will get a *204 No Content* return code.

Example 12. Delete a *dhcp/subnet* resource

```
DELETE /restconf/data/dhcp:dhcp/subnet=10.254.239.0%2F27

HTTP/1.1 204 No Content
```

Root resource discovery

RESTCONF makes it possible to specify where the RESTCONF API is located, as described in the RESTCONF RFC 8040.

As per default, the RESTCONF API root is */restconf*. Typically there is no need to change the default value although it is possible to change this by configuring the RESTCONF API root in the `ncs.conf` file as:

Example 13. NSO configuration for RESTCONF

```
<restconf>
```

```
  <enabled>true</enabled>
  <root-resource>my_own_restconf_root</root-resource>
</restconf>
```

The RESTCONF API root will now be */my_own_restconf_root*.

A client may discover the root resource by getting the */.well-known/host-meta* resource as shown in the example below:

Example 14. Example returning */restconf*

```
   The client might send the following:

      GET /.well-known/host-meta
      Accept: application/xrd+xml

   The server might respond as follows:

      HTTP/1.1 200 OK

      <XRD xmlns='http://docs.oasis-open.org/ns/xri/xrd-1.0'>
          <Link rel='restconf' href='/restconf'/>
      </XRD>
```

Note In this document, all examples will assume the RESTCONF API root to be */restconf*.

Capabilities

A RESTCONF capability is a set of functionality that supplements the base RESTCONF specification. The capability is identified by a uniform resource identifier (URI). The RESTCONF server includes a *capability* URI leaf-list entry identifying each supported protocol feature. This include the *basic-mode* default-handling mode, optional query parameters and may also include other, NSO specific, capability URIs.

How to view the capabilities of the RESTCONF server

To view currently enabled capabilities, use the ietf-restconf-monitoring YANG model, which is available as: */restconf/data/ietf-restconf-monitoring:restconf-state* .

Example 15. NSO RESTCONF capabilities

```
GET /restconf/data/ietf-restconf-monitoring:restconf-state
Host: example.com
Accept: application/yang-data+xml

<restconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-restconf-monitoring"
  xmlns:rcmon="urn:ietf:params:xml:ns:yang:ietf-restconf-monitoring">
<capabilities>
  <capability>
    urn:ietf:params:restconf:capability:defaults:1.0?basic-mode=explicit
  </capability>
  <capability>urn:ietf:params:restconf:capability:depth:1.0</capability>
  <capability>urn:ietf:params:restconf:capability:fields:1.0</capability>
  <capability>urn:ietf:params:restconf:capability:with-defaults:1.0</capability>
  <capability>urn:ietf:params:restconf:capability:filter:1.0</capability>
  <capability>urn:ietf:params:restconf:capability:replay:1.0</capability>
  <capability>http://tail-f.com/ns/restconf/collection/1.0</capability>
  <capability>http://tail-f.com/ns/restconf/query-api/1.0</capability>
  <capability>http://tail-f.com/ns/restconf/partial-response/1.0</capability>
  <capability>http://tail-f.com/ns/restconf/unhide/1.0</capability>
</capabilities>
</restconf-state>
```

The *defaults* capability

This Capability identifies the *basic-mode* default-handling mode that is used by the server for processing default leafs in requests for data resources.

Example 16. The default capability URI

```
          urn:ietf:params:restconf:capability:defaults:1.0
```

The capability URL will contain a query parameter named *basic-mode* which value tells us what the default behaviour of the RESTCONF server is when it returns a leaf. The possible values are shown in the table below:

Table 17. *basic-mode* values

Value	Description
report-all	Values set to the YANG default value are reported.
trim	Values set to the YANG default value are not reported.
explicit	Values that has been set by a client to the YANG default value will be reported.

The values presented in the table above can also be used by the Client together with the *with-defaults* query parameter in order to override the default RESTCONF server behaviour. Added to these values, the Client can also use the *report-all-tagged* value.

Table 18. Additional *with-defaults* value

Value	Description
report-all-tagged	Works as the *report-all* but a default value will include a XML/JSON attribute to indicate that the value is in fact a default value.

Refering back to the example: Example 15, “NSO RESTCONF capabilities”, where the RESTCONF server returned the default capability:

```
urn:ietf:params:restconf:capability:defaults:1.0?basic-mode=explicit
```

It tells us that values that has been set by a client to the YANG default value will be reported but default values that has not been set by the Client will not be returned. Again, note that this is the default RESTCONF server behaviour which can be overridden by the Client by using the *with-defaults* query argument.

Query parameter capabilities

A set of optional RESTCONF Capability URIs are defined to identify the specific query parameters that are supported by the server. They are defined as:

Table 19. Query parameter capabilities

Name	URI
depth	urn:ietf:params:restconf:capability:depth:1.0
fields	urn:ietf:params:restconf:capability:fields:1.0
filter	urn:ietf:params:restconf:capability:filter:1.0
replay	urn:ietf:params:restconf:capability:replay:1.0

Name	URI
with.defaults	urn:ietf:params:restconf:capability:with.defaults:1.0

For a description of the query parameter functionality see the chapter the section called “Query Parameters” .

Query Parameters

Each RESTCONF operation allows zero or more query parameters to be present in the request URI. Query parameters can be given in any order, but can appear at most once. Supplying query parameters when invoking RPCs and actions is not supported, if supplied the response will be 400 (Bad Request) and the *error-app-tag* will be set to *invalid-value*. However, the query parameters *trace-id* and *unhide* are exempted from this rule and supported for RPC and action invocation. The defined query parameters and in what type of HTTP request they can be used are shown in the table below:

Table 20. Query parameters

Name	Method	Description
content	GET,HEAD	Select config and/or non-config data resources.
depth	GET,HEAD	Request limited subtree depth in the reply content.
fields	GET,HEAD	Request a subset of the target resource contents.
exclude	GET,HEAD	Exclude a subset of the target resource contents.
filter	GET,HEAD	Boolean notification filter for event stream resources.
insert	POST,PUT	Insertion mode for *ordered-by user* data resources
point	POST,PUT	Insertion point for *ordered-by user* data resources
start-time	GET,HEAD	Replay buffer start time for event stream resources.
stop-time	GET,HEAD	Replay buffer stop time for event stream resources.
with-defaults	GET,HEAD	Control the retrieval of default values.
with-origin	GET	Include the "origin" metadata annotations, as detailed in the NMDA.

The *content* Query Parameter

The *content* query parameter controls if *configuration*, *non-configuration* or both types of data should be returned.

The allowed values are:

Table 21. The *content* query parameter values

Value	Description
config	Return only configuration descendant data nodes.
nonconfig	Return only non-configuration descendant data nodes.
all	Return all descendant data nodes.

The *depth* Query Parameter

The *depth* query parameter is used to limit the depth of subtrees returned by the server. Data nodes with a value greater than the *depth* parameter are not returned in a response for a GET request.

The value of the *depth* parameter is either an integer between 1 and 65535 or the string "unbounded". The default value is: "unbounded".

The *fields* Query Parameter

The *fields* query parameter is used to optionally identify data nodes within the target resource to be retrieved in a GET method. The client can use this parameter to retrieve a subset of all nodes in a resource.

For a full definition of the *fields* value can be constructed, refer to the RFC 8040, Section 4.8.3.

NOTE: The *fields* query parameter cannot be used together with the *exclude* query parameter. This will result in an error.

Example 22. Example of how to use the *fields* query parameter

```
GET /restconf/data/dhcp:dhcp?fields=subnet/range(low;high)
Accept: application/yang-data+xml

HTTP/1.1 200 OK
<dhcp xmlns="http://yang-central.org/ns/example/dhcp" \
      xmlns:dhcp="http://yang-central.org/ns/example/dhcp">
  <subnet>
    <range>
      <low>10.254.239.10</low>
      <high>10.254.239.20</high>
    </range>
  </subnet>
  <subnet>
    <range>
      <low>10.254.244.10</low>
      <high>10.254.244.20</high>
    </range>
  </subnet>
</dhcp>
```

The *exclude* Query Parameter

The *exclude* query parameter is used to optionally exclude data nodes within the target resource from being retrieved with a GET request. The client can use this parameter to exclude a subset of all nodes in a resource. Only nodes below the target resource can be excluded, not the target resource itself.

NOTE: The *exclude* query parameter cannot be used together with the *fields* query parameter. This will result in an error.

The *exclude* query parameter uses the same syntax and have the same restrictions as the *fields* query parameter, as defined in RFC 8040, Section 4.8.3.

Selecting multiple nodes to exclude can be done the same way as for the *fields* query parameter, as described in RFC 8040, Section 4.8.3.

exclude using wildcards (*) will exclude all child nodes of the node. For lists and presence containers the parent node will be visible in the output but not it's children, i.e. it will be displayed as an empty node. For non-presence containers the parent node will be excluded from the output as well.

exclude can be used together with the *depth* query parameter to limit the depth of the output. In contrast to *fields*, where *depth* is counted from the node selected by *fields*, for *exclude* the depth is counted from the target resource, and the nodes are excluded if *depth* is deep enough to encounter an excluded node.

Example 23. Example of how to use the *exclude* query parameter

When exclude is not used:

```
GET /restconf/data/dhcp:dhcp/subnet
Accept: application/yang-data+xml

HTTP/1.1 200 OK
<subnet xmlns="http://yang-central.org/ns/example/dhcp"
          xmlns:dhcp="http://yang-central.org/ns/example/dhcp">
  <net>10.254.239.0/27</net>
  <range>
    <dynamic-bootp/>
    <low>10.254.239.10</low>
    <high>10.254.239.20</high>
  </range>
  <dhcp-options>
    <router>rtr-239-0-1.example.org</router>
    <router>rtr-239-0-2.example.org</router>
  </dhcp-options>
  <max-lease-time>1200</max-lease-time>
</subnet>
```

Using *exclude* to exclude "low" and "high" from "range", note that these are absent in the output:

```
GET /restconf/data/dhcp:dhcp/subnet?exclude=range(low;high)
Accept: application/yang-data+xml

HTTP/1.1 200 OK
<subnet xmlns="http://yang-central.org/ns/example/dhcp"
          xmlns:dhcp="http://yang-central.org/ns/example/dhcp">
  <net>10.254.239.0/27</net>
  <range>
    <dynamic-bootp/>
  </range>
  <dhcp-options>
    <router>rtr-239-0-1.example.org</router>
    <router>rtr-239-0-2.example.org</router>
  </dhcp-options>
  <max-lease-time>1200</max-lease-time>
</subnet>
```

The *filter*, *start-time* and *stop-time* Query Parameters

These query parameters are only allowed on an event stream resource and is further described in the chapter: the section called “Streams” .

The *insert* Query Parameter

The *insert* query parameters is used to specify how a resource should be inserted within an *ordered-by user* list. The allowed values are as shown in the table below.

Table 24. The *content* query parameter values

Value	Description
first	Insert the new data as the new first entry.
last	Insert the new data as the new last entry. This is the *default value*.
before	Insert the new data before the insertion point, as specified by the value of the *point* parameter.
after	Insert the new data after the insertion point, as specified by the value of the *point* parameter.

This parameter is only valid if the target data represents a YANG list or leaf-list that is *ordered-by user*. In the example below we will insert a new *router* value, first, in the *ordered-by user* leaf-list of *dhcp-options/ router* values. Remember that the default behaviour is for new entries to be inserted last in an *ordered-by user* leaf-list.

Example 25. Insert *first* into a *ordered-by user* leaf-list

```
# Note: we have to split the POST line in order to fit the page
POST /restconf/data/dhcp:dhcp/subnet=10.254.239.0%2F27/dhcp-options?\
     insert=first
Content-Type: application/yang-data+xml

<router>one.acme.org</router>

# If the resource is created, the server might respond as follows:

HTTP/1.1 201 Created
Location /restconf/data/dhcp:dhcp/subnet=10.254.239.0%2F27/dhcp-options/\
         router=one.acme.org
```

To verify that the router value really ended up first:

```
GET /restconf/data/dhcp:dhcp/subnet=10.254.239.0%2F27/dhcp-options
Accept: application/yang-data+xml

HTTP/1.1 200 OK
<dhcp-options xmlns="http://yang-central.org/ns/example/dhcp"
              xmlns:dhcp="http://yang-central.org/ns/example/dhcp">
  <router>one.acme.org</router>
  <router>rtr-239-0-1.example.org</router>
  <router>rtr-239-0-2.example.org</router>
</dhcp-options>
```

The *point* Query Parameter

The *point* query parameters is used to specify the insertion point for a data resource that is being created or moved within an *ordered-by user* list or leaf-list. In the example below we will insert the new *router* value: *two.acme.org*, after the first value: *one.acme.org* in the *ordered-by user* leaf-list of *dhcp-options/router* values.

Example 26. Insert *first* into a *ordered-by user* leaf-list

```
# Note: we have to split the POST line in order to fit the page
POST /restconf/data/dhcp:dhcp/subnet=10.254.239.0%2F27/dhcp-options?\
     insert=after&\
     point=/dhcp:dhcp/subnet=10.254.239.0%2F27/dhcp-options/router=one.acme.org
Content-Type: application/yang-data+xml

<router>two.acme.org</router>

# If the resource is created, the server might respond as follows:

HTTP/1.1 201 Created
Location /restconf/data/dhcp:dhcp/subnet=10.254.239.0%2F27/dhcp-options/\
         router=one.acme.org
```

To verify that the router value really ended up after our insertion point:

```
GET /restconf/data/dhcp:dhcp/subnet=10.254.239.0%2F27/dhcp-options
Accept: application/yang-data+xml

HTTP/1.1 200 OK
<dhcp-options xmlns="http://yang-central.org/ns/example/dhcp"
```

```
              xmlns:dhcp="http://yang-central.org/ns/example/dhcp">
  <router>one.acme.org</router>
  <router>two.acme.org</router>
  <router>rtr-239-0-1.example.org</router>
  <router>rtr-239-0-2.example.org</router>
</dhcp-options>
```

Additional query parameters

There are additional NSO query parameters available for the RESTCONF API. These additional query parameters are described below.

Table 27. Additional Query Parameters

Name	Methods	Description
dry-run	POST, PUT, PATCH, DELETE	Validate and display the configuration changes but do not perform the actual commit. *Neither* CDB *nor* the devices are affected. Instead the effects that would have taken place is showed in the returned output. Possible values are: *xml*, *cli* and *native*. The value used specify in what format we want the returned diff to be.
dry-run-reverse	POST, PUT, PATCH, DELETE	Used together with the *dry-run=native* parameter to display the device commands for getting back to the current running state in the network if the commit is successfully executed. Beware that if any changes are done later on the same data the reverse device commands returned are invalid.
no-networking	POST, PUT, PATCH, DELETE	Do not send any data to the devices. This is a way to manipulate CDB in NSO without generating any southbound traffic.
no-out-of-sync-check	POST, PUT, PATCH, DELETE	Continue with the transaction even if NSO detects that a device's configuration is out of sync. Can't be used together with no-overwrite.
no-overwrite	POST, PUT, PATCH, DELETE	NSO will check that the data that should be modified has not changed on the device compared to NSO's view of the data. Can't be used together with no-out-of-sync-check.
no-revision-drop	POST, PUT, PATCH, DELETE	NSO will not run its data model revision algorithm, which requires all participating managed devices to have all parts of the data models for all data contained in this transaction. Thus, this flag forces NSO to never silently drop any data set operations towards a device.
no-deploy	POST, PUT, PATCH, DELETE	Commit without invoking the service create method, i.e, write the service instance data without activating the service(s). The service(s) can later be re-deployed to write the changes of the service(s) to the network.
reconcile	POST, PUT, PATCH, DELETE	Reconcile the service data. All data which existed before the service was created will now be owned by the service. When the service is removed that data will also be removed. In technical terms the reference count will be decreased by one for everything which existed prior to the service. If manually configured data exists below in the configuration tree that data is kept unless the option *discard-non-service-config* is used.

Name	Methods	Description
use-lsa	POST, PUT, PATCH, DELETE	Force handling of the LSA nodes as such. This flag tells NSO to propagate applicable commit flags and actions to the LSA nodes without applying them on the upper NSO node itself. The commit flags affected are `dry-run`, `no-networking`, `no-out-of-sync-check`, `no-overwrite` and `no-revision-drop`.
no-lsa	POST, PUT, PATCH, DELETE	Do not handle any of the LSA nodes as such. These nodes will be handled as any other device.
commit-queue	POST, PUT, PATCH, DELETE	Commit the transaction data to the commit queue. Possible values are: *async*, *sync* and *bypass*. If the *async* value is set the operation returns successfully if the transaction data has been successfully placed in the queue. The *sync* value will cause the operation to not return until the transaction data has been sent to all devices, or a timeout occurs. The *bypass* value means that if `/devices/global-settings/commit-queue/enabled-by-default` is *true* the data in this transaction will bypass the commit queue. The data will be written directly to the devices.
commit-queue-atomic	POST, PUT, PATCH, DELETE	Sets the atomic behaviour of the resulting queue item. Possible values are: *true* and *false*. If this is set to *false*, the devices contained in the resulting queue item can start executing if the same devices in other non-atomic queue items ahead of it in the queue are completed. If set to *true*, the atomic integrity of the queue item is preserved.
commit-queue-block-others	POST, PUT, PATCH, DELETE	The resulting queue item will block subsequent queue items, which use any of the devices in this queue item, from being queued.
commit-queue-lock	POST, PUT, PATCH, DELETE	Place a lock on the resulting queue item. The queue item will not be processed until it has been unlocked, see the actions **unlock** and **lock** in `/devices/commit-queue/queue-item`. No following queue items, using the same devices, will be allowed to execute as long as the lock is in place.
commit-queue-tag	POST, PUT, PATCH, DELETE	The value is a user defined opaque tag. The tag is present in all notifications and events sent referencing the specific queue item.
commit-queue-timeout	POST, PUT, PATCH, DELETE	Specifies a maximum number of seconds to wait for completion. Possible values are *infinity* or a positive integer. If the timer expires, the transaction is kept in the commit-queue, and the operation returns successfully. If the timeout is not set, the operation waits until completion indefinitely.
commit-queue-error-option	POST, PUT, PATCH, DELETE	The error option to use. Depending on the selected error option NSO will store the reverse of the original transaction to be able to undo the transaction changes and get back to the previous state. This data is stored in the `/devices/commit-queue/completed` tree from where it can be viewed and invoked with the **rollback** action. When invoked the data will be removed. Possible values are: *continue-on-error*, *rollback-on-error* and *stop-on-error*. The *continue-*

Name	Methods	Description
		on-error value means that the commit queue will continue on errors. No rollback data will be created. The *rollback-on-error* value means that the commit queue item will roll back on errors. The commit queue will place a lock with `block-others` on the devices and services in the failed queue item. The **rollback** action will then automatically be invoked when the queue item has finished its execution. The lock will be removed as part of the rollback. The *stop-on-error* means that the commit queue will place a lock with `block-others` on the devices and services in the failed queue item. The lock must then either manually be released when the error is fixed or the **rollback** action under `/devices/commit-queue/completed` be invoked. Read about error recovery in the section called “Commit Queue” in *User Guide* for a more detailed explanation.
trace-id	POST, PUT, PATCH, DELETE	Use the provided trace id as part of the log messages emitted while processing. If no trace id is given, NSO is going to generate and assign a trace id to the processing. The trace-id query parameter can also be used with RPCs and actions to relay a trace-id from northbound requests. The trace-id will be included in the X-Cisco-NSO-Trace-ID header in the response.
limit	GET	Used by the client to specify a limited set of list entries to retrieve. See The value of the *limit* parameter is either an integer greater than or equal to 1, or the string *unbounded*. The string *unbounded* is the default value. the section called “Partial Responses” for an example.
offset	GET	Used by the client to specify the number of list elements to skip before returning the requested set of list entries. See The value of the "offset" parameter is an integer greater than or equal to 0. The default value is 0. the section called “Partial Responses” for an example.
rollback-comment	POST, PUT, PATCH, DELETE	Used to specify a comment to be attached to the Rollback File that will be created as a result of the POST operation. This assume that Rollback File handling is enabled.
rollback-label	POST, PUT, PATCH, DELETE	Used to specify a label to be attached to the Rollback File that will be created as a result of the POST operation. This assume that Rollback File handling is enabled.
rollback-id	POST, PUT, PATCH, DELETE	Return the rollback id in the response if a rollback file was created during this operation. This requires rollbacks to be enabled in the NSO to take effect.
with-service-meta-data	GET	Include FASTMAP attributes such as backpointers and reference counters in the reply. These are typically internal to NSO and thus not shown by default.

Edit Collision Prevention

Two edit collision detection and prevention mechanisms are provided in RESTCONF for the datastore resource: a timestamp and an entity-tag. Any change to configuration data resources will update the timestamp and entity-tag of the datastore resource. This makes it possible for a client to apply precondition HTTP headers to a request.

The NSO RESTCONF API honor the following HTTP response headers: *Etag* and *Last-Modified*, and the following request headers: *If-Match*, *If-None-Match*, *If-Modified-Since* and *If-Unmodified-Since*.

Response headers

- `Etag`: This header will contain an *entity-tag* which is an opaque string representing the latest transaction identifier in the NSO database. This header is only available for the *running* datastore and hence, only relates to configuration data (non-operational).
- `Last-Modified`: This header contains the timestamp for the last modification made to the NSO database. This timestamp can be used by a RESTCONF client in subsequent requests, within the *If-Modified-Since* and *If-Unmodified-Since* header fields. This header is only available for the *running* datastore and hence, only relates to configuration data (non-operational).

Request headers

- `If-None-Match`: This header evaluates to true if the supplied value does not match the latest *Etag* entity-tag value. If evaluated to false an error response with status 304 (Not Modified) will be sent with no body. This header carry only meaning if the entity-tag of the *Etag* response header has previously been acquired.

  The usage of this could for example be a HEAD operation to get information if the data has changed since last retrieval.
- `If-Modified-Since`: This request-header field is used with a HTTP method to make it conditional, i.e if the requested resource has not been modified since the time specified in this field, the request will not be processed by the RESTCONF server; instead, a 304 (Not Modified) response will be returned without any message-body.

  Usage of this is for instance for a GET operation to retrieve the information if (and only if) the data has changed since last retrieval. Thus, this header should use the value of a *Last-Modified* response header that has previously been acquired.
- `If-Match`: This header evaluates to true if the supplied value matches the latest *Etag* value. If evaluated to false an error response with status 412 (Precondition Failed) will be sent with no body. This header carry only meaning if the entity-tag of the *Etag* response header has previously been acquired.

  The usage of this can be in case of a PUT, where *If-Match* can be used to prevent the lost update problem. It can check if the modification of a resource that the user wants to upload will not override another change that has been done since the original resource was fetched.
- `If-Unmodified-Since`: This header evaluates to true if the supplied value has not been last modified after the given date. If the resource has been modified after the given date, the response will be a 412 (Precondition Failed) error with no body. This header carry only meaning if the `Last-Modified` response header has previously been acquired.

  The usage of this can be the case of a POST, where editions are rejected if the stored resource has been modified since the original value was retrieved.

Using Rollbacks

Rolling back configuration changes

If rollbacks have been enabled in the configuration using the *rollback-id* query parameter, the fixed id of the rollback file creating during an operation is returned in the results. The below examples shows creation of a new resource and removal of that resource using the rollback created in the first step.

Example 28. Create a new *dhcp/subnet* resource

```
POST /restconf/data/dhcp:dhcp?rollback-id=true
Content-Type: application/yang-data+xml

<subnet xmlns="http://yang-central.org/ns/example/dhcp">
  <net>10.254.239.0/27</net>
</subnet>

HTTP/1.1 201 Created
Location: http://localhost:8008/restconf/data/dhcp:dhcp/subnet=10.254.239.0%2F27

<result xmlns="http://tail-f.com/ns/tailf-restconf">
<rollback>
  <id>10002</id>
</rollback>
</result>
```

Then using the fixed id returned above as input to the *apply-rollback-file* action:

```
POST /restconf/data/tailf-rollback:rollback-files/apply-rollback-file
Content-Type: application/yang-data+xml

<input xmlns="http://tail-f.com/ns/rollback">
  <fixed-number>10002</fixed-number>
</input>

HTTP/1.1 204 No Content
```

Streams

Introduction

The RESTCONF protocol supports YANG-defined event notifications. The solution preserves aspects of NETCONF event notifications [RFC5277] while utilizing the Server-Sent Events, W3C.REC-eventsource-20150203, transport strategy.

RESTCONF event notification streams are described in Sections 6 and 9.2 of RFC 8040, where also notification examples can be found.

RESTCONF event notification is a way for RESTCONF clients to retrieve notifications for different event streams. Event streams configured in NSO can be subscribed to using different channels such as the *RESTCONF* or the *NETCONF* channel.

More information on how to define a new notification event using Yang is described in RFC 6020.

How to add and configure notifications support in NSO is described in the `ncs.conf(3)` man page.

The design of RESTCONF event notification is inspired by how NETCONF event notification is designed. More information on NETCONF event notification can be found in RFC 5277.

Configuration

For this example we will define a notification stream, named *interface* in the `ncs.conf` configuration file as shown below.

We also enable the builtin replay store which means that NSO automatically stores all notifications on disk, ready to be replayed should a RESTCONF event notification subscriber ask for logged notifications. The replay store uses a set of wrapping log files on disk (of a certain number and size) to store the notifications.

Example 29. Configure an example notification

```
<notifications>
  <eventStreams>
    <stream>
      <name>interface</name>
      <description>Example notifications</description>
      <replaySupport>true</replaySupport>
      <builtinReplayStore>
        <dir>./</dir>
        <maxSize>S1M</maxSize>
        <maxFiles>5</maxFiles>
      </builtinReplayStore>
    </stream>
  </eventStreams>
</notifications>
```

To view the currently enabled event streams, use the ietf-restconf-monitoring YANG model. The streams are available under the */restconf/data/ietf-restconf-monitoring:restconf-state/streams* container.

Example 30. View the example RESTCONF stream

```
GET /restconf/data/ietf-restconf-monitoring:restconf-state/streams
Accept: application/yang-data+xml

HTTP/1.1 200 OK

<streams xmlns="urn:ietf:params:xml:ns:yang:ietf-restconf-monitoring"
         xmlns:rcmon="urn:ietf:params:xml:ns:yang:ietf-restconf-monitoring">

  ...other streams info removed here for brewity reason...

  <stream>
    <name>interface</name>
    <description>Example notifications</description>
    <replay-support>true</replay-support>
    <replay-log-creation-time>
      2020-05-04T13:45:31.033817+00:00
    </replay-log-creation-time>
    <access>
      <encoding>xml</encoding>
      <location>https://localhost:8888/restconf/streams/interface/xml</location>
    </access>
    <access>
      <encoding>json</encoding>
      <location>https://localhost:8888/restconf/streams/interface/json</location>
    </access>
  </stream>
</streams>
```

Note the URL value we get in the *location* element in the example above. This URL should be used when subscribing to the notification events as is shown in the next example.

Subscribe to notification events

RESTCONF clients can determine the URL for the subscription resource (to receive notifications) by sending an HTTP GET request for the *location* leaf with the *stream* list entry. The value returned by the server can be used for the actual notification subscription.

The client will send an HTTP GET request for the (location) URL returned by the server with the *Accept* type *text/event-stream* as shown in the example below. Note that this request works like a *long polling* request which means that the request will not return. Instead, server side notifications will be sent to the client where each line of the notification will be prepended with *data:* .

Example 31. View the example RESTCONF stream

```
GET /restconf/streams/interface/xml
Accept: text/event-stream

   ...NOTE: we will be waiting here until a notification is generated...

HTTP/1.1 200 OK
Content-Type: text/event-stream

data: <notification xmlns='urn:ietf:params:xml:ns:netconf:notification:1.0'>
data:     <eventTime>2020-05-04T13:48:02.291816+00:00</eventTime>
data:     <link-up xmlns='http://tail-f.com/ns/test/notif'>
data:       <if-index>2</if-index>
data:       <link-property>
data:         <newly-added/>
data:         <flags>42</flags>
data:         <extensions>
data:           <name>1</name>
data:           <value>3</value>
data:         </extensions>
data:         <extensions>
data:           <name>2</name>
data:           <value>4668</value>
data:         </extensions>
data:       </link-property>
data:     </link-up>
data: </notification>

   ...NOTE: we will still be waiting here for more notifications to come...
```

Since we have enabled the replay store, we can ask the server to replay any notifications generated since the specific date we specify. After those notifications have been delivered we will continue waiting for new notifications to be generated.

Example 32. View the example RESTCONF stream

```
GET /restconf/streams/interface/xml?start-time=2007-07-28T15%3A23%3A36Z
Accept: text/event-stream
```

```
HTTP/1.1 200 OK
Content-Type: text/event-stream

data: ...any existing notification since given date will be delivered here...

   ...NOTE: when all notifications are delivered, we will be waiting here for more...
```

Errors

Errors occurring during streaming of events will be reported as Server-Sent Events (SSE) comments as described in W3C.REC-eventsource-20150203 as shown in the example below.

Example 33. NSO RESTCONF errors during streaming

```
: error: notification stream NETCONF temporarily unavailable
```

Schema resource

RFC 8040, Section 3.7 describes retrieval of YANG modules used by the server via the RPC operation `get-schema`. The YANG source is made available by NSO in two ways: compiled into the fxs file or put in the loadPath. See the section called “Monitoring of the NETCONF Server”.

The example below show how to list the available Yang modules. Since we are interested by the *dhcp* module we only show that part of the output:

Example 34. List the available Yang modules

```
GET /restconf/data/ietf-yang-library:modules-state
Accept: application/yang-data+xml

HTTP/1.1 200 OK
<modules-state xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library"
               xmlns:yanglib="urn:ietf:params:xml:ns:yang:ietf-yang-library">
  <module-set-id>f4709e88d3250bd84f2378185c2833c2</module-set-id>
  <module>
    <name>dhcp</name>
    <revision>2019-02-14</revision>
    <schema>http://localhost:8080/restconf/tailf/modules/dhcp/2019-02-14</schema>
    <namespace>http://yang-central.org/ns/example/dhcp</namespace>
    <conformance-type>implement</conformance-type>
  </module>

  ...rest of the output removed here...

</modules-state>
```

We can now retrieve the *dhcp* Yang module via the URL we got in the *schema* leaf of the reply. Note that the actual URL may point anywhere. The URL is configured by the *schemaServerUrl* setting in the `ncs.conf` file.

```
GET /restconf/tailf/modules/dhcp/2019-02-14

HTTP/1.1 200 OK
module dhcp {
  namespace "http://yang-central.org/ns/example/dhcp";
  prefix dhcp;

  import ietf-yang-types {
```

```
  ...the rest of the Yang module removed here...
```

YANG Patch Media Type

The NSO RESTCONF API also support the YANG Patch Media Type, as defined in RFC 8072.

A *YANG Patch* is an ordered list of edits that are applied to the target datastore by the RESTCONF server. A YANG Patch request is sent as a HTTP PATCH request containing a body describing the edit operations to be performed. The format of the body is defined in the RFC 8072.

Refering to the dhcp Yang model in our Getting Started chapter; we will show how to use YANG Patch to achieve the same result but with fewer amount of requests.

Create two new resources with YANG Patch

In order to create the resources, we send a HTTP PATCH request where the *Content-Type* indicates that the body in the request consists of a Yang-Patch message. Our Yang-Patch request will initiate two edit operations where each operation will *create* a new subnet. In contrast, compare this with using "plain" RESTCONF where we would have needed two POST requests to achieve the same result.

Example 35. Create a two new *dhcp/subnet* resources

```
PATCH /restconf/data/dhcp:dhcp
Accept: application/yang-data+xml
Content-Type: application/yang-patch+xml

<yang-patch xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-patch">
  <patch-id>add-subnets</patch-id>
  <edit>
    <edit-id>add-subnet-239</edit-id>
    <operation>create</operation>
    <target>/subnet=10.254.239.0%2F27</target>
    <value>
      <subnet xmlns="http://yang-central.org/ns/example/dhcp" \
              xmlns:dhcp="http://yang-central.org/ns/example/dhcp">
        <net>10.254.239.0/27</net>
          ...content removed here for brevity...
        <max-lease-time>1200</max-lease-time>
      </subnet>
    </value>
  </edit>
  <edit>
    <edit-id>add-subnet-244</edit-id>
    <operation>create</operation>
    <target>/subnet=10.254.244.0%2F27</target>
    <value>
      <subnet xmlns="http://yang-central.org/ns/example/dhcp" \
              xmlns:dhcp="http://yang-central.org/ns/example/dhcp">
        <net>10.254.244.0/27</net>
          ...content removed here for brevity...
        <max-lease-time>1200</max-lease-time>
      </subnet>
    </value>
  </edit>
</yang-patch>

# If the YANG Patch request was successful,
# the server might respond as follows:
```

```
HTTP/1.1 200 OK
<yang-patch-status xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-patch">
  <patch-id>add-subnets</patch-id>
  <ok/>
</yang-patch-status>
```

Modify and Delete in the same Yang-Patch request

Let us modify the 'max-lease-time' of one 'subnet' and delete the 'max-lease-time' value of the second 'subnet'. Note that the delete will cause the default value of 'max-lease-time' to take effect, which we will verify using a RESTCONF GET request.

Example 36. Modify and Delete in the same Yang-Patch request

```
PATCH /restconf/data/dhcp:dhcp
Accept: application/yang-data+xml
Content-Type: application/yang-patch+xml

<yang-patch xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-patch">
  <patch-id>modify-and-delete</patch-id>
  <edit>
    <edit-id>modify-max-lease-time-239</edit-id>
    <operation>merge</operation>
    <target>/dhcp:subnet=10.254.239.0%2F27</target>
    <value>
      <subnet xmlns="http://yang-central.org/ns/example/dhcp" \
              xmlns:dhcp="http://yang-central.org/ns/example/dhcp">
        <net>10.254.239.0/27</net>
        <max-lease-time>1234</max-lease-time>
      </subnet>
    </value>
  </edit>
  <edit>
    <edit-id>delete-max-lease-time-244</edit-id>
    <operation>delete</operation>
    <target>/dhcp:subnet=10.254.244.0%2F27/max-lease-time</target>
  </edit>
</yang-patch>

# If the YANG Patch request was successful,
# the server might respond as follows:

HTTP/1.1 200 OK
<yang-patch-status xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-patch">
  <patch-id>modify-and-delete</patch-id>
  <ok/>
</yang-patch-status>
```

To verify that our modify and delete operations took place we make use of two RESTCONF GET request as shown below.

Example 37. Verify the modified max-release-time value

```
GET /restconf/data/dhcp:dhcp/subnet=10.254.239.0%2F27/max-lease-time
Accept: application/yang-data+xml

HTTP/1.1 200 OK
<max-lease-time xmlns="http://yang-central.org/ns/example/dhcp"
                xmlns:dhcp="http://yang-central.org/ns/example/dhcp">
                1234
```

```
</max-lease-time>
```

Example 38. Verify the default values after delete of the max-release-time value

```
GET /restconf/data/dhcp:dhcp/subnet=10.254.244.0%2F27/max-lease-time?\
      with-defaults=report-all-tagged
Accept: application/yang-data+xml

HTTP/1.1 200 OK
<max-lease-time wd:default="true"
                xmlns:wd="urn:ietf:params:restconf:capability:defaults:1.0"
                xmlns="http://yang-central.org/ns/example/dhcp"
                xmlns:dhcp="http://yang-central.org/ns/example/dhcp">
                7200
</max-lease-time>
```

Note how we in the last GET request make use of the *with-defaults* query parameter to request that a default value should be returned and also be tagged as such.

NMDA

Network Management Datastore Architecture (NMDA), as defined in RFC 8527, extends the RESTCONF protocol. This enable RESTCONF clients to discover which datastores are supported by the RESTCONF server, determine which modules are supported in each datastore, and interact with all the datastores supported by the NMDA.

A RESTCONF client can test if a server supports the NMDA by using either the HEAD or GET methods on */restconf/ds/ietf- datastores:operational*, as shown below:

Example 39. Check if the RESTCONF server support NMDA

```
HEAD /restconf/ds/ietf-datastores:operational

HTTP/1.1 200 OK
```

A RESTCONF client can discover which datastores and YANG modules the server supports by reading the YANG library information from the operational state datastore. Note in the example below that, since the result consists of three top-nodes, it can't be represented in XML; hence we request the returned content to be in JSON format. See also: the section called “Collections”.

Example 40. Check what datastores the RESTCONF server support

```
GET /restconf/ds/ietf-datastores:operational/datastore
Accept: application/yang-data+json

HTTP/1.1 200 OK
{
  "ietf-yang-library:datastore": [
    {
      "name": "ietf-datastores:running",
      "schema": "common"
    },
    {
      "name": "ietf-datastores:intended",
      "schema": "common"
    },
    {
      "name": "ietf-datastores:operational",
      "schema": "common"
    }
  ]
}
```

Extensions

To avoid any potential future conflict with the RESTCONF standard, any extensions made to the NSO implementation of RESTCONF is located under the URL path: */restconf/tailf*, or is controlled by means of a vendor specific media type.

Note There is no index of extensions under */restconf/tailf*. To list extensions, access */restconf/data/ietf-yang-library:modules-state* and follow published links for schemas.

Collections

The RESTCONF specification states that a result containing multiple instances (e.g a number of list entries) is not allowed if XML encoding is used. The reason for this is that an XML document can only have one root node.

This functionality is supported if the *http://tail-f.com/ns/restconf/collection/1.0* capability is presented. See also: the section called "How to view the capabilities of the RESTCONF server".

To remedy this, a HTTP GET request can make use of the *Accept:* media type: *application/vnd.yang.collection+xml* as shown in the following example. The result will then be wrapped within a *collection* element.

Example 41. Use of collections

```
GET /restconf/ds/ietf-datastores:operational/\
    ietf-yang-library:yang-library/datastore
Accept: application/vnd.yang.collection+xml

<collection xmlns="http://tail-f.com/ns/restconf/collection/1.0">
  <datastore xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library"
            xmlns:yanglib="urn:ietf:params:xml:ns:yang:ietf-yang-library">
    <name xmlns:ds="urn:ietf:params:xml:ns:yang:ietf-datastores">
       ds:running
    </name>
    <schema>common</schema>
  </datastore>
  <datastore xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library"
             xmlns:yanglib="urn:ietf:params:xml:ns:yang:ietf-yang-library">
    <name xmlns:ds="urn:ietf:params:xml:ns:yang:ietf-datastores">
      ds:intended
    </name>
    <schema>common</schema>
  </datastore>
  <datastore xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library
             xmlns:yanglib="urn:ietf:params:xml:ns:yang:ietf-yang-library">
    <name xmlns:ds="urn:ietf:params:xml:ns:yang:ietf-datastores">
      ds:operational
    </name>
    <schema>common</schema>
  </datastore>
</collection>
```

The RESTCONF Query API

The NSO RESTCONF Query API consists of a number of operation to start a query which may live over several RESTCONF request, where data can be fetch in suitable chunks. The data to be returned is produced by applying an XPath expression where the data also may be sorted.

The RESTCONF client can check if the NSO RESTCONF server support this functionality by looking for the *http://tail-f.com/ns/restconf/query-api/1.0* capability. See also: the section called "How to view the capabilities of the RESTCONF server".

The `tailf-rest-query.yang` and the `tailf-common-query.yang` YANG models describe the structure of the RESTCONF Query API messages. By using the Schema Resource functionality, as described in the section called "Schema resource" , you can get hold of them.

Request and Replies

The API consists of the following Requests:

- *start-query* : Start a query and return a query handle.
- *fetch-query-result* : Use a query handle to repeatedly fetch chunks of the result.
- *immediate-query* : Start a query and return the entire result immediately.
- *reset-query* : (Re)set where the next fetched result will begin from.
- *stop-query* : Stop (and close) the query.

The API consists of the following Replies:

- *start-query-result* : Reply to the start-query request.
- *query-result* : Reply to the fetch-query-result and immediate-query requests.

In the following examples, we'll use this data model:

Example 42. *example.yang* : model for the Query API example

```
container x {
  list host {
    key number;
    leaf number {
      type int32;
    }
    leaf enabled {
      type boolean;
    }
    leaf name {
      type string;
    }
    leaf address {
      type inet:ip-address;
    }
  }
}]
```

The actual format of the payload should be represented either in XML or JSON. Note how we indicate the type of content using the *Content-Type* HTTP header. For XML it could look like this:

Example 43. Example of a *start-query* request

```
POST /restconf/tailf/query
```

```
Content-Type: application/yang-data+xml

<start-query xmlns="http://tail-f.com/ns/tailf-rest-query">
  <foreach>
    /x/host[enabled = 'true']
  </foreach>
  <select>
    <label>Host name</label>
    <expression>name</expression>
    <result-type>string</result-type>
  </select>
  <select>
    <expression>address</expression>
    <result-type>string</result-type>
  </select>
  <sort-by>name</sort-by>
  <limit>100</limit>
  <offset>1</offset>
  <timeout>600</timeout>
</start-query>]
```

The same request in JSON format would look like:

Example 44. JSON example of a *start-query* request

```
POST /restconf/tailf/query
Content-Type: application/yang-data+json

{
 "start-query": {
   "foreach": "/x/host[enabled = 'true']",
   "select": [
     {
       "label": "Host name",
       "expression": "name",
       "result-type": ["string"]
     },
     {
       "expression": "address",
       "result-type": ["string"]
     }
   ],
   "sort-by": ["name"],
   "limit": 100,
   "offset": 1,
   "timeout": 600
 }
}]
```

An informal interpretation of this query is:

For each */x/host* where *enabled* is true, select its *name*, and *address*, and return the result sorted by *name*, in chunks of 100 result items at a time.

Let us discuss the various pieces of this request. To start with, when using XML, we need to specify the name space as shown:

```
<start-query xmlns="http://tail-f.com/ns/tailf-rest-query">
```

The actual XPath query to run is specified by the *foreach* element. In the example below will search for all */x/host* nodes that has the *enabled* node set to true:

```
<foreach>
  /x/host[enabled = 'true']
</foreach>
```

Now we need to define what we want to have returned from the node set by using one or more *select* sections. What to actually return is defined by the XPath *expression*.

Choose how the result should be represented. Basically, it can be the actual value or the path leading to the value. This is specified per select chunk. The possible result-types are: *string* , *path* , *leaf-value* and *inline*.

The difference between *string* and *leaf-value* is somewhat subtle. In the case of *string* the result will be processed by the XPath function: *string()* (which if the result is a node-set will concatenate all the values). The *leaf-value* will return the value of the first node in the result. As long as the result is a leaf node, *string* and *leaf-value* will return the same result. In the example above, the *string* is used as shown below. Note that at least one *result-type* must be specified.

The result-type *inline* makes it possible to return the full sub-tree of data, either in XML or in JSON format. The data will be enclosed with a tag: *data*.

It is possible to specify an optional *label* for a convenient way of labeling the returned data:

```
<select>
  <label>Host name</label>
  <expression>name</expression>
  <result-type>string</result-type>
</select>
<select>
  <expression>address</expression>
  <result-type>string</result-type>
</select>
```

The returned result can be sorted. This is expressed as an XPath expression, which in most cases is very simple and refers to the found node set. In this example we sort the result by the content of the *name* node:

```
<sort-by>name</sort-by>
```

With the *offset* element we can specify at which node we should start to receive the result. The default is 1, i.e., the first node in the resulting node-set.

```
<offset>1</offset>
```

It is possible to set a custom timeout when starting or resetting a query. Each time a function is called, the timeout timer resets. The default is 600 seconds, i.e. 10 minutes.

```
<timeout>600</timeout>
```

The reply to this request would look something like this:

```
<start-query-result>
  <query-handle>12345</query-handle>
</start-query-result>
```

The query handle (in this example '12345') must be used in all subsequent calls. To retrieve the result, we can now send:

```
<fetch-query-result xmlns="http://tail-f.com/ns/tailf-rest-query">
  <query-handle>12345</query-handle>
</fetch-query-result>
```

Which will result in something like the following:

```
<query-result xmlns="http://tail-f.com/ns/tailf-rest-query">
```

```
  <result>
    <select>
      <label>Host name</label>
      <value>One</value>
    </select>
    <select>
      <value>10.0.0.1</value>
    </select>
  </result>
  <result>
    <select>
      <label>Host name</label>
      <value>Three</value>
    </select>
    <select>
      <value>10.0.0.3</value>
    </select>
  </result>
</query-result>
```

If we try to get more data with the *fetch-query-result* we might get more *result* entries in return until no more data exists and we get an empty query result back:

```
<query-result xmlns="http://tail-f.com/ns/tailf-rest-query">
</query-result>
```

Finally, when we are done we stop the query:

```
<stop-query xmlns="http://tail-f.com/ns/tailf-rest-query">
  <query-handle>12345</query-handle>
</stop-query>
```

Reset a Query

If we want to go back in the "stream" of received data chunks and have them repeated, we can do that with the 'reset-query' request. In the example below we ask to get results from the 42:nd result entry:

```
<reset-query xmlns="http://tail-f.com/ns/tailf-rest-query">
  <query-handle>12345</query-handle>
  <offset>42</offset>
</reset-query>
```

Immediate Query

If we want to get the entire result sent back to us, using only one request, we can do this by using the *immediate-query*. This function takes similar arguments as *start-query* and returns the entire result analogous with the result from a *fetch-query-result* request. Note that it is not possible to paginate or set an offset start node for the result list; i.e. the options *limit* and *offset* are ignored.

Partial Responses

This functionality is supported if the *http://tail-f.com/ns/restconf/partial-response/1.0* capability is presented. See also: the section called “How to view the capabilities of the RESTCONF server”.

By default, the server sends back the full representation of a resource after processing a request. For better performance, the server can be instructed to send only the nodes the client really needs in a partial response.

To request a partial response for a set of list entries, use the *offset* and *limit* query parameters to specify a limited set of entries to be returned.

In the following example we retrieve only 2 entries, skipping the first entry and then returning the next two entries:

Example 45. Partial Response

```
GET /restconf/data/example-jukebox:jukebox/library/artist?offset=1&limit=2
Accept: application/yang-data+json

...in return we will get the second and third elements of the list...
```

Hidden Nodes

This functionality is supported if the *http://tail-f.com/ns/restconf/unhide/1.0* capability is presented. See also: the section called "How to view the capabilities of the RESTCONF server".

By default, hidden nodes are not visible in the RESTCONF interface. In order to unhide hidden nodes for retrieval or editing, clients can use the query parameter *unhide* or set parameter *showHidden* to *true* under `/confdConfig/restconf` in confd.conf file. The query parameter *unhide* is supported for RPC and action invocation.

The format of the *unhide* parameter is a comma separated list of

```
<groupname>[;<password>]
```

As an example:

```
unhide=extra,debug;secret
```

This example unhides the unprotected group *extra* and the password protected group *debug* with the password *secret*;.

Configuration Meta-Data

It is possible to associate meta-data with the configuration data. For RESTCONF, resources such as containers, lists as well as leafs and leaf-lists can have such meta-data. For XML, this meta-data is represented as attributes attached to the XML element in question. For JSON, there does not exist a natural way to represent this info. Hence a special special notation has been introduced, based on the RFC 7952, see the example below.

Example 46. XML representation of meta-data

```
<x xmlns="urn:x" xmlns:x="urn:x">
  <id tags=" important ethernet " annotation="hello world">42</id>
  <person annotation="This is a person">
    <name>Bill</name>
    <person annotation="This is another person">grandma</person>
  </person>
</x>
```

Example 47. JSON representation of meta-data

```
{
  "x": {
    "id": 42,
    "@id": {"tags": ["important","ethernet"],"annotation": "hello world"},
    "person": {
      // NB: the below refers to the parent object
      "@@person": {"annotation": "This is a person"},
      "name": "Bill",
      "person": "grandma",
      // NB: the below refers to the sibling object
      "@person": {"annotation": "This is another person"}
    }
  }
}
```

For JSON, note how we represent the meta data for a certain object "x" by another object constructed of the object name prefixed with either one or two "@" signs. The meta-data object "@x" refers to the sibling object "x" and the "@@x" object refers to the parent object.

Note *This differs from the RFC 7952!*

The Authentication Cache

The RESTCONF server maintains an authentication cache. When authenticating an incoming request for a particular *User:Password*, it is first checked if the User exists in the cache and if so, the request is processed. This makes it possible to avoid the, potentially time consuming, login procedure that will take place in case of a cache miss.

Cache entries has a maximum Time-To-Live (TTL) and upon expiry a cache entry is removed which will cause the next request for that User to perform the normal login procedure. The TTL value is configurable via the *auth-cache-ttl* parameter, as shown in the example. Note that, by setting the TTL value to *PT0S* (zero), the cache is effectively turned off.

It is also possible to combine the Clients IP address with the User name as a key into the cache. This behaviour is disabled by default. It can be enabled by setting the *enable-auth-cache-client-ip* parameter to *true*. With this enabled, only a Client coming from the same IP address may get a hit in the authentication cache.

Example 48. NSO configuration of the authentication cache TTL

```
...
<aaa>
   ...
   <restconf>
      <!-- Set the TTL to 10 seconds! -->
      <auth-cache-ttl>PT10S</auth-cache-ttl>
      <!-- Use both "User" and "ClientIP" as key into the AuthCache -->
      <enable-auth-cache-client-ip>false</enable-auth-cache-client-ip>
   </restconf>
   ...
</aaa>
...
```

Client IP via Proxy

It is possible to configure the NSO RESTCONF server to pick up the client IP address via a HTTP header in the request. A list of HTTP headers to look for is configurable via the *proxy-headers* parameter as shown in the example.

To avoid misuse of this feature, only requests from trusted sources will be searched for such a HTTP header. The list of trusted sources is configured via the *allowed-proxy-ip-prefix* as shown in the example.

Example 49. NSO configuration of Client IP via Proxy

```
...
<webui>
   ...
  <use-forwarded-client-ip>
    <proxy-headers>X-Forwarded-For</proxy-headers>
    <proxy-headers>X-REAL-IP</proxy-headers>
    <allowed-proxy-ip-prefix>10.12.34.0/24</allowed-proxy-ip-prefix>
    <allowed-proxy-ip-prefix>2001:db8:1234::/48</allowed-proxy-ip-prefix>
  </use-forwarded-client-ip>
   ...
</webui>
...
```

External token authentication/validation

The NSO RESTCONF server can be setup to pass a long a *token* used for authentication and/or validation of the client. Note that this require *external authentication/validation* to be setup properly. See the section called “External token validation” in *Administration Guide* and the section called “External authentication” in *Administration Guide* for details.

With *token authentication* we mean that the client sends a *User:Password* to the RESTCONF server, which will invoke an external executable that perform the authentication and upon success produces a *token* that the RESTCONF server will return in the *X-Auth-Token* HTTP header of the reply.

With *token validation* we mean that the RESTCONF server will pass along any token, provided in the *X-Auth-Token* HTTP header, to an external executable that performs the validation. This external program may produce a new token that the RESTCONF server will return in the *X-Auth-Token* HTTP header of the reply.

To make this work, the following need to be configured in the `ncs.conf` file:

Example 50. Configure RESTCONF external token authentication/validation

```
...
<restconf>
   ...
  <token-response>
    <x-auth-token>true</x-auth-token>
  </token-response>
   ...
</restconf>
...
```

It is also possible to have the RESTCONF server to return a HTTP *cookie* containing the token.

An HTTP cookie (web cookie, browser cookie) is a small piece of data that a server sends to the user's web browser. The browser may store it and send it back with the next request to the same server. This can be convenient in certain solutions, where typically, it is used to tell if two requests came from the same browser, keeping a user logged-in, for example.

To make this happen, the name of the cookie need to be configured as well as a *directives* string which will be sent as part of the cookie.

Example 51. Configure the RESTCONF token cookie

```
...
<restconf>
```

```
    ...
    <token-cookie>
      <name>X-JWT-ACCESS-TOKEN</name>
      <directives>path=/; Expires=Tue, 19 Jan 2038 03:14:07 GMT;</directives>
    </token-cookie>
    ...
</restconf>
...
```

Custom Response HTTP Headers

The RESTCONF server can be configured to reply with particular HTTP headers in the HTTP response. For example, to support Cross-Origin Resource Sharing (CORS, https://www.w3.org/TR/cors/) there is a need to add a couple of headers to the HTTP Response.

We add the extra configuration parameter in `ncs.conf`.

Example 52. NSO RESTCONF custom header configuration

```
<restconf>
  <enabled>true</enabled>
  <custom-headers>
    <header>
      <name>Access-Control-Allow-Origin</name>
      <value>*</value>
    </header>
  </custom-headers>
</restconf>
```

A number of HTTP header has been deemed so important by security reasons that they, with sensible default values, per default will be included in the RESTCONF reply. The values can be changed by configuration in the `ncs.conf` file. Note that a configured empty value will effectively turn off that particular header from being included in the RESTCONF reply. The headers and their default values are:

- *xFrameOptions* : *DENY*

  The default value indicate that the page cannot be displayed in a frame/iframe/embed/object regardless of the site attempting to do so.
- *xContentTypeOptions* : *nosniff*

  The default value indicate that the MIME types advertised in the Content-Type headers should not be changed and be followed. In particular should requests for CSS or Javascript be blocked in case a proper MIME type is not used.
- *xXssProtection* : *1; mode=block*

  This header is a feature of Internet Explorer, Chrome and Safari that stops pages from loading when they detect reflected cross-site scripting (XSS) attacks. It enables XSS filtering and tell the browser to prevent rendering of the page if an attack is detected.
- *strictTransportSecurity* : *max-age=15552000; includeSubDomains*

  The default value tell browsers that the RESTCONF server should only be accessed using HTTPS, instead of using HTTP. It set the time that the browser should remember this and state that this rule applies to all of the servers subdomains as well.
- *contentSecurityPolicy* : *default-src 'self'; block-all-mixed-content; base-uri 'self'; frame-ancestors 'none';*

  The default value means that: Resources like fonts, scripts, connections, images, and styles will all only load from the same origin as the protected resource. All mixed contents will be blocked and frame-ancestors like iframes and applets is prohibited.

Generating Swagger for RESTCONF

Swagger is a documentation language used to describe RESTful APIs. The resulting specifications are used to both document APIs as well as generating clients in a variety of languages. For more information about the Swagger specification itself and the ecosystem of tools available for it, see swagger.io.

The RESTCONF API in NSO provides an HTTP-based interface for accessing data. The YANG modules loaded into the system define the schema for the data structures that can be manipulated using the RESTCONF protocol. The **yanger** tool provides options to generate Swagger specifications from YANG files. The tool currently supports generating specifications according to OpenAPI/Swagger 2.0 using JSON encoding. The tool supports validation of JSON bodies in body parameters and response bodies, and XML content validation is not supported.

YANG and Swagger are two different languages serving slightly different purposes. YANG is a data modeling language used to model configuration data, state data, Remote Procedure Calls, and notifications for network management protocols such as NETCONF and RESTCONF. Swagger is an API definition language that documents API resource structure as well as HTTP body content validation for applicable HTTP request methods. Translation from YANG to Swagger is not perfect in the sense that there are certain constructs and features in YANG that is not possible to capture completely in Swagger. The design of the translation is designed such that the resulting Swagger definitions are *more* restrictive than what is expressed in the YANG definitions. This means that there are certain cases where a client can do more in the RESTCONF API than what the Swagger definition expresses. There is also a set of well-known resources defined in the RESTCONF RFC 8040 that are not part of the generated Swagger specification, notably resources related to event streams.

Using yanger to generate Swagger

The **yanger** tool is a YANG parser and validator that provides options to convert YANG modules to a multitude of formats including Swagger. You use the **-f swagger** option to generate a Swagger definition from one or more YANG files. The following command generates a Swagger file named `example.json` from the `example.yang` YANG file:

```
yanger -t expand -f swagger example.yang -o example.json
```

It is only supported to generate Swagger from one YANG module at a time. It is possible however to augment this module by supplying additional modules. The following command generates a Swagger document from `base.yang` which is augmented by `base-ext-1.yang` and `base-ext-2.yang`:

```
yanger -t expand -f swagger base.yang base-ext-1.yang base-ext-2.yang -o base.json
```

Only supplying augmenting modules is not supported.

Use the **--help** option to the **yanger** command to see all available options:

```
yanger --help
```

The complete list of options related to Swagger generation is:

```
Swagger output specific options:
  --swagger-host                    Add host to the Swagger output
  --swagger-basepath                Add basePath to the Swagger output
  --swagger-version                 Add version url to the Swagger output.
                                    NOTE: this will override any revision
                                    in the yang file
  --swagger-tag-mode                Set tag mode to group resources. Valid
                                    values are: methods, resources, all
                                    [default: all]
  --swagger-terms                   Add termsOfService to the Swagger
                                    output
  --swagger-contact-name            Add contact name to the Swagger output
  --swagger-contact-url             Add contact url to the Swagger output
  --swagger-contact-email           Add contact email to the Swagger output
  --swagger-license-name            Add license name to the Swagger output
  --swagger-license-url             Add license url to the Swagger output
  --swagger-top-resource            Generate only swagger resources from
                                    this top resource. Valid values are:
                                    root, data, operations, all [default:
                                    all]
  --swagger-omit-query-params       Omit RESTCONF query parameters
                                    [default: false]
  --swagger-omit-body-params        Omit RESTCONF body parameters
                                    [default: false]
  --swagger-omit-form-params        Omit RESTCONF form parameters
                                    [default: false]
  --swagger-omit-header-params      Omit RESTCONF header parameters
                                    [default: false]
  --swagger-omit-path-params        Omit RESTCONF path parameters
                                    [default: false]
  --swagger-omit-standard-statuses  Omit standard HTTP response statuses.
                                    NOTE: at least one successful HTTP
                                    status will still be included
                                    [default: false]
  --swagger-methods                 HTTP methods to include. Example:
                                    --swagger-methods "get, post"
                                    [default: "get, post, put, patch,
                                    delete"]
  --swagger-path-filter             Filter out paths matching a path filter.
                                    Example: --swagger-path-filter
                                    "/data/example-jukebox/jukebox"
```

Using the `example-jukebox.yang` from the RESTCONF RFC 8040, the following example generates a comprehensive Swagger definition using a variety of the Swagger-related options:

Example 53. Comprehensive Swagger generation example

```
yanger -p . -t expand -f swagger example-jukebox.yang \
       --swagger-host 127.0.0.1:8080 \
       --swagger-basepath /restconf \
       --swagger-version "My swagger version 1.0.0.1" \
       --swagger-tag-mode all \
       --swagger-terms "http://my-terms.example.com" \
       --swagger-contact-name "my contact name" \
       --swagger-contact-url "http://my-contact-url.example.com" \
       --swagger-contact-email "my-contact-email@example.com" \
       --swagger-license-name "my license name" \
       --swagger-license-url "http://my-license-url.example.com" \
       --swagger-top-resource all \
       --swagger-omit-query-params false \
       --swagger-omit-body-params false \
       --swagger-omit-form-params false \
       --swagger-omit-header-params false \
       --swagger-omit-path-params false \
       --swagger-omit-standard-statuses false \
       --swagger-methods "post, get, patch, put, delete, head, options"
```

CHAPTER 4

The NSO SNMP Agent

Introduction

The SNMP agent in NSO is used mainly for monitoring and notifications. It supports SNMPv1, SNMPv2c, and SNMPv3.

The following standard MIBs are supported by the SNMP agent:

- SNMPv2-MIB RFC 3418
- SNMP-FRAMEWORK-MIB RFC 3411
- SNMP-USER-BASED-SM-MIB RFC 3414
- SNMP-VIEW-BASED-ACM-MIB RFC 3415
- SNMP-COMMUNITY-MIB RFC 3584
- SNMP-TARGET-MIB and SNMP-NOTIFICATION-MIB RFC 3413
- SNMP-MPD-MIB RFC 3412
- TRANSPORT-ADDRESS-MIB RFC 3419
- SNMP-USM-AES-MIB RFC 3826
- IPV6-TC RFC 2465

Note The usmHMACMD5AuthProtocol authentication protocol and the usmDESPrivProtocol privacy protocol specified in SNMP-USER-BASED-SM-MIB are not supported, since they are not considered secure. The usmHMACSHAAuthProtocol authentication protocol specified in SNMP-USER-BASED-SM-MIB and the usmAesCfb128Protocol privacy protocol specified in SNMP-USM-AES-MIB are supported.

Configuring the SNMP Agent

The SNMP agent is configured through any of the normal NSO northbound interfaces. It is possible to control most aspects of the agent through for example the CLI.

The YANG models describing all configuration capabilities of the SNMP agent reside under `$NCS_DIR/src/ncs/snmp/snmp-agent-config/*.yang` in the NSO distribution.

An example session configuring the SNMP agent through the CLI may look like:

```
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# snmp agent udp-port 3457
admin@ncs(config)# snmp community public name foobaz
admin@ncs(config-community-public)# commit
Commit complete.
admin@ncs(config-community-public)# top
admin@ncs(config)# show full-configuration snmp
snmp agent enabled
snmp agent ip    0.0.0.0
snmp agent udp-port 3457
snmp agent version v1
snmp agent version v2c
snmp agent version v3
snmp agent engine-id enterprise-number 32473
snmp agent engine-id from-text testing
snmp agent max-message-size 50000
snmp system contact ""
snmp system name ""
snmp system location ""
snmp usm local user initial
 auth sha password GoTellMom
 priv aes password GoTellMom
!
snmp target monitor
 ip       127.0.0.1
 udp-port 162
 tag      [ monitor ]
 timeout  1500
 retries  3
 v2c sec-name public
!
snmp community public
 name     foobaz
 sec-name public
!
snmp notify foo
 tag  monitor
 type trap
!
snmp vacm group initial
 member initial
  sec-model [ usm ]
 !
 access usm no-auth-no-priv
  read-view   internet
  notify-view internet
 !
 access usm auth-no-priv
  read-view   internet
  notify-view internet
 !
 access usm auth-priv
  read-view   internet
  notify-view internet
 !
!
snmp vacm group public
 member public
  sec-model [ v1 v2c ]
 !
 access any no-auth-no-priv
  read-view   internet
  notify-view internet
 !
!
snmp vacm view internet
 subtree 1.3.6.1
  included
 !
!
snmp vacm view restricted
 subtree 1.3.6.1.6.3.11.2.1
  included
 !
 subtree 1.3.6.1.6.3.15.1.1
  included
 !
!
```

The SNMP agent configuration data is stored in CDB as any other configuration data, but is handled as a transformation between the data shown above and the data stored in the standard MIBs.

If you want to have a default configuration of the SNMP agent, you must provide that in an XML file. The initialization data of the SNMP agent is stored in an XML file that has precisely the same format as CDB initialization XML files, but it is not loaded by CDB, rather it is loaded at first startup by the SNMP agent. The XML file must be called `snmp_init.xml` and it must reside in the load path of NSO. In the NSO distribution, there is such an initialization file in `$NCS_DIR/etc/ncs/snmp/snmp_init.xml`. It is strongly recommended that this file is customized with another engine id and other community strings and v3 users.

If no `snmp_init.xml` file is found in the load path a default configuration with the agent disabled is loaded. Thus, the easiest way to start NSO without the SNMP agent is to ensure that the directory `$NCS_DIR/etc/ncs/snmp/` is not part of the NSO load path.

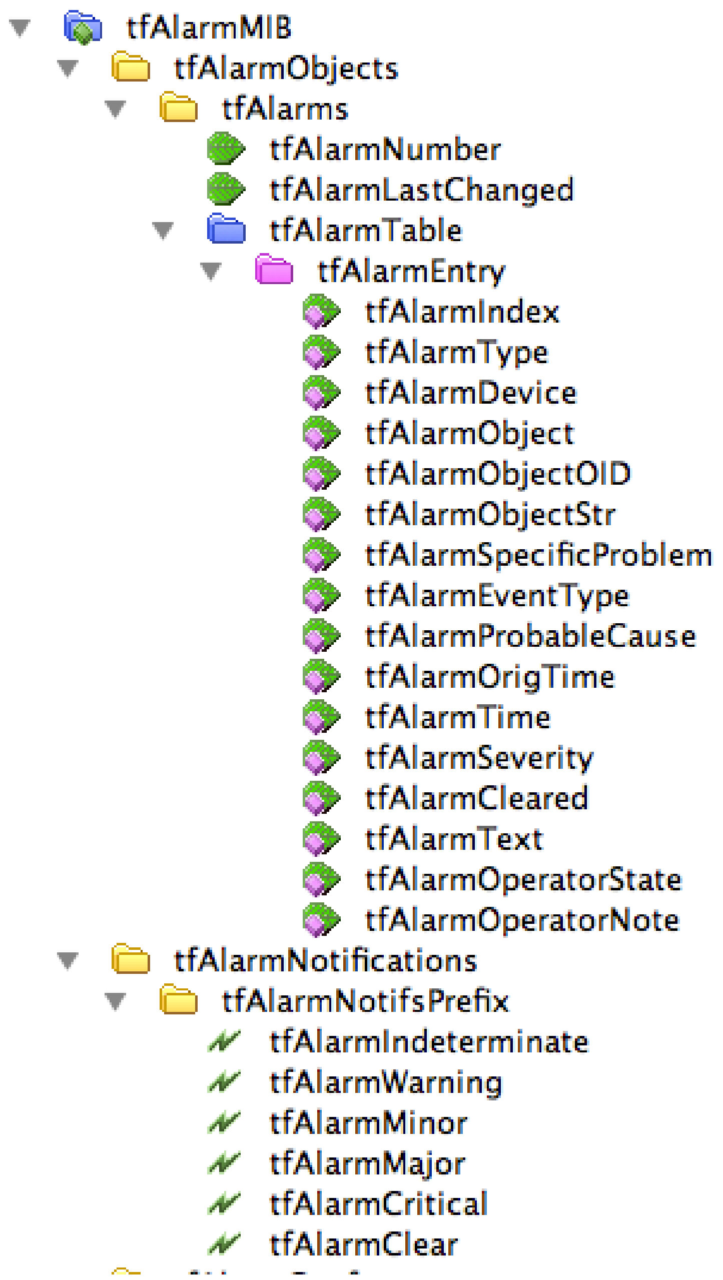
Note, this only relates to initialization the first time NSO is started. On subsequent starts, all the SNMP agent confiuration data is stored in CDB and the `snmp_init.xml` is never used again.

The Alarm MIB

The NSO SNMP alarm MIB is designed for ease of use in alarm systems. It defines a table of alarms and SNMP alarm notifications corresponding to alarm state changes. Based on the alarm model in NSO (see Chapter 5, *NSO Alarms*), the notifications as well as the alarm table contains the parameters that are required for alarm standards compliance (X.733 and 3GPP). The MIB files are located in `$NCS_DIR/src/ncs/snmp/mibs`.

TAILF-TOP-MIB.mib	the tail-f enterprise OID
TAILF-TC-MIB.mib	textual conventions for the alarm mib
TAILF-ALARM-MIB.mib	the actual alarm MIB
IANA-ITU-ALARM-TC-MIB.mib	import of IETF mapping of X.733 parameters
ITU-ALARM-TC-MIB.mib	import of IETF mapping of X.733 parameters

Figure 54. The NSO Alarm MIB



The alarm table has the following columns:

tfAlarmIndex	an imaginary index for the alarm row that is persistent between restarts
tfAlarmType	this provides an identification of the alarm type and together with tfAlarmSpecificProblem forms a unique identification of the alarm
tfAlarmDevice	the alarming network device - can be NSO itself
tfAlarmObject	the alarming object within the device
tfAlarmObjectOID	in case the original alarm notification was a SNMP notification this column identifies the alarming SNMP object

tfAlarmObjectStr	name of alarm object based on any other naming
tfAlarmSpecificProblem	this object is used when the 'tfAlarmType' object cannot uniquely identify the alarm type
tfAlarmEventType	the event type according to X.733 and based on the mapping of the alarm type in the NSO alarm model
tfAlarmProbableCause	the probable cause to X.733 and based on the mapping of the alarm type in the NSO alarm model. Note that you can configure this to match the probable cause values in the receiving alarm system
tfAlarmOrigTime	the time for the first occurrence of this alarm
tfAlarmTime	the time for the last state change of this alarm
tfAlarmSeverity	the latest severity (non clear) reported for this alarm
tfAlarmCleared	boolean indicated if the latest state change reports a clear
tfAlarmText	the latest alarm text.
tfAlarmOperatorState	the latest operator alarm state such as ack
tfAlarmOperatorNote	the latest operator note

The MIB defines separate notifications for every severity level in order to support SNMP managers that only can map severity levels to individual notifications. Every notification contains the parameters of the alarm table.

SNMP Object Identifiers

Example 55. Object Identifiers

```
tfAlarmMIB             node         1.3.6.1.4.1.24961.2.103
tfAlarmObjects         node         1.3.6.1.4.1.24961.2.103.1
tfAlarms               node         1.3.6.1.4.1.24961.2.103.1.1
tfAlarmNumber          scalar       1.3.6.1.4.1.24961.2.103.1.1.1
tfAlarmLastChanged     scalar       1.3.6.1.4.1.24961.2.103.1.1.2
tfAlarmTable           table        1.3.6.1.4.1.24961.2.103.1.1.5
tfAlarmEntry           row          1.3.6.1.4.1.24961.2.103.1.1.5.1
tfAlarmIndex           column       1.3.6.1.4.1.24961.2.103.1.1.5.1.1
tfAlarmType            column       1.3.6.1.4.1.24961.2.103.1.1.5.1.2
tfAlarmDevice          column       1.3.6.1.4.1.24961.2.103.1.1.5.1.3
tfAlarmObject          column       1.3.6.1.4.1.24961.2.103.1.1.5.1.4
tfAlarmObjectOID       column       1.3.6.1.4.1.24961.2.103.1.1.5.1.5
tfAlarmObjectStr       column       1.3.6.1.4.1.24961.2.103.1.1.5.1.6
tfAlarmSpecificProblem column       1.3.6.1.4.1.24961.2.103.1.1.5.1.7
tfAlarmEventType       column       1.3.6.1.4.1.24961.2.103.1.1.5.1.8
tfAlarmProbableCause   column       1.3.6.1.4.1.24961.2.103.1.1.5.1.9
tfAlarmOrigTime        column       1.3.6.1.4.1.24961.2.103.1.1.5.1.10
tfAlarmTime            column       1.3.6.1.4.1.24961.2.103.1.1.5.1.11
tfAlarmSeverity        column       1.3.6.1.4.1.24961.2.103.1.1.5.1.12
tfAlarmCleared         column       1.3.6.1.4.1.24961.2.103.1.1.5.1.13
tfAlarmText            column       1.3.6.1.4.1.24961.2.103.1.1.5.1.14
tfAlarmOperatorState   column       1.3.6.1.4.1.24961.2.103.1.1.5.1.15
tfAlarmOperatorNote    column       1.3.6.1.4.1.24961.2.103.1.1.5.1.16
tfAlarmNotifications   node         1.3.6.1.4.1.24961.2.103.2
tfAlarmNotifsPrefix    node         1.3.6.1.4.1.24961.2.103.2.0
tfAlarmNotifsObjects   node         1.3.6.1.4.1.24961.2.103.2.1
tfAlarmStateChangeText scalar       1.3.6.1.4.1.24961.2.103.2.1.1
tfAlarmIndeterminate   notification 1.3.6.1.4.1.24961.2.103.2.0.1
tfAlarmWarning         notification 1.3.6.1.4.1.24961.2.103.2.0.2
tfAlarmMinor           notification 1.3.6.1.4.1.24961.2.103.2.0.3
tfAlarmMajor           notification 1.3.6.1.4.1.24961.2.103.2.0.4
tfAlarmCritical        notification 1.3.6.1.4.1.24961.2.103.2.0.5
```

```
 tfAlarmClear           notification 1.3.6.1.4.1.24961.2.103.2.0.6
 tfAlarmConformance     node         1.3.6.1.4.1.24961.2.103.10
 tfAlarmCompliances     node         1.3.6.1.4.1.24961.2.103.10.1
 tfAlarmCompliance      compliance   1.3.6.1.4.1.24961.2.103.10.1.1
 tfAlarmGroups          node         1.3.6.1.4.1.24961.2.103.10.2
 tfAlarmNotifs          group        1.3.6.1.4.1.24961.2.103.10.2.1
 tfAlarmObjs            group        1.3.6.1.4.1.24961.2.103.10.2.2
```

Using the SNMP Alarm MIB

Alarm Managers should subscribe to the notifications and read the alarm table in order to synchronize the alarm list. In order to do this you need an access view that matches the alarm MIB and create a SNMP target. Default SNMP settings in NSO lets you read the alarm MIB with v2c and community public. A target is setup in the following way, (assuming the SNMP Alarm Manager has IP address 192.168.1.1 and wants community string public in the v2c notifications):

Example 56. Subscribing to SNMP Alarms

```
$ ncs_cli -u admin -C
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# snmp notify monitor type trap tag monitor
admin@ncs(config-notify-monitor)# snmp target alarm-system ip 192.168.1.1 udp-port 162 \
        tag monitor v2c sec-name public
admin@ncs(config-target-alarm-system)# commit
Commit complete.
admin@ncs(config-target-alarm-system)# show full-configuration snmp target
snmp target alarm-system
 ip       192.168.1.1
 udp-port 162
 tag      [ monitor ]
 timeout  1500
 retries  3
 v2c sec-name public
!
snmp target monitor
 ip       127.0.0.1
 udp-port 162
 tag      [ monitor ]
 timeout  1500
 retries  3
 v2c sec-name public
!
admin@ncs(config-target-alarm-system)#
```

CHAPTER 5

NSO Alarms

Overview

NSO generates alarms for serious problems that must be remedied. Alarms are available over all north-bound interfaces and the 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-snmp-alarm-northbound`.

Alarm type structure

```
alarm-type
    ha-alarm
        certificate-expiration
        ha-node-down-alarm
            ha-primary-down
            ha-secondary-down
    ncs-cluster-alarm
        cluster-subscriber-failure
    ncs-dev-manager-alarm
        abort-error
        bad-user-input
        commit-through-queue-blocked
        commit-through-queue-failed
        commit-through-queue-failed-transiently
        commit-through-queue-rollback-failed
        configuration-error
        connection-failure
        final-commit-error
        missing-transaction-id
        ned-live-tree-connection-failure
        out-of-sync
        revision-error
    ncs-package-alarm
        package-load-failure
        package-operation-failure
    ncs-service-manager-alarm
        service-activation-failure
    ncs-snmp-notification-receiver-alarm
        receiver-configuration-error
    time-violation-alarm
        transaction-lock-time-violation
```

Alarm type descriptions

Table 57. Alarm type descriptions (alphabetically)

Alarm Identity	**Initial Perceived Severity**
abort-error	major
Description	**Recommended Action**
An error happened while aborting or reverting a transaction. Device's configuration is likely to be inconsistent with the NCS CDB.	Inspect the configuration difference with compare-config, resolve conflicts with sync-from or sync-to if any.

Alarm message(s)

- `Device {dev} is locked`
- `Device {dev} is southbound locked`
- `abort error`

Clear condition(s)

If NCS achieves sync with the device, or receives a transaction id for a netconf session towards the device, the alarm is cleared.

Alarm Identity

alarm-type

Description

Base identity for alarm types. A unique identification of the fault, not including the managed object. Alarm types are used to identify if alarms indicate the same problem or not, for lookup into external alarm documentation, etc. Different managed object types and instances can share alarm types. If the same managed object reports the same alarm type, it is to be considered to be the same alarm. The alarm type is a simplification of the different X.733 and 3GPP alarm IRP alarm correlation mechanisms and it allows for hierarchical extensions. A 'specific-problem' can be used in addition to the alarm type in order to have different alarm types based on information not known at design-time, such as values in textual SNMP Notification varbinds.

Alarm Identity	**Initial Perceived Severity**
bad-user-input	critical
Description	**Recommended Action**
Invalid input from user. NCS cannot recognize parameters needed to connect to device.	Verify that the user supplied input are correct.

Alarm message(s)

- `Resource {resource} doesn't exist`

Clear condition(s)

This alarm is not cleared.

Alarm Identity

certificate-expiration

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

Alarm message(s)

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

Clear condition(s)

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

Alarm Identity	Initial Perceived Severity
cluster-subscriber-failure	critical

Description	Recommended Action
Failure to establish a notification subscription towards a remote node.	Verify IP connectivity between cluster nodes.

Alarm message(s)

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

Clear condition(s)

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

Alarm Identity	Initial Perceived Severity
commit-through-queue-blocked	warning

Description

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

Alarm message(s)

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

Clear condition(s)

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

Alarm Identity	Initial Perceived Severity
commit-through-queue-failed	critical

Description	Recommended Action
A queued commit failed.	Resolve with rollback if possible.

Alarm message(s)

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

Clear condition(s)

This alarm is not cleared.

Alarm Identity	**Initial Perceived Severity**
commit-through-queue-failed-transiently	critical
Description	**Recommended Action**
A queued commit failed as it exhausted its retry attempts on transient errors.	Resolve with rollback if possible.

Alarm message(s)

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

Clear condition(s)

This alarm is not cleared.

Alarm Identity	**Initial Perceived Severity**
commit-through-queue-rollback-failed	critical
Description	**Recommended Action**
Rollback of a commit-queue item failed.	Investigate the status of the device and resolve the situation by issuing the appropriate action, i.e., service redeploy or a sync operation.

Alarm message(s)

- `{Reason}`

Clear condition(s)

This alarm is not cleared.

Alarm Identity	Initial Perceived Severity
configuration-error	critical
Description	**Recommended Action**
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.

Alarm message(s)

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

Clear condition(s)

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

Alarm Identity	Initial Perceived Severity
connection-failure	major
Description	**Recommended Action**
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.

Alarm message(s)

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

Clear condition(s)

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

Alarm Identity	Initial Perceived Severity
final-commit-error	critical
Description	**Recommended Action**
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.

Alarm message(s)

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

Clear condition(s)

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

Alarm Identity
ha-alarm
Description
Base type for all alarms related to high availablity. This is never reported, sub-identities for the specific high availability alarms are used in the alarms.

Alarm Identity
ha-node-down-alarm
Description
Base type for all alarms related to nodes going down in high availablity. This is never reported, sub-identities for the specific node down alarms are used in the alarms.

Alarm Identity	Initial Perceived Severity
ha-primary-down	critical
Description	**Recommended Action**
The node lost the connection to the primary node.	Make sure the HA cluster is operational, investigate why the primary went down and bring it up again.

Alarm message(s)

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

Clear condition(s)

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

Alarm Identity	Initial Perceived Severity
ha-secondary-down	critical
Description	**Recommended Action**
The node lost the connection to a secondary node.	Investigate why the secondary node went down, fix the connectivity issue and reconnect the secondary to the HA cluster.

Alarm message(s)

- `Lost connection to secondary`

Clear condition(s)

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

Alarm Identity	Initial Perceived Severity
missing-transaction-id	warning
Description	**Recommended Action**
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	Verify NACM rules on the concerned device.

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

Alarm message(s)

- `{Reason}`

Clear condition(s)

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

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

Alarm Identity
ncs-dev-manager-alarm
Description
Base type for all alarms related to the device manager This is never reported, sub-identities for the specific device alarms are used in the alarms.

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

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

Alarm Identity
ncs-snmp-notification-receiver-alarm
Description
Base type for SNMP notification receiver Alarms. This is never reported, sub-identities for specific SNMP notification receiver alarms are used in the alarms.

Alarm Identity	**Initial Perceived Severity**
ned-live-tree-connection-failure	major
Description	**Recommended Action**
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.

Alarm message(s)

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

Clear condition(s)

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

Alarm Identity	Initial Perceived Severity
out-of-sync	major
Description	**Recommended Action**
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.

Alarm message(s)

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

Clear condition(s)

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

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

Alarm message(s)

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

Clear condition(s)

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

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

Clear condition(s)

This alarm is not cleared.

Alarm Identity	Initial Perceived Severity
receiver-configuration-error	major
Description	**Recommended Action**
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.

Alarm message(s)

- `Configuration has errors.`

Clear condition(s)

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

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

Alarm message(s)

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

Clear condition(s)

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

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

Alarm message(s)

- `Multiple device errors: {str}`

Clear condition(s)

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

Alarm Identity

time-violation-alarm

Description

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

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

Alarm message(s)

- `Transaction lock time exceeded threshold.`

Clear condition(s)

This alarm is cleared when the transaction has finished.