ncclient Documentation

Release 0.6.12

Shikhar Bhushan and Leonidas Poulopoulos

Mar 03, 2025

CONTENTS

ncclient is a Python library for NETCONF clients. It aims to offer an intuitive API that sensibly maps the XML-encoded nature of NETCONF to Python constructs and idioms, and make writing network-management scripts easier. Other key features are:

- Supports all operations and capabilities defined in **RFC 6241**.
- Request pipelining.
- Asynchronous RPC requests.
- Keeping XML out of the way unless really needed.
- Extensible. New transport mappings and capabilities/operations can be easily added.

The best way to introduce is through a simple code example:

```
from ncclient import manager

# use unencrypted keys from ssh-agent or ~/.ssh keys, and rely on known_hosts
with manager.connect_ssh("host", username="user") as m:
    assert(":url" in m.server_capabilities)
    with m.locked("running"):
        m.copy_config(source="running", target="file:///new_checkpoint.conf")
        m.copy_config(source="file:///old_checkpoint.conf", target="running")
```

As of version 0.4 there has been an integration of Juniper's and Cisco's forks. Thus, lots of new concepts have been introduced that ease management of Juniper and Cisco devices respectively. The biggest change is the introduction of device handlers in connection params. For example to invoke Juniper's functions and params one has to re-write the above with **device_params={'name':'junos'}**:

```
from ncclient import manager

with manager.connect(host=host, port=830, username=user, hostkey_verify=False, device_
↪params={'name':'junos'}) as m:
    c = m.get_config(source='running').data_xml
    with open("%s.xml" % host, 'w') as f:
        f.write(c)
```

Respectively, for Cisco Nexus, the name is **nexus**. Device handlers are easy to implement and prove to be futureproof.

The latest pull request merge includes support for Huawei devices with name **huawei** in device_params.

CHAPTER

ONE

SUPPORTED DEVICE HANDLERS

- Alcatel Lucent: *device_params={'name':'alu'}*
- Ciena: *device_params={'name':'ciena'}*
- **Cisco:**
  - CSR: *device_params={'name':'csr'}*
  - Nexus: *device_params={'name':'nexus'}*
  - IOS XR: *device_params={'name':'iosxr'}*
  - IOS XE: *device_params={'name':'iosxe'}*
- H3C: *device_params={'name':'h3c'}*
- HP Comware: *device_params={'name':'hpcomware'}*
- **Huawei:**
  - *device_params={'name':'huawei'}*
  - *device_params={'name':'huaweiyang'}*
- Juniper: *device_params={'name':'junos'}*
- Server or anything not in above: *device_params={'name':'default'}*

Contents:

1.1 `manager` – High-level API

This module is a thin layer of abstraction around the library. It exposes all core functionality.

1.1.1 Customizing

These attributes control what capabilties are exchanged with the NETCONF server and what operations are available through the `Manager` API.

ncclient.manager.**OPERATIONS = {'cancel_commit': <class 'ncclient.operations.edit.CancelCommit'>, 'close_session': <class 'ncclient.operations.session.CloseSession'>, 'commit': <class 'ncclient.operations.edit.Commit'>, 'copy_config': <class 'ncclient.operations.edit.CopyConfig'>, 'create_subscription': <class 'ncclient.operations.subscribe.CreateSubscription'>, 'delete_config': <class 'ncclient.operations.edit.DeleteConfig'>, 'discard_changes': <class 'ncclient.operations.edit.DiscardChanges'>, 'dispatch': <class 'ncclient.operations.retrieve.Dispatch'>, 'edit_config': <class 'ncclient.operations.edit.EditConfig'>, 'get': <class 'ncclient.operations.retrieve.Get'>, 'get_config': <class 'ncclient.operations.retrieve.GetConfig'>, 'get_schema': <class 'ncclient.operations.retrieve.GetSchema'>, 'kill_session': <class 'ncclient.operations.session.KillSession'>, 'lock': <class 'ncclient.operations.lock.Lock'>, 'poweroff_machine': <class 'ncclient.operations.flowmon.PoweroffMachine'>, 'reboot_machine': <class 'ncclient.operations.flowmon.RebootMachine'>, 'rpc': <class 'ncclient.operations.rpc.GenericRPC'>, 'unlock': <class 'ncclient.operations.lock.Unlock'>, 'validate': <class 'ncclient.operations.edit.Validate'>}**

Dictionary of base method names and corresponding *RPC* subclasses. It is used to lookup operations, e.g. *get_config* is mapped to *GetConfig*. It is thus possible to add additional operations to the *Manager* API.

1.1.2 Factory functions

A *Manager* instance is created using a factory function.

ncclient.manager.**connect_ssh**(**args*, ***kwds*)

Initialize a *Manager* over the SSH transport. For documentation of arguments see *ncclient.transport.SSHSession.connect()*.

The underlying *ncclient.transport.SSHSession* is created with `CAPABILITIES`. All the provided arguments are passed directly to its implementation of *connect()*.

To customize the *Manager*, add a *manager_params* dictionary in connection parameters (e.g. *manager_params={'timeout': 60}* for a bigger RPC timeout parameter)

To invoke advanced vendor related operation add *device_params={'name': '<vendor_alias>'}* in connection parameters. For the time, 'junos' and 'nexus' are supported for Juniper and Cisco Nexus respectively.

A custom device handler can be provided with *device_params={'handler':<handler class>}* in connection parameters.

To specify errors to be be ignored in the RPC reply add *errors_params={'ignore_errors': ['error pattern to ignore']}* in connection parameters.

To control the error raising mode add *raise_mode* in the *errors_params* (e.g. *errors_params={'raise_mode': 0}* for ignoring all RPC errors) See *ncclient.operations.rpc.RaiseMode* for valid values of *raise_mode*

ncclient.manager.**connect = <function connect>**

1.1.3 Manager

Exposes an API for RPC operations as method calls. The return type of these methods depends on whether we are in *asynchronous or synchronous mode*.

In synchronous mode replies are awaited and the corresponding *RPCReply* object is returned. Depending on the *exception raising mode*, an *rpc-error* in the reply may be raised as an *RPCError* exception.

However in asynchronous mode, operations return immediately with the corresponding *RPC* object. Error handling and checking for whether a reply has been received must be dealt with manually. See the *RPC* documentation for details.

Note that in case of the *get()* and *get_config()* operations, the reply is an instance of *GetReply* which exposes the additional attributes *data* (as `_Element`) and *data_xml* (as a string), which are of primary interest in case of these operations.

Presence of capabilities is verified to the extent possible, and you can expect a *MissingCapabilityError* if something is amiss. In case of transport-layer errors, e.g. unexpected session close, *TransportError* will be raised.

class ncclient.manager.**Manager**(*session*, *device_handler*, *timeout=30*, *raise_mode=2*)

For details on the expected behavior of the operations and their parameters refer to **RFC 6241**.

Manager instances are also context managers so you can use it like this:

```
with manager.connect("host") as m:
    # do your stuff
```

... or like this:

```
m = manager.connect("host")
try:
    # do your stuff
finally:
    m.close_session()
```

HUGE_TREE_DEFAULT = False

Default for *huge_tree* support for XML parsing of RPC replies (defaults to False)

get_config(*source*, *filter=None*, *with_defaults=None*)

get_config is mapped to *GetConfig*

get_schema(*identifier*, *version=None*, *format=None*)

get_schema is mapped to *GetSchema*

edit_config(*config*, *format='xml'*, *target='candidate'*, *default_operation=None*, *test_option=None*, *error_option=None*)

edit_config is mapped to *EditConfig*

copy_config(*source*, *target*)

copy_config is mapped to *CopyConfig*

delete_config(*target*)

delete_config is mapped to *DeleteConfig*

dispatch(*rpc_command*, *source=None*, *filter=None*)

dispatch is mapped to *Dispatch*

lock(*target='candidate'*)

lock is mapped to *Lock*

unlock(*target='candidate'*)

unlock is mapped to *Unlock*

get(*filter=None*, *with_defaults=None*)

get is mapped to *Get*

close_session()

close_session is mapped to *CloseSession*

kill_session(*session_id*)

kill_session is mapped to *KillSession*

commit(*confirmed=False*, *timeout=None*, *persist=None*, *persist_id=None*)

commit is mapped to *Commit*

cancel_commit(*persist_id=None*)

cancel_commit is mapped to *CancelCommit*

discard_changes()

discard_changes is mapped to *DiscardChanges*

validate(*source='candidate'*)

validate is mapped to *Validate*

create_subscription(*filter=None*, *stream_name=None*, *start_time=None*, *stop_time=None*)

create_subscription is mapped to *CreateSubscription*

reboot_machine()

reboot_machine is mapped to *RebootMachine*

poweroff_machine()

poweroff_machine is mapped to *PoweroffMachine*

locked(*target*)

Returns a context manager for a lock on a datastore, where *target* is the name of the configuration datastore to lock, e.g.:

```
with m.locked("running"):
    # do your stuff
```

... instead of:

```
m.lock("running")
try:
    # do your stuff
finally:
    m.unlock("running")
```

take_notification(*block=True*, *timeout=None*)

Attempt to retrieve one notification from the queue of received notifications.

If block is True, the call will wait until a notification is received.

If timeout is a number greater than 0, the call will wait that many seconds to receive a notification before timing out.

If there is no notification available when block is False or when the timeout has elapse, None will be returned.

Otherwise a `Notification` object will be returned.

async_mode

Specify whether operations are executed asynchronously (*True*) or synchronously (*False*) (the default).

timeout

Specify the timeout for synchronous RPC requests.

raise_mode

Specify which errors are raised as *RPCError* exceptions. Valid values are the constants defined in *RaiseMode*. The default value is *ALL*.

client_capabilities

Capabilities object representing the client's capabilities.

server_capabilities

Capabilities object representing the server's capabilities.

session_id

session-id assigned by the NETCONF server.

connected

Whether currently connected to the NETCONF server.

huge_tree

Whether *huge_tree* support for XML parsing of RPC replies is enabled (default=False) The default value is configurable through *HUGE_TREE_DEFAULT*

1.1.4 Special kinds of parameters

Some parameters can take on different types to keep the interface simple.

Source and target parameters

Where an method takes a *source* or *target* argument, usually a datastore name or URL is expected. The latter depends on the *:url* capability and on whether the specific URL scheme is supported. Either must be specified as a string. For example, *"running"*, *"ftp://user:pass@host/config"*.

If the source may be a *config* element, e.g. as allowed for the *validate* RPC, it can also be specified as an XML string or an `Element` object.

Filter parameters

Where a method takes a *filter* argument, it can take on the following types:

- A tuple of *(type, criteria)*.

  Here *type* has to be one of *"xpath"* or *"subtree"*.

  - For *"xpath"* the *criteria* should be a string containing the XPath expression or a tuple containing a dict of namespace mapping and the XPath expression.
  - For *"subtree"* the *criteria* should be an XML string or an `Element` object containing the criteria.

- A list of *spec*

  Here *type* has to be *"subtree"*.

  - the *spec* should be a list containing multiple XML string or multiple `Element` objects.

- A <*filter*> element as an XML string or an `Element` object.

1.2 Complete API documentation

1.2.1 capabilities – NETCONF Capabilities

ncclient.capabilities.**schemes**(*url_uri*)

Given a URI that has a *scheme* query string (i.e. *:url* capability URI), will return a list of supported schemes.

class ncclient.capabilities.**Capabilities**(*capabilities*)

Represents the set of capabilities available to a NETCONF client or server. It is initialized with a list of capability URI's.

Members

":cap" in caps

Check for the presence of capability. In addition to the URI, for capabilities of the form *urn:ietf:params:netconf:capability:$name:$version* their shorthand can be used as a key. For example, for *urn:ietf:params:netconf:capability:candidate:1.0* the shorthand would be *:candidate*. If version is significant, use *:candidate:1.0* as key.

iter(caps)

Return an iterator over the full URI's of capabilities represented by this object.

1.2.2 xml_ – XML handling

Methods for creating, parsing, and dealing with XML and ElementTree objects.

exception ncclient.xml_.**XMLError**

Bases: NCClientError

Namespaces

ncclient.xml_.**BASE_NS_1_0 = 'urn:ietf:params:xml:ns:netconf:base:1.0'**

Base NETCONF namespace

ncclient.xml_.**TAILF_AAA_1_1 = 'http://tail-f.com/ns/aaa/1.1'**

Namespace for Tail-f core data model

ncclient.xml_.**TAILF_EXECD_1_1 = 'http://tail-f.com/ns/execd/1.1'**

Namespace for Tail-f execd data model

ncclient.xml_.**CISCO_CPI_1_0 = 'http://www.cisco.com/cpi_10/schema'**

Namespace for Cisco data model

ncclient.xml_.**JUNIPER_1_1 = 'http://xml.juniper.net/xnm/1.1/xnm'**

Namespace for Juniper 9.6R4. Tested with Junos 9.6R4+

ncclient.xml_.**FLOWMON_1_0 = 'http://www.liberouter.org/ns/netopeer/flowmon/1.0'**

Namespace for Flowmon data model

ncclient.xml_.**register_namespace**(*prefix*, *uri*)

Registers a namespace prefix that newly created Elements in that namespace will use. The registry is global, and any existing mapping for either the given prefix or the namespace URI will be removed.

ncclient.xml_.**qualify**(*tag*, *ns='urn:ietf:params:xml:ns:netconf:base:1.0'*)

Qualify a *tag* name with a *namespace*, in ElementTree fashion i.e. *{namespace}tagname*.

Conversion

ncclient.xml_.**to_xml**(*ele*, *encoding='UTF-8'*, *pretty_print=False*)

Convert and return the XML for an *ele* (Element) with specified *encoding*.

ncclient.xml_.**to_ele**(*x*, *huge_tree=False*)

Convert and return the Element for the XML document *x*. If *x* is already an Element simply returns that.

huge_tree: parse XML with very deep trees and very long text content

ncclient.xml_.**parse_root**(*raw*)

Efficiently parses the root element of a *raw* XML document, returning a tuple of its qualified name and attribute dictionary.

ncclient.xml_.**validated_element**(*x*, *tags=None*, *attrs=None*)

Checks if the root element of an XML document or Element meets the supplied criteria.

tags if specified is either a single allowable tag name or sequence of allowable alternatives

attrs if specified is a sequence of required attributes, each of which may be a sequence of several allowable alternatives

Raises *XMLError* if the requirements are not met.

1.2.3 transport – Transport / Session layer

Base types

class ncclient.transport.**Session**(*capabilities*)

Base class for use by transport protocol implementations.

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

add_listener(*listener*)

Register a listener that will be notified of incoming messages and errors.

property client_capabilities

Client's Capabilities

property connected

Connection status of the session.

get_listener_instance(*cls*)

If a listener of the specified type is registered, returns the instance.

property id

A string representing the *session-id*. If the session has not been initialized it will be *None*

remove_listener(*listener*)

Unregister some listener; ignore if the listener was never registered.

property server_capabilities

Server's Capabilities

class ncclient.transport.**SessionListener**

Base class for *Session* listeners, which are notified when a new NETCONF message is received or an error occurs.

> **Note**
>
> Avoid time-intensive tasks in a callback's context.

callback(*root*, *raw*)

Called when a new XML document is received. The *root* argument allows the callback to determine whether it wants to further process the document.

Here, *root* is a tuple of *(tag, attributes)* where *tag* is the qualified name of the root element and *attributes* is a dictionary of its attributes (also qualified names).

raw will contain the XML document as a string.

errback(*ex*)

Called when an error occurs.

SSH session implementation

ssh.**default_unknown_host_cb**(*fingerprint*)

An unknown host callback returns *True* if it finds the key acceptable, and *False* if not.

This default callback always returns *False*, which would lead to connect() raising a SSHUnknownHost exception.

Supply another valid callback if you need to verify the host key programmatically.

host is the hostname that needs to be verified

fingerprint is a hex string representing the host key fingerprint, colon-delimited e.g. *"4b:69:6c:72:6f:79:20:77:61:73:20:68:65:72:65:21"*

class ncclient.transport.**SSHSession**(*device_handler*)

Bases: *Session*

Implements a **RFC 4742** NETCONF session over SSH.

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

connect(*host*[, *port=830*, *timeout=None*, *unknown_host_cb=default_unknown_host_cb*, *username=None*, *password=None*, *key_filename=None*, *allow_agent=True*, *hostkey_verify=True*, *hostkey=None*, *look_for_keys=True*, *ssh_config=None*, *bind_addr=None*])

Connect via SSH and initialize the NETCONF session. First attempts the publickey authentication method and then password authentication.

To disable attempting publickey authentication altogether, call with *allow_agent* and *look_for_keys* as *False*.

host is the hostname or IP address to connect to

port is by default 830 (PORT_NETCONF_DEFAULT), but some devices use the default SSH port of 22 so this may need to be specified

timeout is an optional timeout for socket connect

unknown_host_cb is called when the server host key is not recognized. It takes two arguments, the hostname and the fingerprint (see the signature of `default_unknown_host_cb()`)

username is the username to use for SSH authentication

password is the password used if using password authentication, or the passphrase to use for unlocking keys that require it

key_filename is a filename where a the private key to be used can be found

allow_agent enables querying SSH agent (if found) for keys

hostkey_verify enables hostkey verification from ~/.ssh/known_hosts

hostkey_b64 only connect when server presents a public hostkey matching this (obtain from server /etc/ssh/ssh_host_*pub or ssh-keyscan)

look_for_keys enables looking in the usual locations for ssh keys (e.g. `~/.ssh/id_*`)

ssh_config enables parsing of an OpenSSH configuration file, if set to its path, e.g. `~/.ssh/config` or to True (in this case, use `~/.ssh/config`).

sock_fd is an already open socket which shall be used for this connection. Useful for NETCONF outbound ssh. Use host=None together with a valid sock_fd number

bind_addr is a (local) source IP address to use, must be reachable from the remote device.

sock is an already open Python socket to be used for this connection.

keepalive Turn on/off keepalive packets (default is off). If this is set, after interval seconds without sending any data over the connection, a "keepalive" packet will be sent (and ignored by the remote host). This can be useful to keep connections alive over a NAT.

environment a dictionary containing the name and respective values to set

load_known_hosts(*filename=None*)

Load host keys from an openssh `known_hosts`-style file. Can be called multiple times.

If *filename* is not specified, looks in the default locations i.e. `~/.ssh/known_hosts` and `~/ssh/known_hosts` for Windows.

property transport

Underlying paramiko.Transport object. This makes it possible to call methods like `set_keepalive()` on it.

Errors

exception ncclient.transport.**TransportError**

Bases: `NCClientError`

exception ncclient.transport.**SessionCloseError**(*in_buf*, *out_buf=None*)

Bases: *TransportError*

exception ncclient.transport.**SSHError**

Bases: *TransportError*

exception ncclient.transport.**AuthenticationError**

Bases: *TransportError*

exception ncclient.transport.**SSHUnknownHostError**(*host*, *fingerprint*)

Bases: *SSHError*

1.2.4 operations – Everything RPC

class ncclient.operations.**RaiseMode**

Define how errors indicated by RPC should be handled.

Note that any error_filters defined in the device handler will still be applied, even if ERRORS or ALL is defined: If the filter matches, an exception will NOT be raised.

ALL = 2

Don't look at the *error-type*, always raise.

ERRORS = 1

Raise only when the *error-type* indicates it is an honest-to-god error.

NONE = 0

Don't attempt to raise any type of *rpc-error* as *RPCError*.

Base classes

class ncclient.operations.**RPC**(*session*, *device_handler*, *async_mode=False*, *timeout=30*, *raise_mode=0*, *huge_tree=False*)

Base class for all operations, directly corresponding to *rpc* requests. Handles making the request, and taking delivery of the reply.

session is the *Session* instance

device_handler" is the :class: `~ncclient.devices..*DeviceHandler` instance

async specifies whether the request is to be made asynchronously, see *is_async*

timeout is the timeout for a synchronous request, see *timeout*

raise_mode specifies the exception raising mode, see *raise_mode*

huge_tree parse xml with huge_tree support (e.g. for large text config retrieval), see *huge_tree*

DEPENDS = []

Subclasses can specify their dependencies on capabilities as a list of URI's or abbreviated names, e.g. ':writable-running'. These are verified at the time of instantiation. If the capability is not available, *MissingCapabilityError* is raised.

REPLY_CLS

alias of *RPCReply*

_assert(*capability*)

Subclasses can use this method to verify that a capability is available with the NETCONF server, before making a request that requires it. A *MissingCapabilityError* will be raised if the capability is not available.

_request(*op*)

Implementations of *request()* call this method to send the request and process the reply.

In synchronous mode, blocks until the reply is received and returns *RPCReply*. Depending on the *raise_mode* a *rpc-error* element in the reply may lead to an *RPCError* exception.

In asynchronous mode, returns immediately, returning *self*. The *event* attribute will be set when the reply has been received (see *reply*) or an error occured (see *error*).

op is the operation to be requested as an `Element`

property error

`Exception` type if an error occured or *None*.

> **Note**
>
> This represents an error which prevented a reply from being received. An *rpc-error* does not fall in that category – see *RPCReply* for that.

property event

`Event` that is set when reply has been received or when an error preventing delivery of the reply occurs.

property huge_tree

Whether *huge_tree* support for XML parsing of RPC replies is enabled (default=False)

property is_async

Specifies whether this RPC will be / was requested asynchronously. By default RPC's are synchronous.

property raise_mode

Depending on this exception raising mode, an *rpc-error* in the reply may be raised as an *RPCError* exception. Valid values are the constants defined in *RaiseMode*.

property reply

RPCReply element if reply has been received or *None*

request()

Subclasses must implement this method. Typically only the request needs to be built as an `Element` and everything else can be handed off to *_request()*.

property timeout

Timeout in seconds for synchronous waiting defining how long the RPC request will block on a reply before raising *TimeoutExpiredError*.

Irrelevant for asynchronous usage.

class ncclient.operations.**RPCReply**(*raw*, *huge_tree=False*, *parsing_error_transform=None*)

Represents an *rpc-reply*. Only concerns itself with whether the operation was successful.

raw: the raw unparsed reply

huge_tree: parse XML with very deep trees and very long text content

> **Note**
>
> If the reply has not yet been parsed there is an implicit, one-time parsing overhead to accessing some of the attributes defined by this class.

_parsing_hook(*root*)

No-op by default. Gets passed the *root* element for the reply.

property error

Returns the first *RPCError* and *None* if there were no errors.

property errors

List of *RPCError* objects. Will be empty if there were no *rpc-error* elements in reply.

property ok

Boolean value indicating if there were no errors.

property xml

rpc-reply element as returned.

exception ncclient.operations.**RPCError**(*raw*, *errs=None*)

Bases: *OperationError*

Represents an *rpc-error*. It is a type of *OperationError* and can be raised as such.

property info

XML string or *None*; representing the *error-info* element.

property message

The contents of the *error-message* element if present or *None*.

property path

The contents of the *error-path* element if present or *None*.

property severity

The contents of the *error-severity* element.

property tag

The contents of the *error-tag* element.

property type

The contents of the *error-type* element.

Operations

Retrieval

class ncclient.operations.**Get**(*session*, *device_handler*, *async_mode=False*, *timeout=30*, *raise_mode=0*, *huge_tree=False*)

Bases: *RPC*

The *get* RPC.

session is the *Session* instance

device_handler" is the :class: `~ncclient.devices..*DeviceHandler` instance

async specifies whether the request is to be made asynchronously, see `is_async`

timeout is the timeout for a synchronous request, see `timeout`

raise_mode specifies the exception raising mode, see `raise_mode`

huge_tree parse xml with huge_tree support (e.g. for large text config retrieval), see `huge_tree`

REPLY_CLS = <class 'ncclient.operations.retrieve.GetReply'>

See *GetReply*.

request(*filter=None*, *with_defaults=None*)

Retrieve running configuration and device state information.

filter specifies the portion of the configuration to retrieve (by default entire configuration is retrieved)

with_defaults defines an explicit method of retrieving default values from the configuration (see **RFC 6243**)

Seealso

Filter parameters

class ncclient.operations.**GetConfig**(*session*, *device_handler*, *async_mode=False*, *timeout=30*, *raise_mode=0*, *huge_tree=False*)

Bases: *RPC*

The *get-config* RPC.

session is the *Session* instance

device_handler" is the :class: `~ncclient.devices..*DeviceHandler` instance

async specifies whether the request is to be made asynchronously, see `is_async`

timeout is the timeout for a synchronous request, see `timeout`

raise_mode specifies the exception raising mode, see `raise_mode`

huge_tree parse xml with huge_tree support (e.g. for large text config retrieval), see `huge_tree`

REPLY_CLS = <class 'ncclient.operations.retrieve.GetReply'>

See *GetReply*.

request(*source*, *filter=None*, *with_defaults=None*)

Retrieve all or part of a specified configuration.

source name of the configuration datastore being queried

filter specifies the portion of the configuration to retrieve (by default entire configuration is retrieved)

with_defaults defines an explicit method of retrieving default values from the configuration (see **RFC 6243**)

Seealso

Filter parameters

class ncclient.operations.**GetReply**(*raw*, *huge_tree=False*, *parsing_error_transform=None*)

Bases: *RPCReply*

Adds attributes for the *data* element to *RPCReply*.

property data

Same as *data_ele*

property data_ele

data element as an `Element`

property data_xml

data element as an XML string

class ncclient.operations.**Dispatch**(*session*, *device_handler*, *async_mode=False*, *timeout=30*, *raise_mode=0*, *huge_tree=False*)

Bases: *RPC*

Generic retrieving wrapper

session is the *Session* instance

device_handler" is the :class: `~ncclient.devices..*DeviceHandler` instance

async specifies whether the request is to be made asynchronously, see `is_async`

timeout is the timeout for a synchronous request, see `timeout`

raise_mode specifies the exception raising mode, see `raise_mode`

huge_tree parse xml with huge_tree support (e.g. for large text config retrieval), see `huge_tree`

REPLY_CLS = <class 'ncclient.operations.rpc.RPCReply'>

See *RPCReply*.

request(*rpc_command*, *source=None*, *filter=None*)

rpc_command specifies rpc command to be dispatched either in plain text or in xml element format (depending on command)

source name of the configuration datastore being queried

filter specifies the portion of the configuration to retrieve (by default entire configuration is retrieved)

Seealso

Filter parameters

Examples of usage:

```
dispatch('clear-arp-table')
```

or dispatch element like

```
xsd_fetch = new_ele('get-xnm-information')
sub_ele(xsd_fetch, 'type').text="xml-schema"
sub_ele(xsd_fetch, 'namespace').text="junos-configuration"
dispatch(xsd_fetch)
```

class ncclient.operations.**GetSchema**(*session*, *device_handler*, *async_mode=False*, *timeout=30*, *raise_mode=0*, *huge_tree=False*)

Bases: *RPC*

The *get-schema* RPC.

session is the `Session` instance

*device_handler" is the :class: `~ncclient.devices..*DeviceHandler` instance

async specifies whether the request is to be made asynchronously, see `is_async`

timeout is the timeout for a synchronous request, see `timeout`

raise_mode specifies the exception raising mode, see `raise_mode`

huge_tree parse xml with huge_tree support (e.g. for large text config retrieval), see `huge_tree`

REPLY_CLS = <class 'ncclient.operations.retrieve.GetSchemaReply'>

See `GetReply`.

request(*identifier*, *version=None*, *format=None*)

Retrieve a named schema, with optional revision and type.

identifier name of the schema to be retrieved

version version of schema to get

format format of the schema to be retrieved, yang is the default

Seealso

Filter parameters

Editing

class ncclient.operations.**EditConfig**(*session*, *device_handler*, *async_mode=False*, *timeout=30*, *raise_mode=0*, *huge_tree=False*)

Bases: `RPC`

edit-config RPC

session is the `Session` instance

*device_handler" is the :class: `~ncclient.devices..*DeviceHandler` instance

async specifies whether the request is to be made asynchronously, see `is_async`

timeout is the timeout for a synchronous request, see `timeout`

raise_mode specifies the exception raising mode, see `raise_mode`

huge_tree parse xml with huge_tree support (e.g. for large text config retrieval), see `huge_tree`

request(*config*, *format='xml'*, *target='candidate'*, *default_operation=None*, *test_option=None*, *error_option=None*)

Loads all or part of the specified *config* to the *target* configuration datastore.

target is the name of the configuration datastore being edited

config is the configuration, which must be rooted in the *config* element. It can be specified either as a string or an `Element`.

default_operation if specified must be one of { *"merge"*, *"replace"*, or *"none"* }

test_option if specified must be one of { *"test-then-set"*, *"set"*, *"test-only"* }

error_option if specified must be one of { *"stop-on-error"*, *"continue-on-error"*, *"rollback-on-error"* }

The *"rollback-on-error" error_option* depends on the *:rollback-on-error* capability.

class ncclient.operations.**DeleteConfig**(*session*, *device_handler*, *async_mode=False*, *timeout=30*, *raise_mode=0*, *huge_tree=False*)

Bases: *RPC*

delete-config RPC

session is the `Session` instance

*device_handler" is the :class: `~ncclient.devices..*DeviceHandler` instance

async specifies whether the request is to be made asynchronously, see `is_async`

timeout is the timeout for a synchronous request, see `timeout`

raise_mode specifies the exception raising mode, see `raise_mode`

huge_tree parse xml with huge_tree support (e.g. for large text config retrieval), see `huge_tree`

request(*target*)

Delete a configuration datastore.

target specifies the name or URL of configuration datastore to delete

Seealso

Source and target parameters

class ncclient.operations.**CopyConfig**(*session*, *device_handler*, *async_mode=False*, *timeout=30*, *raise_mode=0*, *huge_tree=False*)

Bases: *RPC*

copy-config RPC

session is the `Session` instance

*device_handler" is the :class: `~ncclient.devices..*DeviceHandler` instance

async specifies whether the request is to be made asynchronously, see `is_async`

timeout is the timeout for a synchronous request, see `timeout`

raise_mode specifies the exception raising mode, see `raise_mode`

huge_tree parse xml with huge_tree support (e.g. for large text config retrieval), see `huge_tree`

request(*source*, *target*)

Create or replace an entire configuration datastore with the contents of another complete configuration datastore.

source is the name of the configuration datastore to use as the source of the copy operation or *config* element containing the configuration subtree to copy

target is the name of the configuration datastore to use as the destination of the copy operation

Seealso

Source and target parameters

class ncclient.operations.**Validate**(*session*, *device_handler*, *async_mode=False*, *timeout=30*, *raise_mode=0*, *huge_tree=False*)

Bases: *RPC*

validate RPC. Depends on the *:validate* capability.

session is the `Session` instance

*device_handler" is the :class: `~ncclient.devices..*DeviceHandler` instance

async specifies whether the request is to be made asynchronously, see `is_async`

timeout is the timeout for a synchronous request, see `timeout`

raise_mode specifies the exception raising mode, see `raise_mode`

huge_tree parse xml with huge_tree support (e.g. for large text config retrieval), see `huge_tree`

request(*source='candidate'*)

Validate the contents of the specified configuration.

source is the name of the configuration datastore being validated or *config* element containing the configuration subtree to be validated

Seealso

Source and target parameters

class ncclient.operations.**Commit**(*session*, *device_handler*, *async_mode=False*, *timeout=30*, *raise_mode=0*, *huge_tree=False*)

Bases: *RPC*

commit RPC. Depends on the *:candidate* capability, and the *:confirmed-commit*.

session is the `Session` instance

device_handler" is the :class: `~ncclient.devices..*DeviceHandler` instance

async specifies whether the request is to be made asynchronously, see `is_async`

timeout is the timeout for a synchronous request, see `timeout`

raise_mode specifies the exception raising mode, see `raise_mode`

huge_tree parse xml with huge_tree support (e.g. for large text config retrieval), see `huge_tree`

request(*confirmed=False*, *timeout=None*, *persist=None*, *persist_id=None*)

Commit the candidate configuration as the device's new current configuration. Depends on the *:candidate* capability.

A confirmed commit (i.e. if *confirmed* is *True*) is reverted if there is no followup commit within the *timeout* interval. If no timeout is specified the confirm timeout defaults to 600 seconds (10 minutes). A confirming commit may have the *confirmed* parameter but this is not required. Depends on the *:confirmed-commit* capability.

confirmed whether this is a confirmed commit

timeout specifies the confirm timeout in seconds

persist make the confirmed commit survive a session termination, and set a token on the ongoing confirmed commit

persist_id value must be equal to the value given in the <persist> parameter to the original <commit> operation.

class ncclient.operations.**DiscardChanges**(*session*, *device_handler*, *async_mode=False*, *timeout=30*, *raise_mode=0*, *huge_tree=False*)

Bases: *RPC*

discard-changes RPC. Depends on the *:candidate* capability.

session is the `Session` instance

device_handler" is the :class: `~ncclient.devices..*DeviceHandler` instance

async specifies whether the request is to be made asynchronously, see `is_async`

timeout is the timeout for a synchronous request, see `timeout`

raise_mode specifies the exception raising mode, see `raise_mode`

huge_tree parse xml with huge_tree support (e.g. for large text config retrieval), see `huge_tree`

request()

Revert the candidate configuration to the currently running configuration. Any uncommitted changes are discarded.

class ncclient.operations.**CancelCommit**(*session*, *device_handler*, *async_mode=False*, *timeout=30*, *raise_mode=0*, *huge_tree=False*)

Bases: *RPC*

cancel-commit RPC. Depends on the *:candidate* and *:confirmed-commit* capabilities.

session is the *Session* instance

device_handler" is the :class: `~ncclient.devices..*DeviceHandler` instance

async specifies whether the request is to be made asynchronously, see `is_async`

timeout is the timeout for a synchronous request, see `timeout`

raise_mode specifies the exception raising mode, see `raise_mode`

huge_tree parse xml with huge_tree support (e.g. for large text config retrieval), see `huge_tree`

request(*persist_id=None*)

Cancel an ongoing confirmed commit. Depends on the *:candidate* and *:confirmed-commit* capabilities.

persist-id value must be equal to the value given in the <persist> parameter to the previous <commit> operation.

Flowmon

class ncclient.operations.**PoweroffMachine**(*session*, *device_handler*, *async_mode=False*, *timeout=30*, *raise_mode=0*, *huge_tree=False*)

Bases: *RPC*

poweroff-machine RPC (flowmon)

session is the *Session* instance

device_handler" is the :class: `~ncclient.devices..*DeviceHandler` instance

async specifies whether the request is to be made asynchronously, see `is_async`

timeout is the timeout for a synchronous request, see `timeout`

raise_mode specifies the exception raising mode, see `raise_mode`

huge_tree parse xml with huge_tree support (e.g. for large text config retrieval), see `huge_tree`

request()

Subclasses must implement this method. Typically only the request needs to be built as an `Element` and everything else can be handed off to `_request()`.

class ncclient.operations.**RebootMachine**(*session*, *device_handler*, *async_mode=False*, *timeout=30*, *raise_mode=0*, *huge_tree=False*)

Bases: *RPC*

reboot-machine RPC (flowmon)

session is the `Session` instance

*device_handler" is the :class: `~ncclient.devices..*DeviceHandler` instance

async specifies whether the request is to be made asynchronously, see `is_async`

timeout is the timeout for a synchronous request, see `timeout`

raise_mode specifies the exception raising mode, see `raise_mode`

huge_tree parse xml with huge_tree support (e.g. for large text config retrieval), see `huge_tree`

request()

Subclasses must implement this method. Typically only the request needs to be built as an `Element` and everything else can be handed off to `_request()`.

Locking

class ncclient.operations.**Lock**(*session*, *device_handler*, *async_mode=False*, *timeout=30*, *raise_mode=0*, *huge_tree=False*)

Bases: *RPC*

lock RPC

session is the `Session` instance

*device_handler" is the :class: `~ncclient.devices..*DeviceHandler` instance

async specifies whether the request is to be made asynchronously, see `is_async`

timeout is the timeout for a synchronous request, see `timeout`

raise_mode specifies the exception raising mode, see `raise_mode`

huge_tree parse xml with huge_tree support (e.g. for large text config retrieval), see `huge_tree`

request(*target='candidate'*)

Allows the client to lock the configuration system of a device.

target is the name of the configuration datastore to lock

class ncclient.operations.**Unlock**(*session*, *device_handler*, *async_mode=False*, *timeout=30*, *raise_mode=0*, *huge_tree=False*)

Bases: *RPC*

unlock RPC

session is the `Session` instance

*device_handler" is the :class: `~ncclient.devices..*DeviceHandler` instance

async specifies whether the request is to be made asynchronously, see `is_async`

timeout is the timeout for a synchronous request, see `timeout`

raise_mode specifies the exception raising mode, see `raise_mode`

huge_tree parse xml with huge_tree support (e.g. for large text config retrieval), see `huge_tree`

request(*target='candidate'*)

Release a configuration lock, previously obtained with the lock operation.

target is the name of the configuration datastore to unlock

Session

class ncclient.operations.**CloseSession**(*session*, *device_handler*, *async_mode=False*, *timeout=30*, *raise_mode=0*, *huge_tree=False*)

Bases: *RPC*

close-session RPC. The connection to NETCONF server is also closed.

session is the *Session* instance

device_handler" is the :class: `~ncclient.devices..*DeviceHandler` instance

async specifies whether the request is to be made asynchronously, see `is_async`

timeout is the timeout for a synchronous request, see `timeout`

raise_mode specifies the exception raising mode, see `raise_mode`

huge_tree parse xml with huge_tree support (e.g. for large text config retrieval), see `huge_tree`

request()

Request graceful termination of the NETCONF session, and also close the transport.

class ncclient.operations.**KillSession**(*session*, *device_handler*, *async_mode=False*, *timeout=30*, *raise_mode=0*, *huge_tree=False*)

Bases: *RPC*

kill-session RPC.

session is the *Session* instance

device_handler" is the :class: `~ncclient.devices..*DeviceHandler` instance

async specifies whether the request is to be made asynchronously, see `is_async`

timeout is the timeout for a synchronous request, see `timeout`

raise_mode specifies the exception raising mode, see `raise_mode`

huge_tree parse xml with huge_tree support (e.g. for large text config retrieval), see `huge_tree`

request(*session_id*)

Force the termination of a NETCONF session (not the current one!)

session_id is the session identifier of the NETCONF session to be terminated as a string

Subscribing

class ncclient.operations.**CreateSubscription**(*session*, *device_handler*, *async_mode=False*, *timeout=30*, *raise_mode=0*, *huge_tree=False*)

Bases: *RPC*

create-subscription RPC. Depends on the *:notification* capability.

session is the *Session* instance

device_handler" is the :class: `~ncclient.devices..*DeviceHandler` instance

async specifies whether the request is to be made asynchronously, see `is_async`

timeout is the timeout for a synchronous request, see `timeout`

raise_mode specifies the exception raising mode, see `raise_mode`

huge_tree parse xml with huge_tree support (e.g. for large text config retrieval), see `huge_tree`

request(*filter=None*, *stream_name=None*, *start_time=None*, *stop_time=None*)

Creates a subscription for notifications from the server.

filter specifies the subset of notifications to receive (by default all notificaitons are received)

Seealso

Filter parameters

stream_name specifies the notification stream name. The default is None meaning all streams.

start_time triggers the notification replay feature to replay notifications from the given time. The default is None, meaning that this is not a replay subscription. The format is an RFC 3339/ISO 8601 date and time.

stop_time indicates the end of the notifications of interest. This parameter must be used with *start_time*. The default is None, meaning that (if *start_time* is present) the notifications will continue until the subscription is terminated. The format is an RFC 3339/ISO 8601 date and time.

Exceptions

exception ncclient.operations.**OperationError**

Bases: NCClientError

exception ncclient.operations.**MissingCapabilityError**

Bases: NCClientError

exception ncclient.operations.**TimeoutExpiredError**

Bases: NCClientError

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

X