NCClient Documentation

Release 0.1.1a

Shikhar Bhushan

May 16, 2009

CONTENTS

CHAPTER

ONE

USER DOCUMENTATION

1.1 `manager` module

TODO

1.2 `transport` module

1.2.1 Base types

class **`Session`**(*capabilities*)

Base class for use by transport protocol implementations.

`add_listener`(*listener*)

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

Parameter *listener* – `SessionListener`

`remove_listener`(*listener*)

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

`get_listener_instance`(*cls*)

If a listener of the sspecified type is registered, returns the instance. This is useful when it is desirable to have only one instance of a particular type per session, i.e. a multiton.

Parameter *cls* – class of the listener

`client_capabilities`

Client's `Capabilities`

`server_capabilities`

Server's `Capabilities`

`connected`

Connection status of the session.

`id`

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

`can_pipeline`

Whether this session supports pipelining

class **`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.

Parameters • *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* (`string`) – XML document

errback (*ex*)
Called when an error occurs.

1.2.2 SSH session implementation

static **default_unknown_host_cb** (*host, key*)
An `unknown host callback` returns `True` if it finds the key acceptable, and `False` if not.

Parameters • *host* – the hostname/address which needs to be verified
• *key* – a hex string representing the host key fingerprint

Returns this default callback always returns `False`

class SSHSession (*capabilities*)
Bases: `ncclient.transport.session.Session`

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

connect (*host, [port=830, timeout=None, username=None, password=None, key_filename=None, allow_agent=True, look_for_keys=True]*)
Connect via SSH and initialize the NETCONF session. First attempts the publickey authentication method and then password authentication.

To disable publickey authentication, call with *allow_agent* and *look_for_keys* as `False`

Parameters • *host* – the hostname or IP address to connect to
• *port* – by default 830, but some devices use the default SSH port of 22 so this may need to be specified
• *timeout* – an optional timeout for the TCP handshake
• *unknown_host_cb* – called when a host key is not known. See `unknown_host_cb()` for details on signature
• *username* – the username to use for SSH authentication
• *password* – the password used if using password authentication, or the passphrase to use in order to unlock keys that require it
• *key_filename* – a filename where a the private key to be used can be found
• *allow_agent* – enables querying SSH agent (if found) for keys
• *look_for_keys* – enables looking in the usual locations for ssh keys (e.g. `~/.ssh/id_*`)

transport
The underlying paramiko.Transport object. This makes it possible to call methods like set_keepalive on it.

1.2.3 Errors

1.3 `operations` module

1.3.1 Base type for operations

class RPC (*session, async=False, timeout=None*)
Directly corresponds to <*rpc*> requests. Handles making the request, and taking delivery of the reply.

set_async (*async=True*)
Set asynchronous mode for this RPC.

set_timeout(*timeout*)
Set the timeout for synchronous waiting defining how long the RPC request will block on a reply before raising an error.

reply
`RPCReply` element if reply has been received or `None`

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

This attribute should be checked if the request was made asynchronously, so that it can be determined if `event` being set is because of a reply or error.

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.

event
`Event` that is set when reply has been received or error occured.

async
Whether this RPC is asynchronous

timeout
Timeout for synchronous waiting

id
The *message-id* for this RPC

session
The `Session` object associated with this RPC

1.3.2 RPC replies

class RPCReply(*raw*)
Represents an *<rpc-reply>*. Only concerns itself with whether the operation was successful. Note that if the reply has not yet been parsed there is a one-time parsing overhead to accessing the `ok` and `error`/`errors` attributes.

ok
Boolean value indicating if there were no errors.

error
Short for `errors[0], returning :const:'None` if there were no errors.

class RPCError(*err_dict*)
Bases: `ncclient.operations.errors.OperationError`

Represents an *<rpc-error>*. It is an instance of `OperationError` so it can be raised like any other exception.

type
`string` represeting *error-type* element

severity
`string` represeting *error-severity* element

tag
`string` represeting *error-tag* element

path
`string` or `None`; represeting *error-path* element

message
`string` or `None`; represeting *error-message* element

info
`string` or `None`, represeting *error-info* element

1.3.3 NETCONF Operations

Operations may have a hard dependency on some being capability, or the dependency may depend on user-supplied parameters. In any case, a `MissingCapabilityError` is raised if the server is found to not support the pertinent capability.

The return type for the `request()` methods depends on whether it was instantiated as being asynchronous.

- For asynchronous requests, it will immediately return a `Event` object. This event will be set when a reply is received or an error occurs that prevents a reply from being received. When the event is set, the `reply` and
- For synchronous requests, it will block on the reply

General notes on parameters

Unless otherwise noted, parameters to **:req:'request'** are strings.

Source / Target

Where an operation takes a source or target parameter, it is mainly the case that it can be a datastore name or a URL. The latter, of course, depends on the *:url* capability and whether the capability supports the specific schema of the URL.

Filter expressions

The filter parameter to `request()`, where applicable, can take one of the following types:

- **A `tuple` of *(type, criteria)*.** Here type has to be one of '`"xpath"`' or '`"subtree"`'. The criteria should be an XPath expression for the "xpath" type and in *DictTree XML representation* for a subtree filter.
- A well-formed <*filter*> element in *DictTree XML representation*.

Retrieval

The reply object for these operations will be a `GetReply` instance.

class `Get` (*session, async=False, timeout=None*)

Bases: `ncclient.operations.rpc.RPC`

<*get*> RPC

`request` (*[filter=None]*)

See *source_url*, *filter*.

class `GetConfig` (*session, async=False, timeout=None*)

Bases: `ncclient.operations.rpc.RPC`

<*get-config*> RPC

`request` (*source, [filter=None]*)

See *source_url*, *filter*.

class `GetReply` (*raw*)

Bases: `ncclient.operations.rpc.RPCReply`

Adds attributes for the <*data*> element to `RPCReply`, pertinent to the <*get*> or <*get-config*> operations.

`data`

As an `Element`

`data_xml`

As an XML string

Locking

class **Lock**(*session, async=False, timeout=None*)
Bases: ncclient.operations.rpc.RPC

class **Unlock**(*session, async=False, timeout=None*)
Bases: ncclient.operations.rpc.RPC

class **LockContext**(*session, target*)

Configuration

class **EditConfig**(*session, async=False, timeout=None*)
Bases: ncclient.operations.rpc.RPC

class **CopyConfig**(*session, async=False, timeout=None*)
Bases: ncclient.operations.rpc.RPC

class **CopyConfig**(*session, async=False, timeout=None*)
Bases: ncclient.operations.rpc.RPC

class **Validate**(*session, async=False, timeout=None*)
Bases: ncclient.operations.rpc.RPC

class **Commit**(*session, async=False, timeout=None*)
Bases: ncclient.operations.rpc.RPC

class **DiscardChanges**(*session, async=False, timeout=None*)
Bases: ncclient.operations.rpc.RPC

request(**args, **kwds*)
Subclasses implement this method. Here, the operation is to be constructed as a *DictTree XML representation*, and the result of _request() returned.

class **ConfirmedCommit**(*session, async=False, timeout=None*)
Bases: ncclient.operations.edit.Commit

psuedo-op

request(**args, **kwds*)
Commit changes; requiring that a confirming commit follow

Session management

class **CloseSession**(*session, async=False, timeout=None*)
Bases: ncclient.operations.rpc.RPC

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

request(**args, **kwds*)
Subclasses implement this method. Here, the operation is to be constructed as a *DictTree XML representation*, and the result of _request() returned.

class **KillSession**(*session, async=False, timeout=None*)
Bases: ncclient.operations.rpc.RPC

<*kill-session*> RPC.

1.4 content module

The content module provides methods for creating XML documents, parsing XML, and converting between different XML representations. It uses ElementTree internally.

1.4.1 Namespaces

The following namespace is defined in this module.

BASE_NS
Base NETCONf namespace

Namespaces are handled just the same way as `ElementTree`. So a qualified name takes the form `{namespace}tag`. There are some utility functions for qualified names:

qualify(*tag, [ns=BASE_NS]*)

Returns qualified name

unqualify(*tag*)

Returns unqualified name

Note: It is strongly recommended to compare qualified names.

1.4.2 DictTree XML representation

`ncclient` can make use of a special syntax for XML based on Python dictionaries. It is best illustrated through an example:

```
dtree = {
    'tag': '{ns}a',
    'attrib': {'attr': 'val'},
    'subtree': [ { 'tag': 'child1' }, { 'tag': 'child2', 'text': 'some text' } ]
}
```

Calling `dtree2xml()` on *dtree* would return

```
<?xml version="1.0" encoding="UTF-8"?>
<ns0:a attr="val" xmlns:ns0="ns">
    <child1 />
    <child2>some text</child2>
</ns0:a>
```

In addition to a 'pure' dictionary representation a DictTree node (including the root node) may be an XML literal or an `Element` instances. The above example could thus be equivalently written as:

```
dtree2 = {
    'tag': '{ns}a',
    'attrib': {'attr': 'val'},
    'subtree': [ ET.Element('child1'), '<child2>some text</child2>' ]
}
```

1.4.3 Converting between different representations

Conversions *to* DictTree representation are guaranteed to be entirely dictionaries. In converting *from* DictTree representation, the argument may be any valid representation as specified.

dtree2ele(*spec*)
DictTree -> Element

Return type `Element`

dtree2xml(*spec, [encoding="UTF-8"]*)
DictTree -> XML

Parameter *encoding* – chraracter encoding

Return type string

ele2dtree(*ele*)
DictTree -> Element

Return type `dict`

ele2xml(*ele*)
Element -> XML

Parameter *encoding* – character encoding
Return type `string`

xml2dtree(*xml*)
XML -> DictTree

Return type `dict`

xml2ele(*xml*)
XML -> Element

Return type `Element`

1.4.4 Other utility functions

iselement(*obj*)

See `xml.etree.ElementTree.iselement()`

find(*ele, tag, [nslist=, []]*)
If `nslist` is empty, same as `xml.etree.ElementTree.Element.find()`. If it is not, `tag` is interpreted as an unqualified name and qualified using each item in `nslist`. The first match is returned.

Parameter *nslist* – optional list of namespaces

parse_root(*raw*)
Efficiently parses the root element of an XML document.

Returns a tuple of `(tag, attributes)`, where `tag` is the (qualified) name of the element and `attributes` is a dictionary of its attributes.

validated_element(*rep, tag=None, attrs=None, text=None*)
Checks if the root element meets the supplied criteria. Returns a `Element` instance if so, otherwise raises `ContentError`.

Parameters
- *tag* – tag name or a list of allowable tag names
- *attrs* – list of required attribute names, each item may be a list of allowable alternatives
- *text* – textual content to match

1.4.5 Errors

exception ContentError
Bases: `ncclient.NCClientError`

Raised by methods of the `content` module in case of an error.

1.5 capabilities module

class Capabilities(*capabilities=None*)
Represent the capabilities of client or server. Also facilitates using abbreviated capability names in addition to complete URI.

CHAPTER

TWO

EXTENDING NCCLIENT

This is written in a ‘how-to’ style through code examples.

TODO

MODULE INDEX

N

INDEX

S

T

U

V

X