ncclient Documentation

Release 0.4.1

Shikhar Bhushan nd Leonidas Poulopoulos

February 22, 2014

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 4741**.
- 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 annd 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={'nar
    c = m.get_config(source='running').data_xml
    with open("%s.xml" % host, 'w') as f:
        f.write(c)
```

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

Contents:

CHAPTER 1

manager – High-level API

1.1 Customizing

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

1.2 Factory functions

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

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.

1.4 Special kinds of parameters

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

1.4.1 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.

1.4.2 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.
  - For *"subtree"* the *criteria* should be an XML string or an `Element` object containing the criteria.

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

Complete API documentation

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.

2.2 `xml_` – XML handling

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

exception `ncclient.xml_.`**`XMLError`**
Bases: `ncclient.NCClientError`

x.__init__(...) initializes x; see help(type(x)) for signature

2.2.1 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*.

2.2.2 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*)
Convert and return the `Element` for the XML document *x*. If *x* is already an `Element` simply returns that.

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.

2.3 `transport` – Transport / Session layer

2.3.1 Base types

2.3.2 SSH session implementation

2.3.3 Errors

2.4 `operations` – Everything RPC

2.4.1 Base classes

2.4.2 Operations

Retrieval

Editing

Locking

Session

2.4.3 Exceptions

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*