NCClient: A Python library for NETCONF clients

Shikhar Bhushan

Guided Research Final Report, Spring 2009
Department of Computer Science
Jacobs University Bremen
May 15, 2009

Supervisor:
Prof. Dr. J. Schönwälder
Acknowledgement

I am thankful for the advice and support provided by Prof. Juergen Schönwälder and Ha Tran in the course of this guided research project.
1 Introduction

The Network Configuration protocol (NETCONF) is a network management protocol developed by the Netconf Working Group of the IETF. It is a proposed internet standard since December 2006. [1] RFC 4741

[NETCONF] provides mechanisms to install, manipulate, and delete the configuration of network devices. It uses an Extensible Markup Language (XML)-based data encoding for the configuration data as well as the protocol messages. The NETCONF protocol operations are realized on top of a simple Remote Procedure Call (RPC) layer.

[1]

There has been good industry adoption of the protocol, and most network equipment vendors have either implemented or plan to implement it. It is a significant improvement over proprietary command line interfaces commonly used for configuring network devices. Efforts are also underway to create standard data models for NETCONF and data manipulated by it. [2] This will enable a great degree of interoperability.

While the protocol has been around for a few years now, there is an absence of free software tools and libraries for it. NCClient partially addresses this with a client library in Python. It facilitates creating scripts and applications for network configuration management with NETCONF. It exposes an intuitive API for managing the NETCONF session and making remote procedure calls to it.

NCClient is licensed under the Apache License and is publicly hosted at Google Code, where the source repository and bug tracker is located.

The architecture and design of NCClient is documented in this report. Interoperability issues and suggestions for future development are also discussed. User documentation and a how-to on extending NCClient is included as an Appendix.

2 Features

The most important features of NCClient can be summarized as thus:

- Modular, object-oriented API.
- Transport mapping for SSHv2.
- Supports request pipelining and correct handling of asynchronous protocol messages.
- Supports all operations and capabilities defined in RFC 4741. See Table[1]
- Support for (a)synchronous RPC requests.
- XML using Python dictionaries (optional).
• Extensible – new transport mappings and new capabilities/operations can be added.
• Thread-safe.

Table 1: Capability support

<table>
<thead>
<tr>
<th>Capability</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>:writable-running</td>
<td>Tested</td>
</tr>
<tr>
<td>:candidate</td>
<td>Tested</td>
</tr>
<tr>
<td>:confirmed-commit</td>
<td>Tested</td>
</tr>
<tr>
<td>:rollback-on-error</td>
<td>In principle</td>
</tr>
<tr>
<td>:startup</td>
<td>In principle</td>
</tr>
<tr>
<td>:url</td>
<td>Tested</td>
</tr>
<tr>
<td>:validate</td>
<td>Tested</td>
</tr>
<tr>
<td>:xpath</td>
<td>Tested</td>
</tr>
<tr>
<td>:notification</td>
<td>Not implemented</td>
</tr>
<tr>
<td>:interleave</td>
<td>In principle</td>
</tr>
</tbody>
</table>

3 Architecture

Figure 1 shows the RFC 4741 division of NETCONF into logical layers. The NCClient package closely follows this division, as shown in Figure 2.

Figure 3 shows the classes contained in these modules and the inheritance relationships between them, if any.
4 Design documentation

Here design decisions and certain implementation details which are important from a design point of view are discussed. Therefore, this section uses a lot of Python terminology.

As a general note, the logging module has been used for logging and this is very helpful for debugging the code.

4.1 transport module

SSH has been initially implemented because it is the mandatory transport for NETCONF and generally ubiquitous.

Session is a base type for a concrete implementation. Once connection is established, a thread is spawned, the main loop of which is implemented by the subclass. Session then handles the <hello> message exchange.

Inter-thread communication primarily takes the form of

- Messages that are to be sent
- Received messages / error event, to be dispatched.

For messages to be sent, the request is put into a Queue. Queue is a thread-safe data structure. The thread’s main loop is meant to consume items from the queue.

Received messages are dispatched to attached SessionListener instances. SessionListener provides an interface for a callback and an errback function.

A Session is thus a subject for passive SessionListener instances.

It should be noted that before dispatching a message, the start of the root element of the XML document is parsed, and this is one of the arguments to the callback in addition to the raw XML. This allows the listener’s callback to determine whether it is interested in further processing the message. For example, HelloHandler is only interested in <hello> messages, and
RPCReplyListener is interested in all <rpc-reply> messages and additionally needs to know the message-id attribute.

The above design allows for pipelining of requests and asynchronous messages to be correctly handled. Therefore, it will not be a problem to support the :interleave capability defined in RFC 5277 [3].

SSHSession is an implementation of the RFC 4742 specification for NETCONF over SSH [4]. It uses the paramiko SSHv2 library.

Initially, the high-level paramiko.SSHClient was being utilized. However, this turned out to not support IPv6, so an adapted implementation was written. A patch that fixes this issue in paramiko.SSHClient was also written, which has since been merged upstream. The decision to keep the adapted implementation was kept however, because it allows for finer control on the aspects of connection, host-key verification, and authentication.

Thanks to the extensive functionality of paramiko, SSHSession supports:

- taking into account system known_hosts as well as other specified files for this purpose
- the publickey, password, and keyboard-interactive (to some extent) authentication methods
- authenticating using ssh-agent

With regard to host verification in particular, the SSHSession.connect() function takes an optional callback argument which is invoked with the host key fingerprint when it cannot be verified. This allows the decision to rest with the API client. In case it cannot be verified through any means an exception is raised.

RFC 4742 provides for a special character sequence that must be sent by the server and client after each XML document in the NETCONF exchange. This is used as a basis for delimiting messages. A stateful parser was written such that a character that has been considered once is not considered again. Each parser call, in the worst case, parses READ_BUF_SIZE number of bytes.

4.2 operations module

This module corresponds to the RPC and Operation layers. The main functionality pertaining to making an RPC request and making sure the reply reaches the right object is implemented in the RPC and RPCReplyListener classes.

An RPC object is initialised with a Session instance. At any point the request can be declared asynchronous or an optional timeout for synchronous waiting can be specified.

The RPC object creates a RFC 4122, namespaced, Universally Unique Identifier (UUID) for the message-id attribute. While this may seem excessive, the decision to use a UUID instead of a monotonically increasing integer was based on the fact that for the latter option, a class attribute for the current
message-id would have to be maintained and appropriately protected for thread safety, while UUID creation is a simple function call.

RPC also creates a RPCReplyListener instance and the message-id of the RPC is registered with the listener. RPCReplyListener is a ‘multiton’, in that there may only exist one instance per session. It returns the pre-existing instance at object creation, if so. It maintains a message-id to RPC mapping.

Every operation subclasses from RPC. Operations implement the request() method, or if no parameters are required for that RPC, can simply specify a class attribute that completely defines the operation.

When a request is made on the operation, the subclass calls an internal RPC method which builds the request as an XML document and sends it over the session. In asynchronous mode, a threading.Event object is returned. In synchronous mode, it waits for timeout period (if not specified, indefinitely) for a reply to be delivered to it and returns this.

RPCReplyListener takes care of delivering an <rpc-reply> to the correct RPC object as previously registered, which creates an RPCReply object from it. RPCReplyListener’s registration dictionary is a WeakValueDictionary so as to not hinder garbage collection.

It is important to note here that if the server does not support pipelining, asynchronous operation is disallowed by RPC and there may thus be only one RPC pending at any given time. Pipelining is disabled for Cisco devices, which return invalid <rpc-reply> elements missing the message-id attribute in case of an <rpc-error>.

RPCReply objects represent a <rpc-reply> element. They have a simple boolean attribute for determining whether the reply was <ok/>. In case <rpc-error> elements are parsed, they store a list of RPCError objects. RPCError objects expose string attributes for the error type, severity, tag, path, and message. The error-info element is data model specific and is thus returned as an XML string.

Operations (subclasses of RPC) may define a custom reply class that inherits from RPCReply. This is the case with the Get and GetConfig operations whose reply class is GetReply. GetReply instances maintain a parsed <data> element since this is of primary interest in case of these operations.

Operations can specify their dependencies on capabilities. They may also verify the presence of a capability on the server-side during construction of a request, since the requirement of a capability may depend on the specific parameters a user supplies. For example, the :url capability need only be verified if the user supplies a URL as source or target to request().

All the capabilities and operations specified in RFC 4741 are implemented. In addition, there exists a context manager for the <lock> RPC – LockContext – for use with the with statement, that takes care of locking when the context is entered and unlocking on exit. A pseudo-operation ConfirmedCommit is also implemented.

There also exist several utility functions, like build_filter() (constructs the <filter> element from a user-supplied parameter which may be of sev-
eral types), store_or_url() (determines whether the argument represents a datastore name or a URL and constructs <source/>/<target> accordingly).

4.3 content module

The content module can be summed up as “all to do with XML processing”. “XML processing” may be further qualified with “syntactical”, which is as far as the functionality goes.

It utilizes the excellent xml.etree.ElementTree module internally. Namespace constants and methods for qualifying names are provided.

The module provides several functions for converting between different XML representations.

A unique feature is the possibility to use nested Python dictionaries to represent XML. The details about the syntax are presented in the user documentation. The motivation was that creating XML programatically can be clunky. Python dictionaries seemed to lend themselves naturally to this task.

For example, a RPC is represented internally as:

```python
spec = {
    'tag': content.qualify('rpc'),
    'attrib': {'message-id': self._id},
    'subtree': [opspec]
}
```

opspec depends on the operation. For example, KillSession could specify the following:

```python
opspec = {
    'tag': 'kill-session',
    'subtree': [ {'session-id': sid} ]
}
```

These dictionary representations can be (almost) freely mixed with XML literals and xml.etree.ElementTree.Element instances. So, for example, `opspec` could be the result of:

```python
opspec = ET.Element('kill-session')
sid_ele = ET.Element('session-id')
sid_ele.text = sid
opspec.append(sid_ele)
```

Or, directly, an XML literal:

```python
opspec = ('<kill-session><session-id>' + sid + ' </session-id></kill-session>')</n```

This extends to the user-specified parameters. For instance, the `config` parameter for an EditConfig operation could be in either of these representations.
The `content` module also provides several utility functions, such as for validating that the root element of a given XML representation matches some criteria.

### 4.4 manager module

The manager module is a facade for the NCClient API. Manager instances are created by the `connect_*()` family of factory functions (currently only `connect_ssh()`, with `connect()` being synonymous for it). Manager objects offer a more direct API for operations.

It is currently a thin abstraction layer around the rest of the library, but offers nearly all the features. There are many opportunities for further development.

### 4.5 capabilities module

This module is quite simple and contains a `Capabilities` class that represents a set of capabilities. It is a container class with the added feature that an abbreviated name may be specified when checking for the presence of a capability. The abbreviation is automatically inferred in case it is a standard capability. It also has a simple utility function for determining the schemes supported by a `:url` capability URI.

### 5 Code examples

This is an example of composing the different modules manually:

```python
from ncclient.transport import SSHSession
from ncclient import operations

session = SSHSession()
session.connect('broccoli', 22, 'sbhushan')

try:
    vrep = operations.Validate(session).request(
        source='file://new_config')
    if not vrep.ok:
        raise vrep.error

    with operations.LockContext(session, target='running'):
        cc1 = operations.CopyConfig(session).request(
            source='running', target='file://backup.conf')
        if not cc1.ok:
            raise cc1.error

        cc2 = operations.CopyConfig(session).request(
            source='file://new_config', target='running')
        if not cc2.ok:
            raise cc2.error
```

10
getreply = operations.GetConfig(session).request(
    source='running',
    filter=('xpath', '//the/xpath/filter'))
if getreply.ok:
    print(getreply.data)
else:
    raise getreply.error
except RPCError as e:
    print(e)

Equivalent example, demonstrating high-level API:

from ncclient import manager
m = manager.connect('broccoli', 22, 'sbhushan')
try:
    m.validate(target='file://new_config')
    with m.locked(target='running'):
        m.copy_config(source='running',
                      target='file://backup.conf')
        m.copy_config(source='file://new_config',
                      target='running')
        data = m.get_config(source='running',
                            filter=('xpath', '//the/xpath/filter'))
except RPCError as e:
    print(e)
else:
    print(data)

6 Interoperability

The library was tested against broccoli – Tail-f ConfD running on a virtual machine, and alex – a Cisco device, both part of the Computer Networks and Distributed Systems group’s netlab. At the time of writing, alex runs IOS version 12.4, and broccoli has ConfD version 2.8.1.

In testing, the Tail-f implementation was found to be standards-compliant and thus easy to work with. However, a number of problems were encountered with the Cisco implementation.

In its <hello> greeting, IOS does not namespace the XML. More problematic is that it uses an incorrect namespace for everything else. The base NETCONF namespace is defined as urn:ietf:params:xml:ns:netconf:base:1.0[1], whereas IOS purports it to be urn:ietf:params:netconf:base:1.0. Secondly, when an operation results in an <rpc-error> the parent <rpc-reply> element is sent without a message-id attribute. These issues required working around.

In general, the Cisco implementation seemed to be quite ad-hoc. This is apparent from the fact that it sends/receives configuration data as a text blob instead
There are many other implementations, such as Juniper’s and Nortel’s, that NCClient needs to be tested against.

7 Future development

As of writing, the code is tagged as 0.1.1 alpha. Wider testing is thus the best short-term target to set.

For the longer term, it has to be noted that the design was carefully thought out and it is hoped that no major design flaws are uncovered. The low-level details have been handled, but the library could do with more high-level constructs, for example in Manager. A great development would be a MultiManager API built on top of Manager, that provides support for multi-device configuration management.

More capabilities can be supported, such as :notification which was not implemented for lack of a test subject. At the time of writing, there are internet drafts from the Netconf working group for :with-defaults, :partial-lock, :netconf-monitoring and schema-retrieval capabilities [5].

More transport mappings can be implemented. There are already RFC’s for BEEP [6], SOAP [7] and TLS [8].

This project is just a beginning, but it fills a void. Considering that it is open source and the clients are programmers, it is hoped that the it will quickly mature.
References


APPENDIX: User Documentation

This page intentionally left blank.