REPLACING THE PYANG TOOL

tail-f a Cisco company

Table of Contents

1. Introduction **3**

2. Open Source pyang **3**

2.1 Installation 3

2.2 Install pyang System Wide 4

2.3 Install pyang for the Current User Only 4

2.4 Install pyang to a Virtual Python Environment 5

2.5 Reusing the tailf Plug-in with Open Source pyang 6

2.6 Replacing the tailf Plug-in for Annotation Processing 7

3. Other Alternatives **10**

3.1 Annotations Through Deviations 10

3.2 Using yanger 10

4. For More Information **10**

1. Introduction

The pyang tool used to be called “under the covers” by ConfD’s confdc tool. However, because pyang was written in Python, this had an impact on processing time. For quite some time now, confdc has instead been calling a new tool called “yanger”. The yanger tool is written in Erlang and shows significant performance improvement over pyang.

A deprecation notice for the pyang tool was included in the CHANGES file when ConfD 6.5 was released in Ocotober 2017. As of the ConfD 7.1 release in March 2019, the pyang tool will no longer be included in the ConfD distribution.

This application note looks at alternatives to the use of the ConfD distributed version of pyang and is useful for those ConfD users who use the pyang tool directly.

2. Open Source pyang

2.1 Installation

In many scenarios, the pyang version distributed with ConfD can be immediately replaced by its open source version as most pyang plugins will work with either one. ConfD included an older version of pyang. Because the open source pyang is a newer version, it has several advantages over the pyang tool which was included in the ConfD distribution.

The open source version of pyang is compatible with Python 3 while the ConfD version runs only in the Python 2 environment. It is important to note that Python 2 will reach its End-of-Life soon. Additionally, the open source version of pyang supports both YANG versions 1.0 and 1.1 while the ConfD version only supports YANG 1.0 modules.

The open source version of pyang is available either from a GitHub repository or from the Python Package Index (PyPI). Installing from PyPI is usually more convenient. There are several variants how the tool can be installed:

- Install pyang system wide
- Install pyang for the current user only
- Install pyang in a virtual Python environment

2.2 Install pyang System Wide

Install pyang system wide in order to make the tool available for all users on the system. The command *pip*, if not already installed, is available in your OS distribution package named like *python-pip.*

```
$ sudo pip install pyang
[...]
$ which pyang
/usr/local/bin/pyang
```

2.3 Install pyang for the Current User Only

This installs the package to the user's install directory, typically *~/.local/.* On some distributions, the installation directory may not be part of the environment variable PATH and a command like:

```
export PATH=$PATH:$HOME/.local/bin
```

may need to be added to the user's *.bashrc*.

```
$ pip install --user pyang
[...]
$ which pyang
/home/tester/.local/bin/pyang
```

2.4 Install pyang to a Virtual Python Environment

Python virtual environments can be created by `virtualenv`, available in your OS distribution package named like `python-virtualenv`.

```
$ virtualenv /path/to/env/directory
```

Once the environment exists, you can activate it using:

```
$ source /path/to/env/directory/bin/activate
(envname) $
```

All pip interaction is now run in the context of the environment:

```
(envname) $ pip install pyang
[...]
(envname) $ which pyang
/path/to/env/directory/bin/pyang
(envname) $
```

When you are done with your experiments or you want to switch to the default environment, deactivate it by running deactivate. There can be as many virtual environments as needed, system-wide or per-user.

2.5 Reusing the `tailf` Plug-in with Open Source pyang

Almost all the pyang plug-ins which were available in the ConfD distribution are included with the open source version. One notable exception is the `tailf` plugin. There are two common use cases for the `tailf` plugin - to *sanitize* a module, i.e. to remove some or all ConfD specific content, and to *annotate* a YANG module, i.e. in effect to add YANG statements to a module without touching the module itself.

There are plans to add `sanitize` as a general option in the open source pyang. So, if the plans are realized by the open source maintainers, the `tailf` plug-in will not be needed for this purpose.

While there is currently a replacement for *sanitize*, it is possible to copy use the `tailf` plug-in from a version of ConfD prior to 7.1 and with the open source version of pyang. When doing this, be aware of the following caveats: The `tailf` plugin is not compatible with Python 3 and needs a few modifications before it can be used in a Python 3 environment. Also, the `tailf` plugin is no longer maintained and if the pyang plug-in API changes for whatever reason (and the plug-in uses quite a bit more than just the standard parts of the API), the plug-in will no longer be compatible.

To use the plug-in, it is necessary to copy it out of its usual location in a ConfD distribution (that is from *$CONFD _ DIR/lib/pyang/pyang/plugins/tailf.py*) and place it in some other directory. This is because it needs to be separated from the other ConfD distributed plug-ins there as they would conflict with the plug-ins distributed with the open source version of pyang. To run pyang, with the `tailf` plugin, do the following:

```
$ pyang --plugindir /path/to/tailf/plugin <pyang options>
```

It is also possible to install the plug-in system wide using Python `setuptools`. The pyang tool looks for plug-ins registered with an entry point “*pyang.plugin*”.

2.6 Replacing the `tailf` Plug-in for Annotation Processing

For typical annotation scenarios, the `tailf` plug-in may not be necessary. Annotations are usually used to add an extension statement which is in turn processed by another pyang plug-in. If you are not doing so, then you do not need to be concerned about replacing this functionality. Typically, there is a *annotation module* containing parts like this:

```
import acme-module {
   prefix acme;
 }

 import my-extensions-module {
   prefix myex;
 }

 [...]

 tailf:annotate "/acme:c1/acme:c2" {
   myex:expand-node;
 }
```

Such an annotation module would be passed with "*`-a acme-module-ann.yang`*" to pyang and the `tailf` plugin would add the statement *`myex:expand-node`* to the processed module output. The complete `tailf` plugin is quite complex, but only small part of it is needed to achieve this functionality.

The code below shows a skeleton of a plugin that is able to process annotations of the form *myex:annotate <path>* exactly as the *tailf* plugin did. Note, again, that you need code like this only if you are already using another pyang plug-in which processes the aforementioned *myex* statements. The code below can be added to methods of that plug-in:

```
from pyang import plugin, statements
import optparse

class MyExPlugin(plugin.PyangPlugin):
    def __init__(self):
        super(MyExPlugin, self).__init__(‘myEx’)
        # initialize list of annotation modules
        self.myex_ann_mods = []
        # make sure pyang calls back after a module has been
        # almost processed
        statements.add_validation_phase(‘myex_ann’,
                                        before=’unused’)
        statements.add_validation_fun(‘myex_ann’,
                                       [‘module’],
                                       self.annotate)

    def add_opts(self, optparser):
        # the option for passing annotation modules
        # (instead of the old ‘-a’)
        opt = optparse.make_option(‘--myex-ann’,
                                   dest=’myex_ann’,
                                   default=[],
                                   action=’append’,
                                   help=’MyEx plugin annotations’)
        optparser.add_option(opt)

    def pre_load_modules(self, ctx):
        # make Pyang load and parse all myex annotation modules
        for modpath in ctx.opts.myex_ann:
            self.myex_ann_mods.append(modpath)
            with open(modpath) as moddata:
                ctx.add_module(modpath, moddata.read())

    def annotate(self, ctx, m):
        if m.pos.ref not in self.myex_ann_mods:
            return
        # for annotation module, take and expand all ‘annotate’
        # statements
        for annot in m.search((‘my-extension-module’, ‘annotate’)):
            node = statements.find_target_node(ctx, annot)
            node.substmts.extend(annot.substmts)
```

The validation phase added in the plug-in constructor makes sure that our function *annotate* is called after an annotation module has been almost completely processed by pyang, but before warnings about unused imported modules are generated. Note that in the example above, we are importing *acme-module*, but its only usage is in the annotate statement. By invoking the function *find _ target _ node*, we let pyang know about the usage.

With a plug-in like this, an annotation module may contain this stanza:

```
myex:annotate "/acme:c1/acme:c2" {
    myex:expand-node;
}
```

When you pass such module to pyang like:

```
$ pyang --myex-ann acme-module-ann.yang ...
```

The *emit* phase of your pyang plug-in would find the statement *myex:expand-node* under */c1/c2* in the module *acme-module*.

Note that there are drawbacks to this approach. The *tailf* plugin introduces several ConfD specific YANG modifications (most notably several custom XPath functions). So, a few modules may be refused by pyang without the plug-in or may behave differently.

3. Other Alternatives

3.1 Annotations Through Deviations

The plug-in skeleton described in the previous section makes it possible to add statements to another module. The same effect can be achieved by this sole construct:

```
deviation "/acme:c1/acme:c2" {
  deviate add {
    myex:expand-node;
  }
}
```

The YANG module with this snippet can then be passed to pyang as a *deviation module*. Note, however, that this construction is troublesome. The intended use of deviations is somewhat different and some YANG tools may refuse it. However, pyang accepts and processes it nonetheless.

3.2 Using yanger

The tool which replaces pyang in the ConfD distribution is *yanger* – a fast, extensible YANG validator. It comes with a set of plug-ins, including its own version of the `tailf` plugin. Even though the yanger plug-in API bears some resemblance to the pyang plug-in API, programming a yanger plugin is very different. In particular, because of programming language differences: yanger plugins need to be written in Erlang.

This means that if you are using pyang for a transformation implemented by an existing yanger plug-in, you should be able to switch to yanger without any issues. If you have your pyang plug-ins, rewriting them for yanger is not an easy task and yanger may not be an option for you.

4. For More Information

For more information about ConfD, visit https://www.tail-f.com

Open source pyang GitHub repository: https://github.com/mbj4668/pyang tail-f a Cisco company

www.tail-f.com
info@tail-f.com

Corporate Headquarters

Sveavagen 25
111 34 Stockholm
Sweden
+46 8 21 37 40