YencaP Documentation

Vincent Cridlig, Radu State

YencaP Documentation

Release: yencap-1.1.2

Release: yencapClient-1.1.2

Vincent CRIDLIG, cridligv@loria.fr
Radu STATE, state@loria.fr

Madynes Research team
LORIA-INRIA Lorraine
rue du jardin botanique
Villers-les-Nancy
FRANCE

Contents

List of Figures

1 Introduction

1.1 Information

1.1.1 YencaP's author information

Author names and organism attachment:
Vincent Cridlig, Jerome Bourdellon and Radu State
vincent.cridlig, jerome.bourdellon, radu.state@loria.fr
Madynes Research team
LORIA-INRIA Lorraine
rue du jardin botanique
Villers-les-Nancy
FRANCE

If you have any requests, comments, ideas for improvement or questions concerning YencaP, if you think you find a bug, please send emails to Vincent.Cridlig@loria.fr.

1.1.2 Software information

Software name: YencaP
Version: 1.1.2
Programming language: Python
Operating System: Linux (tested in Fedora core 3 and 4)
License: GNU Lesser General Public License (LGPL)

YencaP is an implementation of Netconf and follows the seventh version of Netconf draft. It implements all NetConf operations, #writable-running, #url, #startup, #candidate and #xpath capabilities. Get-config and get support both *subtree filtering* and *xpath* methods for selecting device data.

YencaP can run over SSH or over a customized XML security (see Figure 1). YencaPClient is a light client to connect to YencaP and can run interactively or not. It is used mostly for test purposes. For both of them, the default underlying protocol is SSH.

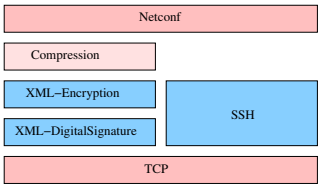

Figure 1: YencaP layers

2 Installation

2.1 Download

YencaP and YencapClient are available in our web page: download.

2.2 System requirements

YencaP requires the following libraries. Packages marked with a star (*) are available in the default package list of Fedora Core 4. Therefore, they can be installed easily using `yum install package-name` if not already installed:

python (python-2.4.1-2) *

python-devel (python-devel-2.4.1-2) *

xmlsec1 (xmlsec1-1.2.7-4) *

xmlsec1-devel (xmlsec1-devel-1.2.7-4) *

xmlsec1-openssl (xmlsec1-openssl-1.2.7-4) *

xmlsec1-openssl-devel (xmlsec1-openssl-devel-1.2.7-4) *

libxml2 (libxml2-2.6.19-1) *

libxml2-devel (libxml2-devel-2.6.19-1) *

libxml2-python (libxml2-python-2.6.19-1) *

PyXML (PyXML-0.8.4-3) *

4Suite (4Suite-1.0-8.b1) *

PyXMLSec 0.2.1 Please choose openSSL option during the installation.

paramiko 1.4 Provides SSH2 support.

pycrypto 2.0.1 Python Cryptography Toolkit

LTXML and PyLTXML-1.3-7-p332.i386.rpm

XSV-2.0-3.noarch.rpm

In order for YencaP to find the right libraries, please update your PYTHONPATH environment variable depending on the installation paths. As python2.4 is getting more and more used, every Python libraries should be somewhere in "/usr/lib/python2.4".

2.3 YencaP

You must have root privileges to perform the YencaP installation. Assume that *yencap-1.1.2.tar.gz* is stored in */tmp* directory. Go to */tmp* directory and unzip the compressed file. Change directory to *yencap-VERSION* and install the software. The *$YENCAP_HOME* environment variable must be set to */usr/local/yencap*:

```
% cd /tmp
% tar -xzvf yencap-{VERSION}.tar.gz
% cd yencap-{VERSION}
% su
% make uninstall (# if you had a previous installed yencap)
% make install
% export YENCAP_HOME="/usr/local/yencap"
% netconfd
```

To start, stop and restart YencaP as a deamon service, run the following.

```
% /etc/init.d/netconfd start    // To start
% /etc/init.d/netconfd stop     // To stop
% /etc/init.d/netconfd restart  // To restart
```

2.4 YencaPClient

You must have root privileges to perform the YencaPClient installation. Assume that yencapClient-VERSION.tar.gz is stored in */tmp* directory. The installation process is similar to YencaP installation:

```
% cd /tmp
% tar -xzvf yencapClient-{VERSION}.tar.gz
% cd yencapClient-{VERSION}
% su
% make uninstall (# if you had a previous installed yencapClient)
% make install
% export YENCAP_CLIENT_HOME="/usr/local/yencapClient"
```

YencapClient can run in interactive or non-interactive mode. The interactive mode allows to open a Netconf session and to send requests dynamically in this session. YencapClient querries the user to get the needed parameters and builds the Netconf requests dynamically from these user choices. The non-interactive mode allows to send one prepared Netconf request.

```
% cd /usr/local/yencapClient
#To use interactive mode, run:
% ./netconfc.py localhost
#To use non-interactive mode with a prepared request (get_config_subtree_ifeth0.xml), run:
% ./netconfc.py localhost tests/get_config_subtree_ifeth0.xml
```

3 User guide

3.1 YencaP

3.1.1 Netconf over SSH

There is two important files to make the agent work over SSH:

- */usr/local/yencap/conf/netconfd.xml*
- */usr/local/yencap/conf/sshUsers.xml*

First, *netconfd.xml* allows to choose the listenning port and the IP version (4 or 6). It also proposes some underlying protocols (SSH and XMLSec). In order to use SSH, set the *status* attribute to *active* and set the path of the private key file of the agent (YencaP).

Second, *sshUsers.xml* allows to configure some well known users that will be able to request the agent. A user can be authenticated with login password or login and public/private key. In order to authenticate users with public/private key, just set the text value of the *allowkeys* tag to the public key of the user. Public/private key is prioritary to password.

The agent is now ready to restart.

3.1.2 Netconf over XMLSec

If you run YencaP over TCP, it is possible to setup *compression* and *encryption* options. To enable these features, edit the *conf/netconfd.xml* file:

- To enable encryption, set the encryption node to 1,
- To disable encryption, set the encryption node to 0,
- To enable compression, set the compression node to 1,
- To disable compression, set the compression node to 0,

Role keys generation In order to generate a key for encryption, the head command line utility is used. The following command creates a key of 128 bits and stores the result in a file named *keyrole1*:

```
cd /usr/local/yencap/bin head -c 128 /dev/urandom > keyrole1
```

You can generate as much key as needed, depending on the number of roles (RBAC) you use. These files must be stored in the *keystore* directory of the Netconf agent. Alternatively, you can use the *keyGenerator.py* utility. In order to build a new role key, run the following command line in *bin* directory and follow the instructions:

```
% python keyGenerator.py
```

The generated key will be stored automatically in *keystore* directory. A Netconf manager must know that key so that the related role can be used. Therefore, once a key was generated, it must be distributed to *authorized* managers.

3.2 YencapClient

3.2.1 Netconf over SSH

One file must be configured to use YencaPClient over SSH:

- */usr/local/yencapClient/conf/yencapClient.xml*

yencapClient.xml allows to configure the server port and the IP version (4 or 6). As for YencaP, you can choose between several underlying transport applications. To select SSH, set the *status* attribute to *active* in the right *application-protocol* tag. Then set the users allowed to use YencaPClient along with a password and/or their private key. Finally, add some allowed agents (name, IP, ... and their public key).

The manager is now ready to send a request to the agent, for instance: `./netconfc.py localhost tests/get_config_subtree_ifeth0.xml`.

3.2.2 Interactive mode

The interactive mode prompts for a login. This login must be known by both YencaP and YencapClient. Depending on the options, the user will be prompt for a password. Once the user is logged in, YencapClient displays the agent's capabilities. Finally, YencapClient displays a menu that allows to choose between the possible Netconf operations. Depending on the operations, some submenus may appear to select different parameters. Figure 2 shows the main menu:

```
--------------------------
[0] get-config
[1] get
[2] edit-config
[3] copy-config
[4] delete-config
[5] lock
[6] unlock
[7] kill-session
[8] close-session
[9] get-modules
Choose Netconf operation:
```

Figure 2: YencaPClient menu

3.3 Simple example

To test the server, you can use the client that allows to open a Netconf session with the server:

```
% ./netconfc agentHostName tests/xpathGetRequest12.xml
```

xpathGetRequest12.xml embeds a Netconf request (get-config) using the subtree filtering method. The message-id attribute is modified by the client according to Netconf protocol rules.

```
<rpc message-id="102" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
      <running/>
    </source>
    <filter type="subtree">
      <netconf>
        <security>
          <rbac>
            <permissions>
              <permission>
                <scope/>
              </permission>
            </permissions>
          </rbac>
        </security>
      </netconf>
    </filter>
  </get-config>
</rpc>
```

Here is the reply for *xpathGetRequest12.xml*.

```
<rpc-reply message-id="1" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<?xml version="1.0" encoding="UTF-8"?>
<netconf>
  <security>
    <rbac>
      <permissions>
        <permission type="+" id="1" op="rw">
          <scope>/netconf/security</scope>
        </permission>
```

```
        <permission type="+" id="2" op="rw">
          <scope>/netconf/interfaces</scope>
        </permission>
        <permission type="+" id="3" op="rw">
          <scope>/netconf/routing/bgp</scope>
        </permission>
        <permission type="+" id="4" op="rw">
          <scope>/netconf/system</scope>
        </permission>
        <permission type="+" id="5" op="r">
          <scope>/netconf/log</scope>
        </permission>
        <permission type="+" id="6" op="rw">
          <scope>/netconf/tests</scope>
        </permission>
      </permissions>
    </rbac>
  </security>
</netconf>
</rpc-reply>
```

In order to get the log of the agent using the xpath capabilities, please try *xpathGetRequest4.xml*

```
<rpc message-id="104" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
      <running/>
    </source>
    <filter type="xpath">
      /netconf/log
    </filter>
  </get-config>
</rpc>
```

The reply will look like this XML document:

```
<rpc-reply message-id="1" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<netconf>
  <log>
    <nbOp>
                <get>0</get>
                <get-config>601</get-config>
                <edit-config>72</edit-config>
                <copy-config>0</copy-config>
                <delete-config>1</delete-config>
                <close-session>429</close-session>
        </nbOp>
    <sessions>
                <number>249</number>
                <average-time>0.91176626481</average-time>
        </sessions>
    <requests>
                <number>672</number>
                <average-time>0.267890569</average-time>
        </requests>
  </log>
</netconf>
</rpc-reply>
```

Many test requests (*get-config*, *copy-config*, *edit-config*) are located in */usr/local/yencapClient/tests*.

4 YencaP Netconf agent architecture

Figure 3 illustrates the architecture of the YencaP NetConf agent. The agent is organized into several functional blocks. The *Socket layer* is in charge of assembling the incoming Netconf messages and also sending prepared messages. The *RPC layer* checks the `<rpc>` level, optionally compress/decompress and encrypt/decrypt the messages. The *Request layer* function is to extract the Netconf operation embedded in the `<rpc>` node. Then it calls the related method (get-config, edit-config, ...) from the *Dispatcher* block.

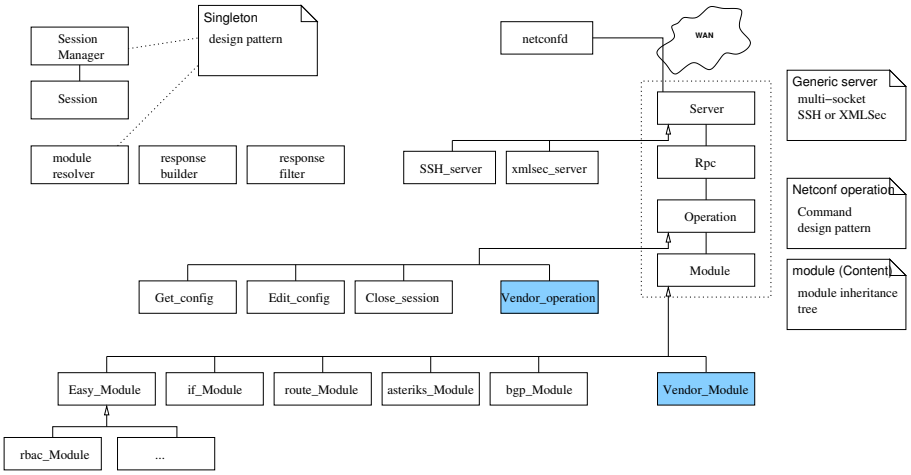

Figure 3: Netconf agent architecture

The *Dispatcher* block is responsible for performing the requested operation. It uses three functionnal blocks that are called serially. The *Modules Resolver* is able to retrieve the list of modules that can handle the Netconf operation. A *Module Register Table*, storing XPath expressions related to each module, is parsed to bind the data received to the right XPath(). A modules list is returned to the *Dispatcher*. All these modules are querried by the *XML Response Builder* to perform the operation. In the case of a *get-config* and if security is activated, the *XML Response Filter* removes all the data that the role is not allowed to read.

Figure 4 shows the home directory and sub-directories along with their functional roles.

ModuleResolver is in charge of loading the different YencaP modules. It also finds the right modules when YencaP receives a Netconf request. Sometimes, ModuleResolver is not able to find the corresponding modules. This happens when a XPath request without absolute path (//user[name='foo']) is processed. In these cases, all YencaP modules are querried by default. If you want the best performances, you may prefer to send absolute XPath requests so that ModuleResolver only querries the corresponding modules.

When Dispatcher querries a set of modules, one of them may reply with an error. In that case, the request processing is stopped and a rpc-error is sent back to the manager, even if the other modules did not go to an error.

```
+ server
  - README
  - TODO
  - netconfd.py           // YencaP daemon (executable file)
  - rbacManager.py        // Manages access control using rbac.xml policy
  - constants.py          // Netconf protocol constants
  - logger.py             // Logs YencaP statistics
  - util.py               // Helps for (de)compression, XML/String convertion
  - xmlResponseBuilder.py // Builds XML response from modules replies
  - moduleResolver.py     // Finds competent modules
  - xmlResponseFilter.py  // Filters XML responses (XPath request and RBAC)
  - sessionManager.py     // Manages Netconf sessions
  - __init__.py           // Defines a new package
  - session.py            // A Netconf session
  + conf
    - candidate.xml       // Netconf candidate configuration file
    - encrypt-tmp.xml     // template to produce encrypted documents
    - log.xml             // log file
    - netconfSchema.xsd   // Netconf protocol schema
    - rbac.xml            // XML access control policy (RBAC model)
    - hello.xml           // Netconf hello message containing capabilities
    - modules.xml         // modules registration file
    - operations.xml         // operations registration file
  + Server                    // Servers directory
    - server.py               // generic server
    - serverSSH.py            // SSH server
    - serverXMLSec.py         // XMLSec server
    - sshModule.py            // SSH authentication module
    - crypto.py               // XML cryptography module
  + Operations                // operationss directory
    + base
      - get_config_operation.py    // get-config
      - get_operation.py           // get
      - copy_config_operation.py   // copy-config
      - delete_config_operation.py // delete-config
      - edit_config_operation.py   // edit-config
      - lock_operation.py          // lock
      - unlock_operation.py        // unlock
      - close_session_operation.py // close-session
      - kill_session_operation.py  // kill-session
    + ext
  + Modules               // modules directory
    - module.py           // generic module implementation (copy-config)
    - easyModule.py       // generic module implementation (edit-config)
    - moduleReply.py      // interface for modules replies
    - meta.xsl            // XSL stylesheet for XSL edit-config generation
    + RBAC_Module         // RBAC module
      - __init__.py       // package definition
      - rbacModule.py     // main RBAC module file
    + System_Module       // module for system data
    + Test_Module         // test module
  + keystore              // stores role keys
    - systemAdmin         // key for systemAdmin role
    - networkAdmin        // key for networkAdmin role
  + docs
    - documentation.pdf   // this documentation
```

Figure 4: YencaP Netconf agent home directory

5 Developer's guide for extensions

5.1 Introduction

The YencaP architecture is designed to be extensible. It makes it easier for vendors to both support a larger data model and develop some new operations. These extensions don't require to modify the core agent. It means that vendors will be able to migrate to future YencaP implementation updates without problems. Also YencaP allows to use different application protocols like SSH, BEEP, SOAP. Only SSH is implemented but the architecture is ready to support these protocols.

- To allow data model extension, YencaP is based on a *module* system. The modules described in an configuration file are loaded dynamically. Each module is responsible for a part of the whole data model. Although YencaP delegates the implementation of Netconf operations to the modules, it provides a programming interface for easy integration in YencaP. Core YencaP also provides an interface for standard modules replies (ModuleReply).

- To allow new operations, YencaP uses a *command design pattern*. In this design pattern, a Netconf operation is defined as a class and inherits from a general *command* class (e.g. Get_config_operation class inherits from Operation class). The advantage is that a new operation can be added simply by coding a new class. As for modules, a configuration file describes the available operations, thus avoiding to update the code of core YencaP. Core YencaP also provides an interface for standard operation replies (OperationReply).

- To allow new application protocols, YencaP defines a generic server. It can be inherited to support for instance BEEP or SOAP. A SSH_Server and XMLSec_Server are already implemented. A configuration file describes which one must be used.

5.2 Extending the data model

5.2.1 Introduction

In order to develop and integrate a new module in the YencaP software, some guidelines have to be followed. Each module implements a feature (service configuration, service monitoring) and will be in charge of performing the Netconf requests related to its data. The whole agent configuration abstract view is an XML tree. A module is responsible for a subtree, which can be a subtree of another module. For instance, if moduleA is in charge of a subtree whose root node is /foo/bar, moduleB can be in charge of a subtree whose root node is /foo/bar/fooB. It is also possible to define two modules as following: ModuleA is in charge of /foo/bar[@name='A'] and ModuleB is in charge of /foo/bar[@name='B'], thus allowing different equivalent vendors to distinguish their own subtree.

In the code, python modules are dynamically imported, meaning that nothing has to be updated in the core YencaP Agent.

5.2.2 Code Auto-generation

You can generate a new YencaP module with *moduleGenerator* Python module. The command line is the following:

```
% python moduleGenerator.py moduleName InheritedModuleName Xpath
```

First parameter, *moduleName*, is the name of the module to be created. Second parameter, *InheritedModuleName*, is the parent inherited module. It must be one of *Module* or *EasyModule*. Third parameter, Xpath, is the XPath expression giving the XML node where the configuration data must be appended in the whole configuration tree. The *modules.xml* file is automatically updated and the directory for the new module is automatically created along with two files: *__init__.py* to build a new package and the main class of the new YencaP module.

5.2.3 Module registration

The conf/modules.xml file describes the list of module that are loaded by the agent. It means that, if a 'Quagga' module is defined in this module, netconfd will try to find the 'Quagga' directory in the 'Module' directory. Then netconfd will build an instance of this module dynamically.

In Figure 5, the module name is 'Quagga'. This name must be the same as its directory. The 'mainFileName' is the main file of the module and implements the Netconf operations: 'get-config', 'edit-config', 'copy-config', ...

```
<module>
 <name>Quagga</name>
 <mainFileName>BGP_Module</mainFileName>
 <className>BGP_Module</className>
 <xpath>/netconf/routing/quagga</xpath>
 <xsdfile>bgp.xsd</xsdfile>
 <parameters>
  <host>127.0.0.1</host>
  <port>2605</port>
  <password>a</password>
  <enable_pass></enable_pass>
 </parameters>
</module>
```

Figure 5: BGP Module Registration

The className is the Class of the object that will be instantiated. This object is then stored in a list of Netconf modules. This class must inherit from the "Module" class defined in module.py.

The xpath element defines the element the module is responsible for in the whole configuration. It means that the module specific configuration will be a subtree of the whole configuration. The root of this subtree is the node given by the xpath expression. Only one node of the whole configuration must match this xpath expression.

The "parameters" element describes the parameters needed by the module when being instanciated. This set of parameters is given to the __init__ method as a dictionnary.

5.2.4 Implementing the main class

The code of your module must be located in the directory you created within Modules directory. There are two ways to implement the main class of a new module:

1. The main class of the new module can inherit from *Module* class. In that case, one has to implement at least *get-config* and *edit-config*. *copy-config* is generic and is a particular case of *edit-config*
2. The main class of the new module can inherit from *EasyModule* class. In that case, one has to implement at least *get-config* and *copy-config*. *edit-config* is already implemented and relies on *get-config* and *copy-config*.

In both cases, you can override the Netconf methods.

Inheriting Module Your main class must inherit from the Module class. The "Module" class implements the copy-config method since it can be implemented on the base of "edit-config" with special parameters. Therefore, a developer is free to override it or not in its Netconf module. Typically, a developer will only implement *get-config* and *edit-config*.

Figure 6 illustrates a module *SampleModule* inheriting from *Module* and implementing *get-config* and *edit-config*. The *dictionnary* parameter contains the parameters of the module entry in the modules.conf file.

```
from Modules.module import Module

class SampleModule(Module):
 def __init__(self, dictionnary):
  ...

 def get-config(self):
  ...

 def edit-config(self):
  ...
```

Figure 6: Sample module implementation

Inheriting EasyModule In the last version of the agent, the module developper task is simplified to the implementation of *get-config* and *copy-config*. Experience with Netconf showed that, in most cases, *copy-config* implementation is easier than the *edit-config* one.

Therefore, a generic *edit-config* is now implemented in the super class. Inheriting from *EasyModule* is particularly interesting when the configuration to be managed by the module is a XML file. In that case, adding a new module consists in writing 4 lines. Figures 7 and 8 illustrate the implementation of the RBAC module and the Log module.

If the data you want to manage within hte module is not simply a XML file, you will have to implement *get-config* and *copy-config*. By default, EasyModule considers that a XML file contains the managed data.

```
from Modules.easyModule import EasyModule

class RBACModule(EasyModule):

 def __init__(self, arguments):
  self.dataFile = "conf/rbac.xml"
```

Figure 7: RBAC module implementation

```
from Modules.easyModule import EasyModule

class Log_Module(EasyModule):

 def __init__(self, arguments):
  self.dataFile = "conf/log.xml"
```

Figure 8: Log module implementation

Figure 9 illustrates the approach followed for the *edit-config* netconf operation. The idea is that each *edit-config* is translated to an equivalent xsl document. A meta stylesheet can parse the *edit-config* and generate the related XSL document. Then the generated xsl stylesheet is applied to the device configuration. The latter is retrieved with a local get-config. Then a call to the copy-config is performed to setup the new configuration.

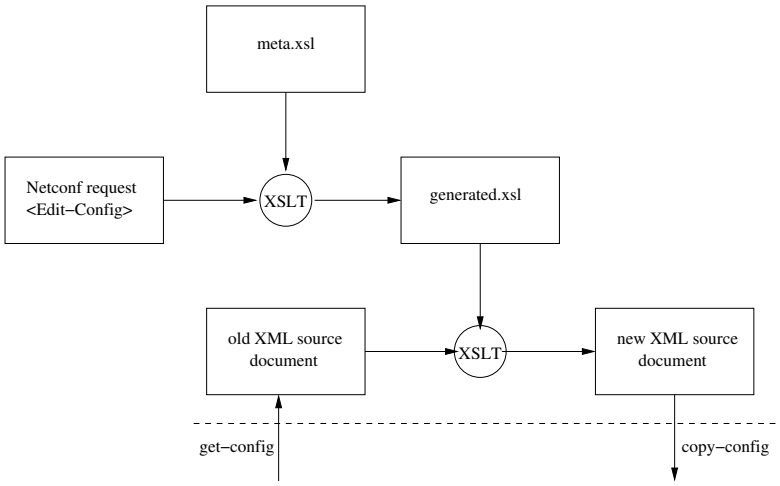

Figure 9: Edit-config

In the current version, the *xc:operation* must be present in the edit-config.

5.2.5 Returning the Reply to Core YencaP

A generic class *ModuleReply* is provided in order to help module developer to return the module reply to the core agent. A ModuleReply can embed an error, an ok reply or some XML data. Here is an example code to be used in the case of :

- An error:

```
moduleReply = ModuleReply(
error_type=ModuleReply.APPLICATION,
error_tag=ModuleReply.OPERATION_FAILED,
error_severity=ModuleReply.ERROR,
error_message="An helpful error message"
return moduleReply
```

- A success:

```
modulereply = ModuleReply()
return modulereply
```

- A get* request:

```
modulereply = ModuleReply(replynode=xmlnode)
return modulereply
```

5.3 Extending the capabilities (operations)

5.3.1 Creating a new operation

Let's create an new operation called My_operation. The first step is to create a file my_operation.py in *yencap/Operations/ext* directory (*yencap/Operations/base* directory is dedicated to the base Netconf operations). Our convention is use the pattern Aaaaa_operation for the class name and aaaaa_operation.py for the python file name. For multiple words operation, use underscore like in Get_config_operation.

My_operation must inherit from Operation class. It also must implement the constructor __init__() without parameters, the setParameters() and execute() methods. __init__() is supposed to instantiate the "my" operation parameters with the default values. setParameters() parses the XML operation node, extracts the parameters and checks the validity of the request. execute() the operation.

Three main attributes are inherited from Operation class:

- self.operation: the XML node containing the operation:

```
<my>
        <param1>v1</param1>
        <param2>v2</param2>
</my>
```

- self.operationReply: The object that must be replied by the execute method to the rpc layer.

- self.session: the Netconf session of the current manager.

```
from Operations.operation import Operation
from Operations.operationReply import OperationReply
from xml.dom import Node

class My_operation(Operation):
        """
            Concrete command (see command design pattern) for "my" operation.
        """

        def __init__(self):
                """
                        Constructor of a My_operation command.
                """
                # Initializes the default value for filtering method

        def setParameters(self):
                """
                        Parse the XML operation and get the fields
                """
                # python code
                # get value v1 from parameter param1
                # get value v2 from parameter param2

        def execute(self):
                """
                        Execute the get-config operation.
                """
                # python code
                # Use v1 and v2 to perform the operation.
                self.operationReply.setNode(xmlreplynode)
                return self.operationReply
```

Figure 10: Sample operation implementation

5.3.2 Connecting the operation to YencaP

In order to activate this new operation, what must be done is to update the yencap/conf/operations.xml file with the following XML node:

```
<operation>
        <name>my</name>
        <mainFileName>ext.my_operation</mainFileName>
        <className>My_operation</className>
</operation>
```

Figure 11: Update to operations.xml file

5.4 Implementing a new application protocol (BEEP, SOAP, ...)

Still in development.

TODO: mv netconfd.xml to servers.xml (like operations.xml or modules.xml)

6 Appendix

6.1 Module

```
import string
from modulereply import ModuleReply

class Module:
 """
  This class should be inherited by each module that want
  to be add to the Agent.
  Its specified the main functions needed for the Agent to
  follow the NETCONF protocol.
 """

 # CONFIGURATION DATASTORES
 RUNNING_TARGET = "running"
 CANDIDATE_TARGET = "candidate"
 STARTUP_TARGET = "startup"

 # ERROR OPTION
 ROLL_BACK_ON_ERROR = "rollback-on-error"
 STOP_ON_ERROR   = "stop-on-error"
 IGNORE_ERROR   = "ignore-error"

 # TEST OPTION
 SET = "set"
 TEST_AND_SET = "test-and-set"

 # OPERATION ATTRIBUTE TAG
 OPERATION_TAG = "operation"
 XPATH_TAG = "xpath"
 CONFIG_URI = "urn:ietf:params:xml:ns:netconf:base:1.0"

 # OPERATION VALUES
 CREATE_OPERATION  = "create"
```

```
MERGE_OPERATION  = "merge"
REPLACE_OPERATION  = "replace"
DELETE_OPERATION = "delete"
NONE_OPERATION  = "none"

def __init__(self, arguments):
 """
  This method should be overrided by the module.
  @type arguments : dictionary,
  @param arguments : data specified in modules.xml for module creation.
 """
 pass

def close(self):
 """
  Should be overrided if the module needs to close some resurces,i.e. sockets.
 """
 pass

def setAttributes(self, name, path, fileName):
 """
  It is used by the Agent to manipulate the modules.It shouldn't be overrided.
  @type  name: string
  @param name: name of the module
  @type  path: string
  @param path: xpath specifying the node for the module.
  @type  fileName: string
  @param fileName: Main file name of the Module
 """
 self.name = name
 self.path = path
 self.fileName = fileName


def get(self):
 """
  Generate the device's XML state data if any of the current module.
  @rtype: ModuleReply
  @return: It should return the device's configuration or an error
  ** Relates to the netconf get-config operation
 """
 xmlreply = ModuleReply(
 error_type=ModuleReply.APPLICATION,
 error_tag=ModuleReply.OPERATION_NOT_SUPPORTED,
 error_severity=ModuleReply.ERROR,
 error_message="OPERATION-NOT-SUPPORTED")
 return xmlreply


def getConfig(self):
 """
  Generate the device's XML configuration of the current module.
  @rtype: ModuleReply
  @return: It should return the device's configuration or an error
  ** Relates to the netconf get-config operation
 """
 xmlreply = ModuleReply(
 error_type=ModuleReply.APPLICATION,
 error_tag=ModuleReply.OPERATION_NOT_SUPPORTED,
 error_severity=ModuleReply.ERROR,
 error_message="OPERATION-NOT-SUPPORTED")
 return xmlreply
```

```
def copyConfig(self, sourceNode, targetName, urlValue=None):
 """
  Copy the sourceNode's XML configuration of the current module
  to the targenName.
  @type targetName: string
  @param targetName: Is always the "running" configuration datastore because
        "startup", "candidate") is managed by the core Netconf module
  @rtype: ModuleReply
  @return: It should return the device's configuration or an error
  ** Relates to the netconf copy-config operation
 """
 if (targetName == self.RUNNING_TARGET):
  return self.editConfig(self.REPLACE_OPERATION, self.SET, self.STOP_ON_ERROR,
  self.RUNNING_TARGET, sourceNode)

 if (targetName == self.CANDIDATE_TARGET or targetName == self.STARTUP_TARGET):
  # Replace the candidate xml file with the docsource
  f = open("conf/" + targetName + ".xml",'w')
  PrettyPrint(sourceNode, f)
  f.close()
  return ModuleReply()

 if (targetName == "url" and urlValue != None):
  xmlreply = ModuleReply(error_type=ModuleReply.APPLICATION,
  error_tag=ModuleReply.OPERATION_NOT_SUPPORTED,
  error_severity=ModuleReply.ERROR,
  error_message="OPERATION-NOT-SUPPORTED")
  return xmlreply

def editConfig(self, defaultoperation, testoption, erroroption, target,
 confignode, targetnode=None):
 """
  Apply the request specified in configNode to the targetnode.
  @type defaultoperation: MERGE_OPERATION | REPLACE_OPERATION | NONE_OPERATION
  @param defaultoperation : as specified in NETCONF protocol
  @type testoption : SET | TEST_AND_SET
  @param testoption : as specified in NETCONF protocol
  @type erroroption : STOP_ON_ERROR | IGNORE_ERROR | ROLL_BACK_ON_ERROR
  @param erroroption : as specified in NETCONF protocol
  @type target : RUNNING_TARGET | CANDIDATE_TARGET | STARTUP_TARGET
  @param target : as specified in NETCONF protocol
  @type targetnode : string
  @param targetnode : if the target is RUNNING_TARGET or STARTUP_TARGET
       it will be ignored otherwise should be the node of the CANDIDATE_TARGET
       that this module should procees
  @rtype: ModuleReply
  @return: It should return a success or error message.
  ** Relates to the netconf edit-config operation
 """
 xmlreply = ModuleReply(error_type=ModuleReply.APPLICATION,
 error_tag=ModuleReply.OPERATION_NOT_SUPPORTED,
 error_severity=ModuleReply.ERROR,
 error_message="OPERATION-NOT-SUPPORTED")
 return xmlreply

def rollBack(self):
 """
  It restablish the last state of the device's configuration. Note that the
  last editConfig should had received erroroption equal to
  ROLL_BACK_ON_ERROR in order to achived this request,
```

```
   otherwise returns in error.
   @rtype: ModuleReply
   @return: It should return the device's configuration or an error
  """
  xmlreply = ModuleReply(error_type=ModuleReply.APPLICATION,
  error_tag=ModuleReply.OPERATION_NOT_SUPPORTED,
  error_severity=ModuleReply.ERROR,
  error_message="OPERATION-NOT-SUPPORTED")
  return xmlreply

 def manage(self, xpath=None):
  """
   Shouldn't be overrided, it is used by the Netconf Agent.
   @rtype : boolean
   @return: A boolean value if this module is responsible for the request
   xpath expression
  """
  tab1 = xpath.split('/')
  tab2 = self.path.split('/')
  i=1
  while i<len(tab1) and i<len(tab2) and tab1[i]==tab2[i] :
   i=i+1
  return i==(len(tab2)) or i==(len(tab1))
```

6.2 EasyModule

```
from Modules.module import Module
from Modules.modulereply import ModuleReply
from Ft.Xml.Domlette import NonvalidatingReader, PrettyPrint
from Ft.Xml import XPath, InputSource
from Ft.Xml.Xslt import Processor, DomWriter
import util

metaFile = "Modules/meta.xsl"


class EasyModule(Module):

 def __init__(self, arguments):
  pass


 # This method must return the root node
 # corresponding to the current module
 def getConfig(self):
  # This is for test purpose
  doc = NonvalidatingReader.parseUri("file:"+self.dataFile)
  xmlreply = ModuleReply(replynode=doc.documentElement)
  return xmlreply

 # This method must return the root node
 # corresponding to the current module
 def copyConfig(self, sourceNode, targetName, urlValue=None):
  util.printNodeToFile(sourceNode, self.dataFile)
  moduleReply = ModuleReply()
  return moduleReply


 def editConfig(self,defaultoperation,testoption,erroroption,target,configNode,targetNode=None):
  """
```

```
 Apply a BGP request from the confignode to the targetnode.
 @type defaultoperation: MERGE_OPERATION | REPLACE_OPERATION | NONE_OPERATION
 @param defaultoperation : as specified in NETCONF protocol
 @type testoption : SET | TEST_AND_SET
 @param testoption : as specified in NETCONF protocol
 @type erroroption : STOP_ON_ERROR | IGNORE_ERROR | ROLL_BACK_ON_ERROR
 @param erroroption : as specified in NETCONF protocol
 @type target : RUNNING_TARGET | CANDIDATE_TARGET | STARTUP_TARGET
 @param target : as specified in NETCONF protocol
 @type targetnode : string
 @param targetnode : if the target is RUNNING_TARGET or STARTUP_TARGET it will be ignored otherwise should be the
 @rtype: ModuleReply
 @return: It returns a success or error message.
 ** Relates to the netconf edit-config operation
 """

 try:
  # Generate a stylesheet equivalent to the edit-config
  df = InputSource.DefaultFactory
  editXMLRequest = df.fromString(util.convertNodeToString(confignode), 'urn:dummy')
  stylesheet = df.fromUri("file:"+metaFile, 'urn:sty')
  p = Processor.Processor()
  p.appendStylesheet(stylesheet)
  wr = DomWriter.DomWriter()
  p.run(editXMLRequest, writer=wr)
  generatedStyleSheet = wr.getResult()

  # Apply the generated stylesheet to the source document
  inputStyleSheet = df.fromString(util.convertNodeToString(generatedStyleSheet), 'urn:sty')
  oldXMLDocument = self.getConfig().getXMLNodeReply()
  inputDocument = df.fromString(util.convertNodeToString(oldXMLDocument), 'urn:dummy')
  p = Processor.Processor()
  p.appendStylesheet(inputStyleSheet)
  wr = DomWriter.DomWriter()
  p.run(inputDocument, writer=wr)
  newXMLDoc = wr.getResult()

  # Copy the new document over the old one
  xmlReply = self.copyConfig(newXMLDoc, target)
  return xmlReply

 except Exception,exp:
  print str(exp)
  moduleReply = ModuleReply(
  error_type=ModuleReply.APPLICATION,
  error_tag=ModuleReply.OPERATION_FAILED,
  error_severity=ModuleReply.ERROR,
  error_message=str(exp))
  return moduleReply
```

6.3 ModuleReply

```
from Ft.Xml.Domlette import NonvalidatingReader, implementation
from Ft.Xml import EMPTY_NAMESPACE

class ModuleReply:
 """
 This class is used as a container of data for every module reply to the Agent.
 """
```

```
### Tag ###

IN_USE     = "IN_USE"
INVALID_VALUE   = "INVALID_VALUE"
TOO_BIG     = "TOO_BIG"
MISSING_ATTRIBUTE   = "MISSING_ATTRIBUTE"
BAD_ATTRIBUTE    = "BAD_ATTRIBUTE"
UNKNOWN_ATTRIBUTE  = "UNKNOWN_ATTRIBUTE"
MISSING_ELEMENT   = "MISSING_ELEMENT"
BAD_ELEMENT    = "BAD_ELEMENT"
UNKNOWN_ELEMENT   = "UNKNOWN_ELEMENT"
ACCESS_DENIED   = "ACCESS_DENIED"
LOCK_DENIED    = "LOCK_DENIED"
RESOURCE_DENIED   = "RESOURCE_DENIED"
ROLLBACK_FAILED   = "ROLLBACK_FAILED"
DATA_EXISTS    = "DATA_EXISTS"
DATA_MISSING   = "DATA_MISSING"
OPERATION_NOT_SUPPORTED = "OPERATION_NOT_SUPPORTED"
OPERATION_FAILED  = "OPERATION_FAILED"
PARTIAL_OPERATION  = "PARTIAL_OPERATION"

###  Error-type ###

TRANSPORT     = "transport"
RPC      = "rpc"
PROTOCOL    = "protocol"
APPLICATION    = "application"

###  Severity  ###

ERROR     = "error"
WARNNING    = "warnning"

### Error-info ###

BAD_ATTRIBUTE_INFO  = "bad-attribute"
BAD_ELEMENT_INFO  = "bad-element"
SESSION_ID_INFO   = "session-id"
OK_ELEMENT_INFO   = "ok-element"
ERR_ELEMENT_INFO  = "err-element"
NOOP_ELEMENT_INFO  = "noop-element"

def __init__(self,replynode=None ,error_type=None, error_tag=None,
error_severity=None, error_tag_app= None, error_path= None, error_message=None):
 """
 It creates an Struture that will be used in every function that a module has been called by the Agent
 If the request was succesfull it should create an instance with any parameters
 If the request produced an XML node that should be included in the reply the instance should be created just with
 Otherwise, it shoud specify the error according to the NETCONF Protocol
 """

 self.replynode = replynode
 self.error_type = error_type
 self.error_tag = error_tag
 self.error_severity = error_severity
 self.error_tag_app = error_tag_app
 self.error_path = error_path
 self.error_message = error_message
 self.error_info = []

def setReplyNode(self,xmlnode):
 self.replynode = xmlnode
```

```
def setErrorType(self,error_type):
 self.error_type = error_type

def setErrorTag(self,error_tag):
 self.error_tag = error_tag

def setSeverityError(self,error_severity):
 self.error_severity = error_severity

def setErrorAppTag(self,error_tag_app):
 self.error_tag_app = error_tag_app

def setErrorPath(self,error_path):
 self.error_path = error_path

def setErrorMessage(self,error_message):
 self.error_message = error_message

def addErrorInfo(self,error_info_tag,error_message):
 self.error_info.append((error_info_tag,error_message))

def isError(self):
 return self.error_type != None

def createNode(self,doc,parent,tag,value=None):
 element = doc.createElementNS(EMPTY_NAMESPACE,tag)
 parent.appendChild(element)
 if (value != None):
  text = doc.createTextNode(value)
  element.appendChild(text)
 return element

def getXMLNodeReply(self):
 """
  Generate the XML node with the contents of the instance.
  @rtype: cDomlette
  @return: It is an XML node.
 """
 if (self.isError()):
  doc = implementation.createDocument(EMPTY_NAMESPACE, None, None)

  rpcerrornode = self.createNode(doc=doc, parent=doc,tag="rpc-error",value=None)
  self.createNode(doc=doc,parent=rpcerrornode,tag="error-type",value=self.error_type)
  self.createNode(doc=doc,parent=rpcerrornode,tag="error-tag",value=self.error_tag)
  self.createNode(doc=doc,parent=rpcerrornode,tag="error-severity",value=self.error_severity)
  if ( self.error_tag_app != None):
   self.createNode(doc=doc,parent=rpcerrornode,tag="error-app-tag",value=self.error_tag_app)
  if ( self.error_path != None):
   self.createNode(doc=doc,parent=rpcerrornode,tag="error-path",value=self.error_path)
  if ( self.error_message != None):
   self.createNode(doc=doc,parent=rpcerrornode,tag="error-message",value=self.error_message)
  if ( self.error_info != {}):
   infonode = self.createNode(doc=doc,parent=rpcerrornode,tag="error-info",value=None)
   for key,item in self.error_info:
    self.createNode(doc=doc,parent=infonode,tag=key,value=item)
  self.replynode = doc.documentElement
 elif (self.replynode == None):
  doc = implementation.createDocument(EMPTY_NAMESPACE, None, None)
  self.createNode(doc=doc,parent=doc,tag="ok",value=None)
  self.replynode = doc.documentElement
```

```
return self.replynode
```