NETCONF, speaking the network language

Leonidas Poulopoulos
(leopoul@noc.grnet.gr)
Software Development

Yannis Mitsos
(ymitos@noc.grnet.gr)
NOC Head

Greek Research & Technology Network (GRNET) NOC
56, Mesogeion Ave.
11527 Athens, Greece

Paper Type: Technical

Abstract

Traditional management protocols seem to struggle when it comes to modern networks management. Services on demand and advanced operations require accurate and rapid device configuration. Where SNMP seems inadequate, NETCONF comes to fill the gap. NETCONF introduces a protocol for XML-oriented communication with the device. Developing a network services provisioning application with NETCONF invokes XML parsing, building, searching and expression matching. As XML manipulation may impose a burden to the actual development of applications, in this paper we present an XML-to-Python objects proxy that allows for efficient communication with NETCONF enabled devices. Our proxy offloads the development process from all the XML based operations thus targeting effective and rapid application development. It is important to mention that, having released the proxy as open source software, the community can contribute and build network management tools and service provisioning platforms on top of it.

Keywords

NETCONF, XML, proxy, Python, objects

1. Introduction

As networks grow in size and complexity, the need to effectively manage a vast number of resources is of paramount importance. Traditional management protocols such as the Simple Network Management Protocol – SNMP - (William Stallings, 1999), (SNMP RFCs) are widely deployed and incorporated in network management tools and platforms. SNMP is a legacy protocol which has been widely deployed and is in use for several decades. It is based on a request-response messaging protocol using a connectionless transport, generally the User Datagram Protocol – UDP. SNMP excels at retrieving data especially for monitoring purposes. It is widely used in inventory building tools and network discovery platforms. SNMP structure allows for rapid retrieval of data along with fairly easy development of tools.

When it comes to service management and especially to automated service provisioning, SNMP is often inadequate. SNMP’s configuration-write (snmp-set) functions are not being fully utilized by many vendors, partly because of a lack of security in SNMP versions before SNMPv3 and partly because many devices simply are not capable of being configured via individual MIB objects. Network administrators often avoid using snmp-set for advanced service provisioning, eg. creation of VPNs. During, the last few years a new network management protocol arose, NETCONF (NETCONF WG), (NETCONF Wikipedia), allowing for provisioning of advanced network services. NETCONF is based on an XML messaging protocol using a connection-oriented transport, generally the Transmission Control Protocol. As NETCONF became a standard and gained reputation, GRNET Network Operations Center (GRNET NOC) has steered towards the NETCONF path. Pioneering, GRNET NOC has developed a novel NETCONF device proxy that can form the basis for developing advanced network management services.

GRNET (Greek Research and Technology Network) provides Internet connectivity and services to the Greek Universities and academic and research institutes. GRNET maintains points-of-presence in all major Greek cities (approximately 40) and leases dark-fiber across the country for its backbone and access network.

2. NETCONF as a management protocol

NETCONF does not aim to be an SNMP replacement. One of NETCONF’s goals was to overcome limitations encountered with SNMP as an efficient and effective means for performing many network management functions. From a technical point of view NETCONF is operating over a secure transport layer, such as SSH v2. Thus, compared to SNMP, NETCONF will require greater processing overhead on the network, especially for a lower number of managed objects (Hedstrom et al. 2011). On the other hand, NETCONF is very close to device configuration thus the knowledge of a device's command set can aid in building NETCONF expressions. As NETCONF allows for retrieval and application of configuration, it proves to be a robust choice when it comes to building a network management system that would involve applying configuration to devices rather than retrieving it. That is the reason why the majority of the latest proprietary Network Management Systems – NMSes - developed by big network equipment vendors seem to deploy NETCONF under the hood.
GRNET’s network is a mixed-vendor environment with a large number of devices supporting NETCONF. Until now, network configuration, day-to-day operations and services were fulfilled mostly using cli scripting. From the aspect of network management and monitoring, only network data retrieval was accomplished via SNMP, especially for graphing purposes (GRNET mon platform) and monitoring-alerting. But as our client demands grew and new network services were designed, a trend to automate network configuration emerged.

In theory, NETCONF communication with NETCONF-enabled devices is pretty straightforward. An SSH client connects to a device's NETCONF subchannel and commands are sent XML-RPC formatted. XML-RPC commands usually resemble cli commands. The really interesting and challenging part is when attempting to design and develop a platform based on NETCONF. Two major software components are required in order to develop a NETCONF-based Network Management and Service Provisioning System; a NETCONF client that will provide the communication channel between the device and the system and a network device proxy on top of it that will generate the device configuration. Depending on the programming language, a NETCONF client can be built from scratch or obtained from existing implementations (NETCONF Implementations) many of which are open source.

GRNET Network Operation Center, towards developing a NETCONF Network Management and Service Provisioning System, designed and implemented an in-house network device proxy module using open-source tools and platforms. We have to underline that there were no open-source modular implementations of NETCONF device proxies available, which led us to building our own from scratch.

2.1 Problems with NETCONF/XML in application development

As it is already mentioned, our scope was to develop an advanced services provisioning platform with open source tools and open source code. Having steered our project development towards Python during the last 4 years we went for a Python solution. We chose ncclient (ncclient project) to act as our northbound interface, the NETCONF client. Ncclient provides a Python NETCONF API for scripting and application development. It supports all (NETCONF RFC6241) operations and capabilities and its source code allows for further development and enhancements.

Several tests were conducted to verify that ncclient meets our requirements and is compatible with our devices’ NETCONF implementation. An initial test verified that the pure ssh communication with the device NETCONF subchannel was successful as shown in Figure 1.

Figure 1. NETCONF get-config via ssh

The equivalent test using ncclient is depicted in Figure 2.

Figure 2. NETCONF get-config via Python ncclient
For both, the pure NETCONF session and the ncclient, sending configuration requests and receiving configuration parts is actually transported using XML. The design and development of a platform with ncclient requires a module in the application-platform that converts service requests to XML configuration. Such an approach is depicted in Figure 3.

Figure 3. Developing applications with ncclient

In Figure 3, configuration requests from the network services application are sent XML-formatted towards the ncclient module. Translating service requests to XML configuration can become a hard task, especially when there is a great level of detail in the service request. Even if this issue is solved, another one comes up. Usually, for reconciliation and integration purposes, the retrieval of configuration or part of it is required. In such cases, a get-configuration request is sent and the response from the network device is received via the ncclient as an XML formatted document. The XML document has to be parsed with an XML parsing library, identified against its main components and then, with the help of regular expression pattern matching, the required info is extracted. The major drawback in the whole approach is that the development effort steers towards XML parsing and matching rather than implementing the core application functionality. In a design like this, the source code tends to be custom made for each application, thus resulting in non reusable components.

2.2 No more XML - nxpy

Being faced with such issues, we developed a reusable component that takes over the XML parsing, composing and decomposing functionalities. To boost our implementation we integrated transformation functions from and to programming language objects. On top of these, we released this component as an open-source project to ease future development of network provisioning platforms based on NETCONF. This is an advantage compared to commercial solutions (Tail-f ConFD) in terms of cost. Regarding existing open source implementations, (NETCONF Implementations) there are not any reusable modules available on top of which applications could be developed. As our module is becoming an indispensable component in our development efforts, we continuously integrate features and functionalities that match the needs of a wider community.

The device proxy, nxpy (nxpy NETCONF Proxy), is developed in Python and provides a thin layer that acts as a getter/setter when it comes to communicating with the network devices. Where nxpy excels is the ability to perceive certain device configuration blocks once they are retrieved from the network devices. Towards the network devices direction, XML generation from Python objects allows for building device configuration in an object-oriented programming manner. Both functionalities are powered by an XML-to-Python objects proxy that lies in the core of the module.
The device proxy provides a southbound and a northbound interface as shown in Figure 4. The southbound interface sends and receives XML structured NETCONF commands and configuration blocks. Translation from certain device configuration blocks to Python objects is defined in the heart of the module and can be easily extended horizontally to include more configuration blocks or vertically to include more configuration options in the same block. Once passed to the NETCONF client, the XML formatted command is sent to the device; if a configuration block is sent, the configuration is applied to the device. What makes our proxy unique is the northbound interface. The northbound interface is a Python. Under the hood, the device XML configuration is transformed into Python objects and, to move one step further, the objects come with setters and getters modules. This allows for communication with the devices, both ways.

Imagine an application that incorporates a function which retrieves a device configuration via a getter method. The XML configuration is then transformed into a Python object. The object attributes are the configuration parts in a layered manner. Each attribute may have child attributes, depending on the depth of the configuration and so on. Every key attribute is an object itself. And eventually, every object and attribute has setter methods. This allows for targeted configuration changes, additions and removals via object oriented programming. Once the application is done with retrieving, adding, changing or deleting, the configuration can be pushed back to the device. The new configuration is translated from Python objects to device XML NETCONF configuration. The whole process is depicted in the following communication chart (Figure 5).

![Figure 4. Developing applications with an XML-Python proxy](image)

![Figure 5. nxy communication process](image)

Bootstrapping begins with (1), the application sends a get_configuration request directly to NETCONF client (ncclient) and initializes an empty Python Device() nxy object. Ncclient contacts the network device via NETCONF and sends (2) the corresponding NETCONF command, XML formatted. The device responds to ncclient via NETCONF and sends the configuration in XML format (3). The client pushes the configuration back to the NETCONF proxy -nxy- (4) and the Device object binds to the device configuration. In nxy the XML configuration is translated to Python objects. The
application interacts only with the nxpy objects (5). Once the modification phase is over, the application transforms the objects back to XML configuration and sends it to ncclient (6). Ncclient applies (7) the XML configuration to the device and the response (8) is pushed (9) to the application as clear text. During the whole process the application has no interaction with the XML device configuration. In essence, the proxy saves the developers from the process of XML parsing and expression matching. This approach eliminates the need to build XML-related components and the development focuses on the actual platform. The Python objects interface allows for developing network applications that match network technologies and operations’ requirements and needs. And to go a step further, our implementation allows, apart from XML, exporting to JSON format as well.

2.3 nxpy configuration examples – comparison

The following examples illustrate how nxpy excels over pure NETCONF sessions and ncclient-only sessions. The examples depict how a simple task - changing the device name – is carried out using each of the three ways. Moreover they demonstrate how nxpy deals with complex configuration tasks.

2.3.1 NETCONF Session
As it was mentioned, using a pure netconf session requires sending commands in an XML-RPC manner. Thus changing the device name would require the knowledge of the exact XML syntax to accomplish it. In Figure 6, the highlighted configuration leads to the device name alteration.

As we are focusing on applications built on top of NETCONF this is only an informative example that aims at understanding what happens under the hood in the case that we are trying to perform a simple task.

2.3.2 Ncclient-only Session
This example goes back to the implementation of Figure 3, where ncclient applies XML formatted commands/requests to the underlying devices. In this case the developer must construct the XML expressions that lead to the desired changes on the device. For the case of changing the device name, the procedure is depicted in Figure 7.
This approach uses XML building libraries. The highlighted part shows the configuration required to change the device name. Compared to a pure SSH implementation, the usage of robust and stable XML libraries eliminates any syntax-error concerns, though it poses some limitations. The developer needs to know the tree format of the device configuration. The source code is usually not reusable. Complex configuration requests would lead to complex expressions which may be difficult to follow and debug. The most noticeable weakness of this approach is that communication between ncclient and the device is always one-way, development-wise. Ncclient only sends RPC commands and receives RPC replies. As a client, ncclient has no knowledge of what is sent and what is received, as long as it is valid and properly formatted. Nxpy comes to fill this gap.

2.3.3 Nxpy Session

This example is the evolution on the ncclient-only approach and reflects Figure 4 and Figure 5. For the same task, changing the device name the process is shown in Figure 8.

The highlighted single line is the actual part where the device name changes. The major difference with the previous example is that nxpy retrieves the device configuration and maps it to Python objects. Thus any changes carried out on Python objects are reflected on the device configuration.

A more complex example depicts the ease with which configuration changes can be made and pushed back to the device. In the previous example, the configuration was retrieved and changes were carried out on the device object. In the following example the existing device configuration will not be taken into consideration. Instead, the configuration will be built from scratch and only for the part that we intend to change or create. In short we will build a flow route that blocks traffic from the Internet to a certain IP address.

Initially, there is no configuration applied to the device as shown in Figure 9.
Then with nxpy we initialize an empty Python object which is called Device. We bind a flow to the device and then attach a route to the flow. For the record, a flow can consist of many routes. Eventually, we set the route parameters. The source code for this task is shown in Figure 10.

What has to be underlined is the ease with which this task is developed. Device configuration is done with object handling. In approximately five (5) seconds, the configuration is applied to the device, as confirmed by Figure 11.

Compared to an ncclient-only approach, nxpy gives great power over device manipulation, plus it is closer to the development process. Thus, freed from the need to construct XML documents, the whole process is enhanced towards the actual application development.

2.4 nxpy deployments

Nxpy lies in the heart of two network service provisioning platforms we operate. It is a core component of our Firewall on Demand (FoD) platform (Poulopoulos et al., 2012), (FoD webpage), (FoD project). FoD is a network security services platform that allows for bgp flowspec rule creation and application to our core routers. FoD is developed in-house by GRNET NOC and is based on the Python Django framework. The service is currently in production in our production network and its architecture is depicted in Figure 12.
In FoD, nxpy is used to translate XML device configuration to objects and vice versa. Apart from this profound functionality, nxpy forms the basis of a daily script that performs reconciliation. This script gathers the device configuration, transforms it to Python objects and compares them with the Django objects that are stored in the platform database. In case of mismatches, interested parties are notified about the errors. It has to be mentioned that FoD has mitigated a number of attacks with two of them being severe; namely a 15Gbps DDoS and a 7.5 Gbps DNS Amplification DDoS. The following (Figure 13) is a source code snapshot that illustrates the interaction between nxpy and the application core.

Besides Firewall on Demand, nxpy together with ncclient are deployed in GRNET’s AutoBAHN (GEANT AutoBAHN) pilot instance. AutoBAHN is a GEANT-provided provisioning tool that integrates with an NREN’s own systems to facilitate the multi-domain dynamic circuit provisioning service - GÉANT Bandwidth on Demand- BoD-(GEANT BoD). AutoBAHN’s Multidomain architecture is depicted in Figure 14.
The southbound interface of a single AutoBAHN instance is called Technology Proxy - TP. The TP translates network configuration requests to device configuration. AutoBAHN by design, ships with a Java-based expect-cli interface towards network devices. This practically means that all requests to the TP go via the expect cli interface. The issues we came across can be summarized by the following:

- Expect is prone to errors in the event of a device firmware upgrade that affects the device command prompt.
- Application of the configuration is carried out in a serial manner regarding the network devices. This could impose scalability and latency issues in large networks.

To solve the above, we replaced the Java TP with a Python TP that wraps nxpy and ncclient. Our implementation replaced the traditional serial manner with a threaded one. NETCONF is not susceptible to errors throughout the firmware versions. Apart from configuration application and reconciliation our TP executes check commands that verify whether VPNs are created and what is their status. The result is a faster, more stable, more scalable and efficient TP on top of nxpy.

3. Community applications support

Our NETCONF proxy does not target only our internal tools and services. As open software, anyone deploying NETCONF-capable devices, ranging from core routers to switches and even NETCONF-enabled servers, can use it. Extending the device proxy with configuration elements is straightforward and building applications on top of it does not pose any particular difficulty. The ability to export configuration in commonly used data formats (JSON, XML) make it a perfect candidate for integration with a plethora of development frameworks.

For now, the python device proxy module supports a large number of Juniper device configuration elements such as, interface naming, addressing, vlans, along with some advanced network features such as VPN creation and verification and BGP flow spec rule creation and synchronization.

4. Summary – Future plans

As services on demand and advanced operations require accurate and rapid device configuration GRNET steers towards NETCONF as a management protocol. With NETCONF invoking XML related operations that impose a burden to the actual development of applications, we developed an XML-to-Python objects proxy. Our proxy, nxpy, translates XML RPC configuration to Python objects and vice versa. This allows for efficient communication with NETCONF enabled devices and boosts the development process. Nxpy is an open-source project and is available for download via the nxpy project page (Nxpy NETCONF Proxy). Thus the community can contribute and build network management tools and service provisioning platforms on top of it.

Future plans on nxpy include the automated generation of Python classes based on device configuration schema (XML or YANG). This would allow for configuring even the most complex aspect of a network device. Nxpy is planned to be incorporated in a bgp reconciliation tool, currently under development.

Acknowledgements

The authors would like to thank Rowan Thorpe, Developer/Technical Support at GRNET NOC, for the syntax and grammar checking of the paper.

References

William Stallings, 1999, SNMP, SNMPv2, SNMPv3, and RMON 1 and 2, Addison-Wesley, Third Edition


GRNET, http://www.grnet.gr

Brian Hedstrom, Akshay Watwe, Siddharth Sakthidharan, 2011, Protocol Efficiencies of NETCONF versus SNMP for Configuration Management Functions, Network Automation, Paris, France

GRNET Monitoring platform, http://mon.grnet.gr


Leonidas Poulopoulos, Michalis Mamalis, Andreas Polyrakis, 2012, FireCircle: GRNET’s approach to advanced network security services’ management via bgp flow-spec and NETCONF, TNC2012, Reykjavik, Iceland

FoD webpage, https://fod.grnet.gr

FoD project, https://code.grnet.gr/projects/flowspy

GEANT AutoBAHN: http://autobahn.geant.net/

GEANT Bandwidth on Demand (BoD): http://www.geant.net/service/BoD/pages/home.aspx

Vitae

Leonidas Poulopoulos received his Diploma in Electrical and Computer Engineering from the University of Patras in 2005 and his M.Sc degree on Computer Science from the Department of Computer Engineering and Informatics (University of Patras) in 2010. Currently, he is (with) the development team of GRNET NOC. He designs and develops network management applications and web platforms and quite often, a mix of both. His job/interest/research profile can be found at: http://www.linkedin.com/in/leopoul

email: leopoul@noc.grnet.gr

phone: +30 210 7471096, +30 697 3845436

Dr. Yannis Mitsos received his Diploma in Electrical and Computer Engineering from the Democritus University of Thrace in 1999. His Diploma thesis was focused on the implementation of control algorithms using digital signal processor interworking with FPGAs. Since 2003, he holds a PhD degree from the Electrical & Computer Engineering Department of the National Technical University of Athens (NTUA). His PhD thesis explores the performance analysis of distributed functionality in high-performance network processors. Since 2005 he works for GRNET focusing on the deployment of regional infrastructure projects in SEE. Currently, he is head of GRNET NOC. His profile can be found at: http://www.linkedin.com/pub/yannis-mitsos/2/42a/1b1

email: ymitsos@noc.grnet.gr

phone: +30 210 7475687