Automatic Configuration of Opaque Network Functions in CMS

Automatic Configuration of Opaque Network Functions in CMS

Serena Spinoso*, Marco Leogrande†, Fulvio Risso*, Sushil Singh† and Riccardo Sisto*
*Department of Control and Computer Engineering, Politecnico di Torino, Italy
†PLUMgrid, Sunnyvale, CA, USA

***Abstract*—Cloud Management Systems (CMS) such as OpenStack are commonly used to manage IT resources such as computing and storage in large datacenters. Recently, CMS are starting to offer customers also the possibility to customize their network infrastructure, allowing each tenant to build his virtual network made of elementary blocks such as traffic monitors, switches, routers, firewalls, and more. However, tenants have to choose those network services among the list of services made available by the CMS and have no possibilities to customize the applications they want.**

This paper examines some of the modifications required in CMS to support a tenant-centric network service model, in which each tenant can install and configure their preferred network functions, without being limited to use only the list provided by the CMS. A prototype implementation validates the proposed approach and demonstrates the extent of the modifications in terms of languages and software components.

I. Introduction

The concepts of Software Defined Network (SDN) [1] and Network Function Virtualization (NFV) [2] allow Network Service Providers (NSPs) and companies to give more freedom to their customers. Unfortunately today any changes about location and settings of customers Virtual Machines (VMs) in data center networks have to be managed by the operator. In addition, tenants can use just the functions provided by their NSP. For these reasons, network operators are looking at new possible scenarios where tenants are offered the possibility to create Virtual Networks [3] managed and configured by the tenants themselves without requiring operators action. In this way a tenant could define how his traffic should be processed using a set of network functions chosen by himself. This could allow also a tenant to decide how to connect his resources (VMs), without having directly control of the physical network. In particular if these functions are distributed in the operator network or if they are all located in a data center, this service does not change from a tenant point of view. Today NSPs, which offer cloud-based solutions, leverage a Cloud Management System (CMS) to manage computing and storage in their data center, and another component, called Network Operating System (NOS), for network management. NOS and CMS interact to guarantee a multi-tenant environment: the NOS receives from the CMS a virtual network definition for each tenant and configurations for each function of that network. This interaction is limiting, because the tenant is allowed to build his virtual network only by choosing from network functions provided by his NSP. Hence if a tenant would like to insert a different function, the operator has to modify his system, taking care of the integration of the new function in terms of configuration and communication with the other components (like other network functions or the NOS).

In this paper, we propose a possible solution that enables configuration of network functions that are opaque from the operator point of view. In particular a network function is a module that processes traffic in a specific manner and could be implemented in software or deployed into a physical network element (e.g., firewall, DPI, NAT, router, etc). In our vision, NSPs could allow tenants to insert new functions, written by any programmer, in their virtual network, but the operator should not know how those functions work and which type of functions they are. Thus operators would handle these opaque network functions like black boxes, assuring however their total integration in the operator's network. This means that tenants have to be able to configure any functions in one of the ways supported by the functions themselves: taking the example of a tenant that uses a firewall, he has to be able to load a set of protection policies on the firewall, similarly a traffic's pattern in a DPI to check possible attacks.

The remainder of this paper is composed as follows: in Section II, we describe the different works that have completed our background; Section III presents an overview of our architecture; in Section IV, a prototype of our solution is described in detail; in Section V, we demonstrate the validity of our implementation through two use cases; finally Section VI concludes presenting possible future works.

II. Related Work

The research world has presented different works somehow related to ours. Among these works, we can find possible architectures for managing Network Service Chains (NSCs). One of these architectures was described in the work made by Beliveau [4], while the NSC Architecture (NSCA) is presented in [5] . However, such architectures do not have any mechanism to extend the set of functions allowed, and hence to introduce new functions neither to configure them. In addition, the concept of chain is more static than a virtual network: traffic can follow just one path chosen based on tenants policies, rather than being able to follow any arbitrary path in the network.

Another proposal related to virtual service chains is being developed within the European project UNIFY. The approach taken by this project is close enough to ours, because, in UNIFY, NSPs can distribute network functions in the whole network, locating management aspects in an automated orchestration engine [6] [7]. The UNIFY project has also expressed the need to have a service abstraction model for defining and programming service chains: however, at the best of our knowledge, it seems that the configuration of the single network functions that compose a service chain is overlooked, leaving the configuration issue an open topic.

A service description is needed by the CMS to understand the basic requirements of an opaque network function. H. Song has noted in [8] the need of a standardization of the information model, in order to represent the user's functional and resource requirements, and to map and apply these requirements to the underlying infrastructure. Literature helps us with different solutions, which address description at service level and at resources level, from both the hardware (physical and virtual) and software points of view. One of these proposals is VXDL [9] that is defined as a language for describing a virtual network topology, including storage, computing and links, and a virtual timeline, to specify when a certain resource is needed. Unfortunately this temporal constraint is difficult to synchronize with the orchestration engine. In this context, another example is the network-centric cloud architecture proposed in [10], where a centralized control layer should manage the resources available for all network services.

Finally there have been several approaches in literature for configuring network functions like the NETCONF [11] and SNMP [12] protocols. However from an operator point of view, the use of such protocols is quite limiting because tenants can use just those network functions which support such configuration protocols, while we are envisioning an architecture flexible as much as possible.

III. The proposed architecture

In our architecture, the main actors are: *operator* (or *Network Service Provider*, *NSP*), *tenant* and *programmer*.

The main objective of our architecture is to give flexibility to *tenants*, by allowing the set of functions available to a tenant to be extended according to the tenants needs. Reaching this goal by progressively increasing the overall number of network functions offered by the NSP is not trivial, because any requirement coming from a tenant might imply a huge integration cost; also, different tenants might request support for different network functions. This is why our proposal focuses on giving the possibility for a tenant to introduce any new network functions implemented by third parties (we refer to them as *programmers*) in his vitual network, and be able to configure them through a unified API provided by his *network operator*.

We also would like to relieve the programmer from the burden of integrating his own network functions, implemented as Virtual Network Functions (VNFs)[1], in every specific NSP architecture. The VNFs should be readably usable in any present and future architecture, without the need of specific integration efforts.

Finally the network operator should be able to load into his own network any third-party VNF without additional complications. Furthermore, we would like to avoid the insertion of any VNF-specific configuration plug-ins inside the network operator's CMS: this avoids the problem of supporting arbitrary front-ends inside the unified view offered by the CMS.

A. Challenges

There are challenges to be solved both when inserting such VNFs inside a virtual network and when configuring them. With respect to the insertion problem, there should be a way to load a VNF into a virtual network and link it to the other ones; furthermore, the spectrum of VNF configuration methods is very wide and, even if we can categorize them in common types, every function has its own quirks.

The insertion problem can be solved already by many CMS. If a programmer can provide a disk image of his VNF, a CMS can treat it like a regular Virtual Machine; also, since many of their network plug-ins already support stitching VMs inside a virtual network, a basic level of insertion can be achieved today. Many of the outstanding issues are related to the configuration phase instead; hence we focused our attention on them.

We also believe that, by having a rich configuration service, less complexity is needed in the insertion phase. As an example, let us consider the case of a third-party router deployed into a virtual network: in a traditional scenario, a tenant is required to deploy the router into a virtual network, then access its configuration interface through a virtual console (or similar mechanism) to configure the network interfaces of the router in terms of IP address, routing protocols, etc. In our vision, there should be no need to access this VNF-specific interface, and the tenant should be able to configure the router through the same API that he used to deploy the router in the network. In addition, having a suitable configuration service, an automatic configuration service could be enabled, for example, in the case of tenant's configuration errors. Considering the same router and a third-party web cache connected to the same subnet, if the tenant changes the subnet prefix and reconfigures just the IP address of the router interface, the NOS could be able to recognize such misconfiguration and hence should have the means to fix this error configuring properly the web cache.

Inserting opaque functions might bear possible high risks for NSPs: due to the lack of relationship between the programmer and the NSP that is installing an unknown function

[1]We use indifferently the terms "network function" and "VNF".

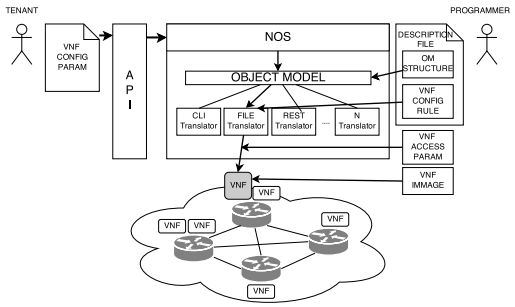

Figure 1: Architecture overview and translator workflow.

into his network element, the NSP could take precautions verifying that this function respects certain parameters. This problem, however, is out of our scope and it was also taken into consideration by other works, like [13], that addressed the possibility to run software modules in network elements.

B. Architecture overview

Figure 1 shows a high-level view of the whole system architecture. Each tenant can control his virtual network through a global interface, that is an operator-defined API. For each of the VNFs in the tenant's network, the NOS will receive configuration messages from the API and will interact properly with the actual VNF. Since each VNF could be ultimately configured through different configuration methods (e.g., file, CLI, REST, etc), and with specific details (formats, commands, etc), it is important to make able the NOS to configure a VNF regardless of their respective intrinsic details.

Having a unified description format helps all the actors involved: the programmer can define the VNF configuration format and supported methods in a way that is recognized by any network operator, that, in turn, is able to insert and use any VNF that adheres to the unified description format. This format also simplifies the projection of the configuration of the VNF through the tenant-visible API, since it is independent from the actual configuration methods used. The unified description format allows the transparent use of a configuration method among a list of standard ones. To be inserted opaquely, however, a VNF should support one or more of those methods, and the operator should support any of them on his system (according to any specific policies that might arise).

C. Configuration translators

Each configuration method could require specific parameters: for example, for a configuration through CLI, it is necessary to know which command enables administrative authorization. For this reason the architecture includes, for each configuration method, a specific *configuration translator* that is aware of all the particular techniques and parameters needed for that method. As shown in Figure 1, each translator configures the VNF directly. Having separate translators also makes the system more extensible and manageable, as it allows an easier insertion, replacement and removal of configuration methods: when the operator wants to support a new configuration method, the operator has just to make available a new translator.

Configuration translators receive multiple inputs (Figure 1) : *(i)* the tenants configuration received from the operator-defined API and saved into an *object model* to know the actual values that should be set inside the VNF; *(ii)* the *VNF configuration rules*, to know the format required to deploy those configuration values into the VNF; *(iii)* a set of *VNF access parameters* required to connect to the VNF (e.g., IP address of the VNF, root password, etc...) and to load the configuration into it.

The structure of the object model and the VNF configuration rules are VNF-specific; they are both provided by the programmerthrough a *description file*, written in the unified description format. This allows the programmer to write the description file only once, and use the same file even across different NSPs. The VNF access parameters are, instead, translator-specific and VNF-independent: the number and type of these parameters is standardized for each translator, but their actual run-time values are set by either the network operator or the programmer, depending on the specific case.

D. Configuration translators inputs

An instance of the *object model*, specific for a VNF, collects the configuration parameters of that VNF, provided by the tenant. The object model instance is self-descriptive: in other words, one can discover its structure from the instance itself. This is important because when the configuration translator receives the object model instance, it can derive the structure of the model that was used by the programmer in the description file; this is crucial to generate the VNF configuration in the right format. Using an object model also makes easier to change in a transparent way the global API provided by the operator and avoids data-structure formats specific for translator to collect the VNF configuration chosen by the tenant.

The *VNF configuration rules* are a set of directives used to drive the translator in generating the VNF configuration in the right format (Figure 1). They express the way to translate the structure and content of the object model instance into the specific structure required by the VNF configuration method. If a specific VNF supports multiple configuration methods, the programmer can include VNF configuration rules for all of them in the same description file.

The *VNF access parameters* are used to instruct the configuration translator about how to connect to the VNF and load the configuration provided by tenant. As explained before, the programmer does not set all of these parameters,

because some of them might be tied to some management aspect internal to the NOS, like VNF location.

All of these inputs will be used to generate the final VNF configuration, following the workflow shown in Figure 1. Taking the example of a firewall, a user would like to define the network policy rules. In this case, the object model instance contains the set of policy rules themselves; the VNF configuration rules specify the format of policy rules in the particular VNF architecture; VNF access parameters describe how to program the policy rules inside the firewall (e.g., the IP address, port and protocol required to connect to the firewall to deploy the configuration).

IV. Architecture Implementation

This section describes a prototype implementation of the architecture presented. We have also validated its workflow using two use cases described in the next section. We start to present some details, which have been left out of the description to keep the architecture more generic, about the choice of the languages used for the description file and the VNF access parameters. Then we describe our prototype and its validation.

Listing 1: YANG language example.

```
module router {
 import ietf-inet-types { prefix inet; }
 import ietf-yang-types { prefix yang; }
 list interfaces {
  //api:file:header "//Beginning of the Config File";
  //api:file:list_format "%NAME {\n";
  //api:file:separators "\n}\n";
  //api:file:footer "}\n //End of the Config File";
  key name;
  leaf name { type string; }
  list ethernet {
   //api:file:list_format "%NAME %VALUE {\n";
   //api:file:separators "\n";
   //api:file:footer "}\n";
   key name;
   leaf name { type string; }
   leaf address {
    //api:file:leaf_format "%NAME %VALUE\n";
    type inet:ipv4-address; }
   leaf hwid {
    //api:file:leaf_format "hw-id %VALUE\n";
    type yang:mac-address;}
  }
 }
}
```

A. Languages Choices

The YANG language [14] has been chosen for the description file. YANG is a data modeling language developed by IETF to model configuration and state data manipulated by the NETCONF protocol. In particular YANG was chosen for several reasons: it is orthogonal to network protocols and it is implementation-independent and human-readable; it is also a language developed with network configuration in mind and extensible, as it allows creation of user-defined statements.

In our case, the configuration data for a VNF is modeled in YANG by creating an object model specific for that VNF. An example of a possible YANG description file for a router is shown in Listing 1, where we define a structure to save the state of Ethernet interfaces. The idea is to have a data structure to enumerate all interfaces of a given router and, for each of them, store all of the network and physical addresses associated with that interface[2]. Accordingly, a top-level `interfaces` list is defined to include the names for all the interfaces to be configured; a nested `ethernet` list contains all addresses specific for an Ethernet interface.

YANG provides by default a number of directives to validate some proprieties of its statements. Examples of directives provided by YANG are: type checking; a default value for a `leaf` statement; definition of mandatory or optional statements (like `leaf`, `list`, `leaf-list` and others). Other simple validations are possible through the definition of new YANG types. A more complex validation system would require an extension of the YANG language[3].

Since, in the proposed solution, the description file includes both the structure of the object model and the VNF configuration rules, it means that those rules have to be specified in the YANG language as well.

B. VNF configuration rules syntax

VNF configuration rules take the form of special comments in the description file (Listing 1). These rules are defined in a particular statement with the following structure: `//api:<Translator_N>:<Rule_N> <Rule_V>`

where `<Translator_N>` specifies which configuration translator the rule belongs to and `<Rule_N>` and `<Rule_V>` represent the rule name and value. This allows us to group all the rules for a specific translator under a specific prefix: we can consider them similar to a programming language namespace, that allows us to reuse a rule name across translators, if we need to. `<Translator_N>` can assume values like "`file`", "`cli`", "`rest`", etc that denote the translators created in our system.

As an example, let us consider a translator to configure VNFs using files: each rule for this translator is preceded by the prefix "`//api:file:`". We can see some of them in Listing 1: `separators`, `list_format`, `leaf_format`, `header` and `footer`. All rule values are interpreted as strings. When generating the configuration file, `header` and `footer` are printed respectively before and after the current element (e.g., `list` or `leaf`), while `separators` is used to separate child nodes of the current element (of course it is not applicable to a `leaf` statement, which does not have child nodes by definition). Furthermore

[2]Usually a network interface is assigned only one network and physical address, but this is not true in the general case.

[3]In fact could be needed to have existence constraints: this is case when a parameter could exist only if another one was set or if this one has a particular value.

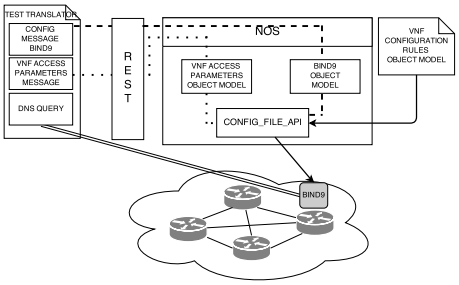

Figure 2: Prototype schema and test workflow.

`list_format` and `leaf_format` work like a `printf` of the C language, in which `%NAME` and `%VALUE` are expanded with values depending on the context. In particular, `%NAME` and `%VALUE` represent respectively the name of their YANG node (e.g., "ethernet" for the `list ethernet` and "address" for the `leaf address`) and its actual value (in the case of a `list`, it will be the value of its key). None of the keywords is mandatory.

C. VNF access paramters syntax

The default features of the YANG language are enough to define the VNF access parameters: what is needed is a configuration-oriented language. To keep the system definition uniform, YANG has been used for the VNF access parameters as well. In addition, it is interesting to note that many parameters stored in the VNF access parameters represent networking parameters (e.g., IPv4 or IPv6 address, MAC address and others). Hence, in addition to the built-in types, it is possible to leverage the YANG derived type statements defined by IETF in [15].

D. Prototype Implementation

In our prototype, a C++ library, called `Config_API`, has been designed to implement different configuration methods, one per translator. In particular to validate the architecture, we implemented a translator, called `Config_File_API`, to configure a VNF using files, regardless of their format (e.g., XML, text or more). This translator receives as inputs: *(1)* the YANG object model instance of a VNF (that contains the tenant's configuration); *(2)* the VNF configuration rules (specified in the YANG description file); *(3)* the VNF access parameters (defined in a different file, as shown in Figure 1).

With respect to the VNF access parameters, the translator expects the NOS to convert the configuration received from the tenant through REST into another object model, which is instead specific for the `Config_File_API` (dashed line in Figure 2).

For the sake of simplicity, in our implementation we have set all the VNF access parameters as configurable by the tenant. In a real world scenario, however, some of these parameters (e.g., the IP address where the VNF is located) should be managed just by the operator.

Finally we can note that our solution supports functions that require multiple configuration files. The `Config_File_API` library can be instructed to write different portions of the same YANG file into different configuration files, so that VNFs that require it can dump different parts of their data into different locations. This can be done because of the object model abstraction: for the purpose of the `Config_File_API` library, a YANG list at topmost level of the YANG file is no different from another list nested under it.

Listing 2: An excerpt of the Bind9 YANG description file.

```
module bind9 {
 list zone {
  //api:file:list_format "%NAME \"%VALUE\" {\n";
  //api:file:separators ";\n";
  //api:file:footer "};\n ";
  key name;
  leaf name {  type string;  }
  leaf type {
   //api:file:leaf_format "%NAME %VALUE\n";
   type string; }
  leaf file {
   //api:file:leaf_format "%NAME \" %VALUE\"\n";
   type string; }
  leaf master {
   //api:file:leaf_format "%NAME { %VALUE; };\n";
   type string; }
 }
}
```

V. Testing

Our prototype was validated using two network functions: Bind9 and Vyatta Core. Bind9 is an implementation of a DNS server and we have defined a YANG description file for this VNF collecting all the information needed to guarantee its correct behavior regardless of the role it is configured to act as: an excerpt of this description file is shown in Listing 2.

For our test, we have started manually, in our prototype, an instance of Bind9 and we have configured it to act as Secondary Master (which gets the zone data from another Name-server that is the Primary Master for that zone) by editing its object model through the REST interface. To better understand the test, we show an excerpt of the final configuration file, automatically generated by our system, where we have defined a zone in Bind9 syntax (Listing 4). In particular our test first uses a bash script to send HTTP messages to the NOS through the REST interface. After that, the Bind9 instance is interrogated directly to validate that the expected configuration was created and was loaded correctly. The workflow of our test is shown in Figure 2, as well as the structure of our prototype: first of all, we have sent two messages to set the VNF access parameters and the configuration parameters for Bind9; the Config_File_API has read its three inputs already explained,to generate and load the configuration file into the VNF; we have interro-

gated directly the Bind9 VNF to verify that all the process worked fine.

We have done a similar test for the second use case, Vyatta Core, which is a software router. Listing 1 shows an excerpt of YANG description file for this router. For our test, we configured an Ethernet interface, defining its IP address and the other main parameters, as shown in Listing 3. As in the previous case, we have validated our configuration with another bash script. This test, as in the previous case, has created an instance of an `ethernet list` in the YANG object model and has set its parameters. Then we have validated the Level 3 configuration of the Vyatta Core instance by testing its reachability through an ICMP request.

Listing 3: Vyatta configuration file.

```
interfaces {
 ethernet eth0 {
 address dhcp
 duplex auto
 hw-id 00:0c:29:64:66:1c
 mtu 1500
 smp_affinity auto
 speed auto
 }
}
```

Listing 4: BIND9 configuration file.

```
zone "example.com" {
 type slave;
 file "db.example.com";
 masters { 192.168.1.10; };
}
```

VI. Conclusion and Future Work

This paper focuses on opaque network function configuration inside NSP's networks. After illustrating the type of services that NSPs provide to their customers, the need of the tenant-centric model was motivated and it was illustrates how to extend the typical CMS architecture to integrate third-party VNFs. To do this, we leverage the use of a VNF description file that allows the NOS to know the main aspects of an external VNF.

Finally we presented a prototype of our solution. This prototype was validated by implementing VNF configuration through configuration files, using a solution that is independent from the specific format used by the VNF for its configuration files (e.g., XML, text file or proprietary). Our tests produced a sucessful validation and it consisted of a specific translator to create configuration files, which interacted with two different network functions: Bind9 (a DNS server) and Vyatta Core (a software router).

Possible future extensions could be the addition of more intensive validation mechanisms, since currently we leverage only the validation instruments provided by YANG. In particular this type of work could regard both the validation of configuration output (e.g., more complex constrains checking) and validation of the correct integration in the system (e.g., guarantee that all requirements defined by a final user are respected or guarantee the expected behavior of the virtual network). Also our solution could be tested with other types of VNFs to validate different configuration file formats.

Acknowledgment

The authors would like to thank PLUMgrid, Inc, a startup based in California, USA, which has supported this work.

References

[1] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: An intellectual history of programmable networks," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–98, Apr. 2014.

[2] "Network function virtualization." White Paper, Oct. 2012.

[3] N. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization," *Comput. Netw.*, vol. 54, pp. 862–876, Apr. 2010.

[4] A. Beliveau, "draft-beliveau-sfc-architecture-00." IETF Draft, 2013.

[5] P. Quinn and A.Beliveau, "draft-quinn-sfc-arch-04." IETF Draft, 2014.

[6] A. Császár, W. John, M. Kind, C. Meirosu, G. Pongrácz, D. Staessens, A. Takács, and J. Westphal, "Unifying cloud and carrier network," *Proceedings of. DCC, Dresden, Germany, to appear Dec*, 2013.

[7] W. John, K. Pentikousis, G. Agapiou, E. Jacob, M. Kind, A. Manzalini, F. Risso, D. Staessens, R. Steinert, and C. Meirosu, "Research directions in network service chaining," *CoRR*, vol. abs/1312.5080, 2013.

[8] H. Song, "draft-song-opsawg-virtual-network-function-config-00." IETF Draft, 2013.

[9] G. P. Koslovski, P. V.-B. Primet, and A. S. Charão, "VXDL: Virtual resources and interconnection networks description language," in *GridNets* (P. V.-B. Primet, T. Kudoh, and J. Mambretti, eds.), vol. 2 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 138–154, Springer, 2008.

[10] F. Pamieri and S. Pardi, "Enhanced network support for scalable computing clouds," in *Cloud Computing*, pp. 127–144, Springer, 2010.

[11] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. B. Ed., "Network Configuration Protocol (NETCONF)," RFC 6241, RFC Editor, June 2011.

[12] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "A Simple Network Management Protocol (SNMP)," RFC 6241, RFC Editor, May 1990.

[13] J. W. Lee, R. Francescangeli, J. Janak, S. Srinivasan, S. A. Baset, H. Schulzrinne, Z. Despotovic, and W. Kellerer, "Netserv: active networking 2.0," in *Communications Workshops (ICC), 2011 IEEE International Conference on*, pp. 1–6, IEEE, 2011.

[14] M. Bjorklund, "YANG - A data modeling language for the Network Configuration Protocol (NETCONF)," RFC 6020, RFC Editor, October 2010.

[15] J. Schoenwaelder, "Common YANG Data Type," RFC 6991, RFC Editor, July 2013.