A Two-Layered Data Model Approach for Network Services

Connectivity services are ubiquitous to enterprises, and many enterprises are looking to outsource basic networking services traditionally implemented using on-premise network equipment. The rising expectations on service providers to rapidly change the definition of services and the ability to introduce new types of network elements is leading to exploding complexity in the orchestration layer. The severity of this problem is such that the ability to introduce new services and new device vendors in the network is reduced due to the time and cost associated with such changes.

Carl Moberg and Stefan Vallin

Abstract

Connectivity services are ubiquitous to enterprises, and many enterprises are looking to outsource basic networking services traditionally implemented using on-premise network equipment. The rising expectations on service providers to rapidly change the definition of services and the ability to introduce new types of network elements is leading to exploding complexity in the orchestration layer. The severity of this problem is such that the ability to introduce new services and new device vendors in the network is reduced due to the time and cost associated with such changes.

We illustrate that a two-layered data model approach using the YANG language can help overcome these challenges. We use an example describing the process of implementing an IP MPLS VPN service in a network comprising sets of provider edge (PE) routers and customer edge (CE) routers. The example includes a service model and decomposition logic using data transformation, and we show the resulting configuration.

Introduction

Connectivity services are becoming ubiquitous to enterprises, and can be said to be one of the basic requirements for organizations to perform well in a connected world. This has created enormous opportunities for service providers that are able to scale their networks to meet the rising demands for cheap communication services across many geographical markets. As a result, the size of networks in terms of endpoints, as well as the amount of traffic they are required to carry, are expected to rise sharply over time.

At the same time, many enterprises are looking to outsource the networking services that they have traditionally implemented using on-premise network equipment. Examples of such services are end-point security using packet filtering, application acceleration, and collaboration tools. This shift in the location of applications inevitably leads to additional complexity for the service provider offering to host and manage such services on behalf of customers.

Carl Moberg and Stefan Vallin are with Cisco Systems–Tail-f Engineering, Stockholm, Sweden

There are two factors contributing to the above challenges [1]:

- The wide variety of services and the fact that they change at an increasingly rapid pace as service providers compete in the market.
- The challenge of keeping device-related configuration consistent and aligned with the intent of the deployed services along their lifecycle.

Current orchestration and activation solutions lack formal mapping between the service and device layer configuration models. These systems have historically not been built on data modeling technologies, but have been purpose-built for specific services or types of services. This hard-coding of service definitions and device types assume that the structure and syntax of the input data for service lifecycle operations does not change significantly over time, but that the definition of what constitutes a service and how that is provisioned and activated is relatively static. The rising expectations on service agility (the ability to rapidly change the definition of what services are) and device agility (the ability to introduce new types of network elements from new vendors) leads to exploding complexity in the integration code and dependency on manual steps. The severity of this problem is such that the ability to express new services and introduce new vendors in the network is reduced due to the time and cost associated with such changes.

COMMUNICATIONS STANDARDS

For example, experience from a large North American service provider shows that the delivery time for point-to-point VPN services for enterprise customers is counted in weeks, even when disregarding time to deliver the physical infrastructure. This delivery time is attributed to numerous manual steps where, for example, work orders are written in natural language in desktop document formats and where network engineers subsequently interpret the orders and translate them into hand-written command line interface (CLI) commands based on which vendors was deployed for a certain geographic area.

We will illustrate a two-layered data model approach using the YANG [2] language that can help overcome these underlying challenges. Splitting the management solution into a service layer and device layer is not new, but is a common pattern in network management software. However, in most deployments these layers are characterized by the following:

- The *service layer* is most often only described in an information-model, not a data model.
- The *device layer* has various application program interfaces (APIs) and command line interfaces (CLIs). APIs are most often RPC-oriented.

These interfaces create a gap between the service instances and corresponding device-configurations, a mapping from informal and often incomplete models to sets of CLI commands or operations to an RPC-based APIs.

Our suggested solution is characterized by the following:

- *Unified YANG data modeling* for both services and devices.

 0163-6804/16/$25.00 © 2016 IEEE

• *Transformation* from service models to device models.

The principle has been illustrated in previous work by Vallin *et al.* [3]. It provides the following benefits:

- Using a data modeling language to express service-models introduces strict formalism.
- A single data modeling language for device configuration removes hurdles around the variety of APIs and CLIs.
- Applying the same data modeling language in both layers enables formal transformation and validation techniques.

This leads us to suggest a way forward by decomposing the problem and applying known solutions to the resulting sub-problems.

Applying Data Models

We are suggesting the introduction of two distinct layers of formal data models:

- A *service layer* hosting the entities that represent the order-able services and their components.
- A *device layer* representing the configuration of the participating network elements.

We suggest using the same data modeling language in both layers, allowing for easy application of data transformation. By opening up for the use of transformation, we can leverage well known technologies, allowing for conversion of a set of data values from the data format of a source data system into the data format of a destination data system. Our system uses the YANG data modeling language for these purposes.

The application of data modeling languages provides a number of additional features that we can leverage in our system, including:

Correctness: The specification (data model) is the implementation. With traditional systems, there is always a significant risk that software developers misinterpret informal service specifications and implement them incorrectly.

Completeness: Model mapping specifies only how a service is created; the other operations are automatically generated by the system. With other systems, only a subset of the create, read, update, and delete (CRUD) operations are typically implemented completely. This leads to a need for manual configuration to complement the tasks supported by the system.

Service Agility: The northbound and southbound interfaces, as well as database schemas of the orchestration system, are automatically derived from the data models. With other approaches, these are manual tasks that become increasingly complex as the number of services and device types grows.

The process of defining a new service type starts with asking these fundamental questions:

1. What input parameters do we require when we create the service? That is, what are the parameters sent from a higher-level system such as an order management system or self-service portal?
2. What is the resulting set of device configurations to implement the service instance in the network?
3. What does the mapping between the service parameters and the device configuration look like?

```
module: l3vpn
   +--rw vpn
      +--rw l3vpn* [name]
         +--rw name          string
         +--rw endpoint* [id]
            +--rw id              string
            +--rw as-number       uint32
            +--rw ce
            |  +--rw device       -> /ncs:devices/
            |                        device/name
            |  +--rw local
            |  |  +--rw interface-name?     string
            |  |  +--rw interface-number?   string
            |  |  +--rw ip-address?         inet:
            |  |    ipv4-address
            |  +--rw link
            |     +--rw interface-name?     string
            |     +--rw interface-number?   string
            |     +--rw ip-address?         inet:
            |       ipv4-address
            +--rw pe
               +--rw device       -> /ncs:devices/
                                     device/name
               +--rw link
                  +--rw interface-name?     string
                  +--rw interface-number?   string
                  +--rw ip-address?         inet:
                    ipv4-address
```

Listing 1. Service Model, Tree Representation.

Informal and Formal Service Models: The answer to the first question above is captured in the service model. The first version of this model is normally an informal description of the data as an unordered list on paper. We use an IP VPN using MPLS transport as an example:

- VPN name.
- BGP AS number.
- A list of end-points defined as:
  - –CE device, interface identifier, and IP address.
  - –PE device, interface identifier, and IP address.

In this first version we list the PE devices as input. However, if we make sure that our system knows about CE-to-PE links, then the PE devices do not have to be identified in the input, but can be inferred from the topology. This serves to illustrate the power of model-driven systems: our system can handle both scenarios. The decision boils down to the question of how system borders are defined. In this example we use a model where explicit CEs and PEs will be used.

The next step is to transform the informal service model into a formal model expressed in the standardized YANG data modeling language. This activity should be done with a team of stakeholders including the product definition owner, networking subject matter experts, and YANG experts. The steps involved include working through the informal service definition as exemplified above, expressing them using the YANG language and associated best practices. The exact details vary based on the requirements from the product definition owner, the nature of the systems that will operate on the data models, and any expectations on modularity and extensibility of the models over time.

Device Models and Configurations: To answer the second question above: What are the resulting device configurations in the network corresponding to a service instance? This is defined by subject matter experts (in this case network engineers) who

```
module l3vpn {

  namespace "http://example.com/l3vpn";
  prefix l3vpn;

  import ietf-inet-types {
    prefix inet;
  }

  import tailf-ncs {
    prefix ncs;
  }

grouping endpoint-grouping {
    leaf interface-name {
      type string;
    }

    leaf interface-number {
      type string;
    }
    leaf ip-address {
      type inet:ipv4-address;
    }
  }

  container vpn {

    list l3vpn {
      description "Layer3 VPN";

      key name;
      leaf name {
        type string;
      }

      list endpoint {
        key "id";
        leaf id{
          type string;
        }

        leaf as-number {
          description "AS used within all VRF of the VPN";
          mandatory true;
          type uint32;
        }
        container ce {
          leaf device {
            mandatory true;
            type leafref {
              path "/ncs:devices/ncs:device/ncs:name";
            }
          }
          container local {
            uses endpoint-grouping;
          }
          container link {
            uses endpoint-grouping;
          }
        }

        container pe {
          leaf device {
            mandatory true;
            type leafref {
              path "/ncs:devices/ncs:device/ncs:name";
            }
          }
          container link {
            uses endpoint-grouping;
          }
        }
      }
    }
  }
}
```

Listing 2. YANG Service Model.

have experience with manually configuring devices to implement a VPN. Their contribution is the target configuration that should be the result of a service instance. This then is the input to the mapping process described below.

This step can be done either after or before informally defining the service model and its attributes. It might seem that device configurations is too low-level of a concern during service implementation; it is not. Rather, it is the most important step, since it is required to make the service operational in the network. Customer expectations are not satisfied just because the service exists in the supporting systems, and bills are sent. The intent of the service must be configured and activated in the network.

Data Model Mapping: The third step in the process of defining a new service type is to associate each attribute in the service model with the corresponding location in the device models as the basis for our data model transformation. We refer to this as mapping the service model into device models. There are two ways to specify this mapping in our system:

- **Declarative templates:** In many cases a straightforward template that maps service attributes one-to-one to device configuration models is sufficient.
- **Programmatic data model mapping:** In some cases algorithmic expressions are needed to map service attributes to device configuration models. In these cases a programmatic mapping approach can be used.

There is a fundamental difference between data model mapping and the traditional approaches using explicit procedures, workflows, device configuration templates, and other imperative techniques. Data model mapping provides a single mapping describing how service configuration data sets shall be transformed to the device configuration data sets. At run-time this can cover any operation, including instance modifications and deletion. Traditional techniques using workflows or CLI script templates, for example, have a fundamental limitation in that every possible operation on a data set needs an explicit definition. This is not true for data model mapping approaches.

An important question is whether to start with an informal service model and work our way down through a formal service model toward the device configuration (top-down), or start with the device configurations and work our way up to a service model (bottom-up). The preferred direction should be guided by the context at hand and take into account aspects such as, for example, requirements from systems consuming the service models, and if it is at all possible to change the resulting configuration.

An Examples of YANG as the Data Modeling Language for IP MPLS VPN

This example describes the process of implementing a VPN connectivity service in a network comprising sets of PE routers running Cisco IOS-XR and CE routers running Cisco IOS. The process described typically requires a couple of working days for a network engineer. The service model consists of 90 lines of YANG and the service mapping template is 180 lines of XML. The result is a fully functional VPN provisioning system with service-aware northbound interfaces including REST, CLI, Web UI, and a database schema for internal use. It also includes southbound interface drivers toward the network elements. The system has automatically derived CRUD operations including adding, updating, and removing end-points, changing BGP AS numbers while the service is running, and decommissioning of service

```
interface GigabitEthernet0/2
 description Link to PE
 ip address 10.1.1.1 255.255.255.252
exit
interface GigabitEthernet0/9
 description Local network
 ip address 192.168.0.1 255.255.255.0
exit
router bgp 65001
 neighbor 10.1.1.2 remote-as 100
 neighbor 10.1.1.2 activate
 redistribute connected
```

Listing 3. CE configuration.

```
vrf volvo
 address-family ipv4 unicast
  import route-target
   65001:1
  exit
  export route-target
   65001:1
  exit
 exit
exit
interface GigabitEthernet 0/0/0/1
 description link to CE
 ipv4 address 10.1.1.2 255.255.255.252
 vrf volvo
exit
router bgp 100
 vrf volvo
  rd 65001:1
  address-family ipv4 unicast
  exit
  neighbor 10.1.1.1
   remote-as 65001
   address-family ipv4 unicast
    as-override
   exit
  exit
 exit
exit
```

Listing 4. PE configuration.

instances with automatic clean-up of associated device configurations.

In this example we will work top-down, starting with the service model for the VPN and mapping the service parameters to a known set of configuration parameters in the devices. This is then the input data for our transform.

Service Model: The VPN service attributes according to our informal description is structured as a list of the entries, each with the following content:

- VPN name (the list key)
- A list of VPN end-points each with the following structure:
  - –Endpoint identifier (list key)
  - –BGP AS number
  - –Parameters related to a CE device:
    - * CE device identifier
    - * Customer interface identifier
    - * Customer IP address
    - * Uplink interface identifier
    - * Uplink IP address
  - –Parameters related to a PE device:
    - * PE device identifier
    - * Downlink interface identifier
    - * Downlink IP address

```
<!-- CE template for Cisco IOS routers -->
<name>{/endpoint/ce/device}</name>
<config tags="merge">
  <interface xmlns="urn:ios">
    <GigabitEthernet tags="nocreate">
      <name>{link/interface-number}</name>
      <description tags="merge">Link to PE</description>
      <ip tags="merge">
        <address>
          <primary>
            <address>{ip-address}</address>
            <mask>255.255.255.252</mask>
          </primary>
        </address>
      </ip>
    </GigabitEthernet>
    <GigabitEthernet tags="nocreate">
      <name>{local/interface-number}</name>
      <description tags="merge">Local network</description>
      <ip tags="merge">
        <address>
          <primary>
            <address>{ip-address}</address>
            <mask>255.255.255.0</mask>
          </primary>
        </address>
      </ip>
    </GigabitEthernet>
  </interface>
  <router xmlns="urn:ios">
    <bgp>
      <as-no>{../as-number}</as-no>
      <neighbor>
        <id>{pe/link/ip-address}</id>
        <remote-as>100</remote-as>
        <activate/>
      </neighbor>
      <redistribute>
        <connected>
        </connected>
      </redistribute>
    </bgp>
  </router>
</config>
</device>
```

Listing 5. CE XML mapping template.

Listing 1 shows the formal structure of the YANG service data model based on the tree-output from the pyang compiler; Listing 2 shows the full YANG module.

Device Configuration: We start by looking at the configuration in the CE router related to the IP VPN service instance in a network. Listing 3 is the subset of the CE-configuration; Listing 4 is the subset of the PE-configuration directly related to the IP VPN. The highlighted strings are all data that will be referenced from the service instances. This gives us all the output data required in the next step, where we map the service structure into applicable locations in the resulting device configuration according to the above.

Service Mapping: The next step is to define the mapping of service attributes to device configuration parameters. This example will use the declarative template model for the mapping. Our system uses an XML-based templating language to represent the output structure. Listing 5 and Listing 6 are the example XML

```
<!-- PE template for Cisco IOS-XR routers -->
  <vrf xmlns="http://tail-f.com/ned/cisco-ios-xr">
    <vrf-list>
      <name>{string(/name)}</name>
      <address-family>
        <ipv4>
          <unicast>
            <import>
              <route-target>
                <address-list>
                  <name>{../as-number}:1</name>
                </address-list>
              </route-target>
            </import>
            <export>
              <route-target>
                <address-list>
                  <name>{../as-number}:1</name>
                </address-list>
              </route-target>
            </export>
          </unicast>
        </ipv4>
      </address-family>
    </vrf-list>
  </vrf>
  <interface xmlns="http://tail-f.com/ned/cisco-ios-
      xr" tags="nocreate">
    <GigabitEthernet>
      <id>{link/interface-number}</id>
      <description tags="merge">link to CE</description>
      <ipv4 tags="merge">
        <address>
          <ip>{ip-address</ip>
          <mask>255.255.255.252</mask>
        </address>
      </ipv4>
      <vrf tags="merge">{string(/name)}</vrf>
    </GigabitEthernet>
  </interface>
  <router xmlns="http://tail-f.com/ned/cisco-ios-xr"
       tags="merge">
    <bgp>
      <bgp-no-instance>
        <id>100</id>
        <vrf tags="merge">
          <name>{string(/name)}</name>
          <rd>{../as-number}:1</rd>
          <address-family>
            <ipv4>
              <unicast>
              </unicast>
            </ipv4>
          </address-family>
          <neighbor>
            <id>{../ce/link/ip-address}</id>
            <remote-as>{../../as-number}</remote-as>
            <address-family>
              <ipv4>
                <unicast>
                  <as-override>
                  </as-override>
                </unicast>
              </ipv4>
            </address-family>
          </neighbor>
        </vrf>
      </bgp-no-instance>
    </bgp>
  </router>
</config>
```

Listing 6. PE XML mapping template.

templates for the CE-devices and PE-devices, respectively, and the highlighted syntax are references into the service model using XPath [4] expressions.

The XPath expressions in the highlighted strings use relative and absolute paths to reference parts of the service model. When an instance of the service is configured, our system transforms the service instance data by resolving the paths in the device templates in the context of the service instance data. This process produces a set of output data in a device-structure format. This can then be transformed into ordered sequences of commands or operations, depending on the nature of the protocol used toward the devices.

The transformation context also provides some additional data, including for example the device identifier captured in the *device* element, and the per-device namespace that is used to match on which subset of the template to apply. This allows us to have a single XML template defining the output data structures for many vendors and OS versions at the same time.

Conclusion

We have illustrated that a two-layered approach using the YANG data modeling language in both the service layer and device layer can help overcome current challenges to deploying service configurations in networks. The use of data modeling languages allows for application of declarative transformation technologies to the decomposition step between the service layer and device layer. This significantly reduces the complexity in the integration layer between the two, allowing for rapid development of services over time.

We have shown this using an example consisting of a simple IP MPLS VPN service data model and how it is applied to CE and PE routers running Cisco IOS and IOS-XR. The declarative decomposition is expressed using an XML-based device template language using XPath to reference values in service instances.

Acknowledgment

Our big thanks go to the Tail-f engineering team.

References

[1] W. Enck *et al.*, "Configuration Management at Massive Scale: System Design and Experience," *Proc. 2007 USENIX: 21st Large Installation System Administration Conf. (LISA '07)*, 2007.
[2] M. Bjorklund, "YANG — A Data Modeling Language for the Network Configuration Protocol (NETCONF)," RFC 6020, Oct. 2010.
[3] S. Wallin and C. Wikström, "Automating Network and Service Configuration Using NETCONF and YANG," *Proc. 25th Int'l. Conf. Large Installation System Administration*, 2011.
[4] W3C "XML Path Language (XPath) 2.0 (2nd Edition)," 2010.

Biographies

Carl Moberg (camoberg@cisco.com) is a technology director with Tail-f, a Cisco company specializing in OSS and NMS systems, and network management standards. He joined Cisco from the Tail-f Systems acquisition. Carl spends his time across standards work, large scale implementations, and product-oriented technology strategy. He has presented extensively on the subject of model driven network management, with a focus on NETCONF and YANG and is a member of the IETF YANG Model Coordination Group.

Stefan Vallin (svallin@cisco.com) contributes more than 20 years of network and service management experience to Tail-f, a Cisco company specializing in OSS and NMS systems, and network management standards. He has presented at a wide variety of international network engineering and management conferences. Before joining Tail-f, he worked as a OSS solutions architect at Data Ductus. Dr. Vallin also served as a network management specialist at Ericsson and a research engineer at the University of Linköping. He holds a M.Sc. degree in computer science from Linköping University, and received his doctorate in network management from Luleå University of Technology.