SCTE·ISBE CABLE-TEC EXPO 2017
DENVER, CO
OCTOBER 17-20

2017 Fall Technical Forum
SCTE·ISBE • NCTA • CABLELABS

Bridging the Gap Between ETSI-NFV and Cloud Native Architecture

A Technical Paper prepared for SCTE/ISBE by

YuLing Chen
Senior Technical Leader
Cisco Systems Inc.
375 East Tasman Drive
San Jose CA 95134
408-393-5606
yulingch@cisco.com

Alon Bernstein
Distinguished Engineer
Cisco Systems Inc.
375 East Tasman Drive
San Jose, CA 95134
alonb@cisco.com

Table of Contents

List of Figures

List of Tables

Introduction

In recent years, Network Function Virtualization (NFV) has been introduced into the Telecom industry to deliver reliable and efficient commercial networking services in programmable standard hardware systems, called Virtualized Network Functions (VNFs). NFV promises benefits in the savings of operational and capital expenditure (OpEx and CapEx), as well as the increased automation, operations simplification, business agility, and faster time to market.

The cloud native microservices container architecture was originated from the webscale providers such as Amazon, Google, and Netflix. The approach of cloud native is to break down a monolithic application into small microservices and deploy as containers in the cloud. One of the attractions of this approach is that applications can be tested in an iterative and distributed model, without taking applications offline. In the cloud world, large scale applications have been developed, tested, and deployed with more agility using this distributed model.

Since 2016, several large service providers have publicly embraced the move to a microservices architecture in the telco cloud. [1] There have been announcements from major service providers to use containers to build out their network function virtualization infrastructure. Some key telecommunications equipment suppliers are using microservices to implement some of their software. Open-source initiatives are moving towards microservices and containers. In NFV space, there is a trend of moving from the virtual appliance based solutions to the cloud native approach, which is referred to as the Cloud Native NFV.

The NFV world has been following ETSI NFV references. However, most of the ETSI published documents were based on case studies and Proof of Concepts built on virtual appliances. There is a gap between ETSI NFV and the cloud native approach. With more and more cloud native solutions appear in NFV, there is a need to augment the existing ETSI NFV specifications so as to continue guiding the NFV world towards interoperability and standardization.

To support this effort, this paper identifies the elements in the ETSI NFV Management and Orchestration (MANO) reference architecture that need to be adjusted when applying the cloud native approach in NFV. We also propose a pragmatic software architecture that realizes the NFV MANO functionality using the cloud native approach. With the focus on the network service design and deployment, which is the core functionality of the NFV Management and Orchestration systems, we exercise the TOSCA language for the service modeling in the cloud native environment.

The Cloud Native Trend in NFV

Since 2012, driven by leading telecoms network operators, the European Telecommunication Standards Institute (ETSI) has been working on NFV requirement prioritization, high level architectural framework definition, development guideline specification, and Proof of Concept organization. ETSI has published a series of documentation and specifications in these related efforts. The documentation has been widely referenced and adopted in the NFV space.

What ETSI NFV advocates has been the moving of network functions from specialized proprietary hardware to virtualized software that can be deployed on standard hardware equipment. [2] Now, with the

cloud native adoption in NFV, we observed that the network functions together with the MANO systems are moving into the cloud as a form of microservices containers.

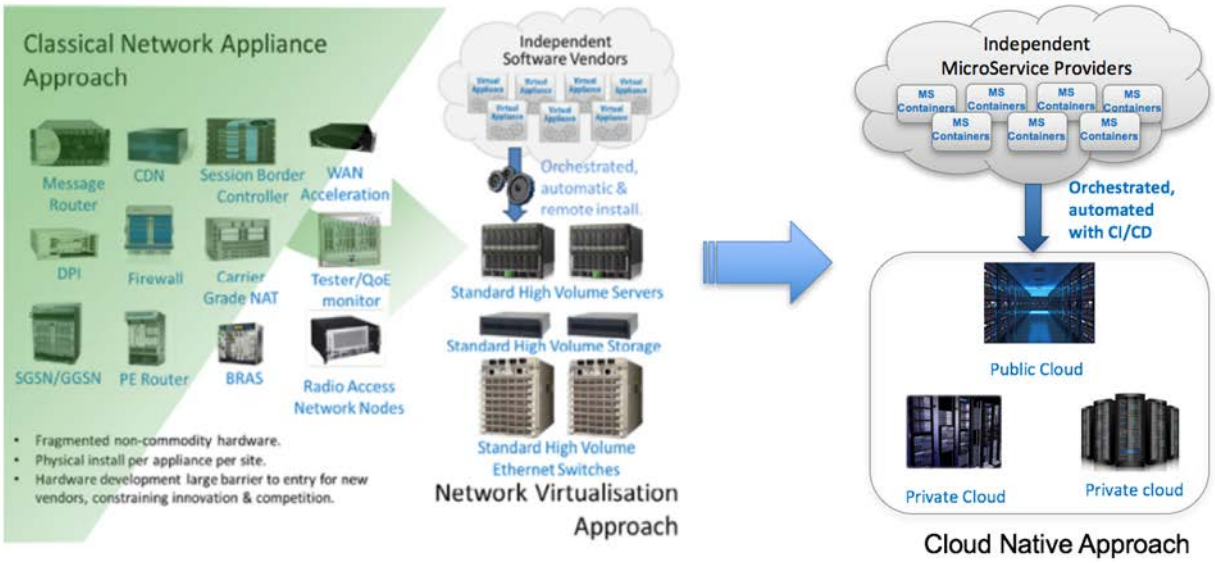

Figure 1 - The Trend of the Cloud Native NFV

As described in Figure 1, with the Network Virtualization Approach, which ETSI-NFV has been focusing on, the classical network appliances with non-commodity hardware move to the software based virtual appliances deployed in the standard equipment. With the cloud native approach, the network functions, which were implemented as monolithic applications, now are broken down into smaller microservices, and deployed as containers in both the public and private clouds. Leveraging Continuous Integration and Deployment (CI/CD), these microservices containers are orchestrated and deployed with automation. The independent software vendors who used to produce full-fledged network functions now become the vendors of smaller microservices.

More specifically, we observed the adoption of the cloud native solutions in the following NFV areas:

- VNFs
  - More and more VNFs are broken down into smaller microservices containers.
  - More and more NFV applications are packaged as microservices containers and deployed in the cloud native environment.
- MANOs
  - More and more NFV Management and Orchestration systems are deployed in the microservice container environment
- VIMs
  - Container orchestrators such as Kubernetes and Docker Swarm appear in NFV applications

ETSI NFV Adaptation to the Cloud Native Architecture

ETSI NFV ISG has published a series of specifications including the ETSI MANO GS (Group Specification) *Network Functions Virtualisation (NFV); Management and Orchestration*. [3] The specification lays out the NFV MANO objectives and concepts, defines the high level reference architectural framework, and specifies the information elements in an NFV MANO system .ETSI Management and Orchestration Architectural Framework

ETSI NFV reference architectural framework defines three functional blocks in the NFV-MANO domain: NFV Orchestrator (NFVO), VNF Manager(s) (VNFMs), and Virtualized Infrastructure Manager(s) (VIMs).

- **NFV Orchestrator (NFVO)**

NFVO is responsible for the on-boarding of a new Network Service (NS) composed of multiple VNFs, VNF forwarding graph, Virtual Links, and, as an option, Physical Network Functions (PNFs). The orchestrator also controls the life cycle of the Network Service, validates and authorizes NFVI resource requests, manages global resources, as well as the policy of the Network Service instances.

- **VNF Manager(s) (VNFMs)**

The VNFM focuses on the life cycle management of individual VNF instances. A VNF manager takes the responsibility of the management of a single VNF instance, or the management of multiple VNF instances of the same type. VNFM also serves as an overall coordination and adaptation role for configuration and event reporting between the VIM and the EM systems of traditional operator architectures.

- **Virtualized Infrastructure Manager(s) (VIMs)**

The VIM is responsible for controlling and managing the NFVI compute, storage and network resources. At the same time, it collects performance measurements in the infrastructure and makes the data available from other functional blocks for monitoring purposes.

The NFV-MANO architectural framework also identifies main reference points for the exchange of data between the corresponding defined functional blocks.

Proposed ETSI MANO Reference Architecture Augmentation for Cloud Native NFV

As a standard specification, ETSI focuses on high level architecture, development guidelines, and interoperability enabled by open interfaces. Most of the specifications in ETSI MANO GS continue serving the purpose when applying to the Cloud Native NFV. Nevertheless, augmentation is needed in some areas because of the differences between the VM based and cloud native solutions.

In the cloud native architecture, the network functions are deployed in the cloud as microservices containers. The granularity of the deployed instances is much smaller based on microservices design and implementation. A VNF in ETSI context would contain multiple microservice containers working together in the cloud native context. This adds complexity to the management and orchestration of the system. The fully distributed architecture, coordination and communication among the microservices, fault monitoring and recovery in smaller but more specific portion of the software, all contribute to the complexity of the MANO system.

On the other hand, the cloud native architecture brings advantages to NFV MANO systems including some critical pain points. One pain point in the VM based solutions is to provide high availability to the service providers by spawning new instances in the cases of faults, failures or scaling out to handle larger workloads. The time needed for spinning up a new VM has been the bottleneck in the VM based solutions. Using the cloud native approach, because the granularity of the independent unit for recovering or scaling out is much smaller, and the container start/stop is much faster, the latency of spawning a new network function composed of a set of microservice container instances is much smaller than that in the VM based solutions.

To highlight the differences between the VM based solution and the cloud native based approach in NFV, we propose the Cloud Native NFV high level reference architecture with the revised terminologies as an augmentation to the original ETSI NFV MANO reference architecture, which is illustrated in Figure 2.

As described in the diagram, The Network Function Cloud Infrastructure (NFCI) contains the public cloud and the private cloud(s) with containerization layer(s) to provide the infrastructure for the network services deployed as containers in the Cloud Native architecture. On top of the NFCI, a set of microservice containers work together to realize the functionality provided by a Cloud Network Function (CNF). Each microservice in a CNF is called a Cloud Network Function MicroService (CNFMS). A list of CNFs chain together with traffic flowing through the network functions becomes a CNFFG.

NFV Orchestrator (NFVO), CNF Manager (CNFM), and Cloud Infrastructure Manager (CIM) are the functional blocks of NFV MANO systems. The NFV MANO system communicates with OSS/BSS, CNFs in CNFFG, and NFCI through interfaces to manage and orchestrate the network services provided by the network functions deployed as microservices in the NFCI.

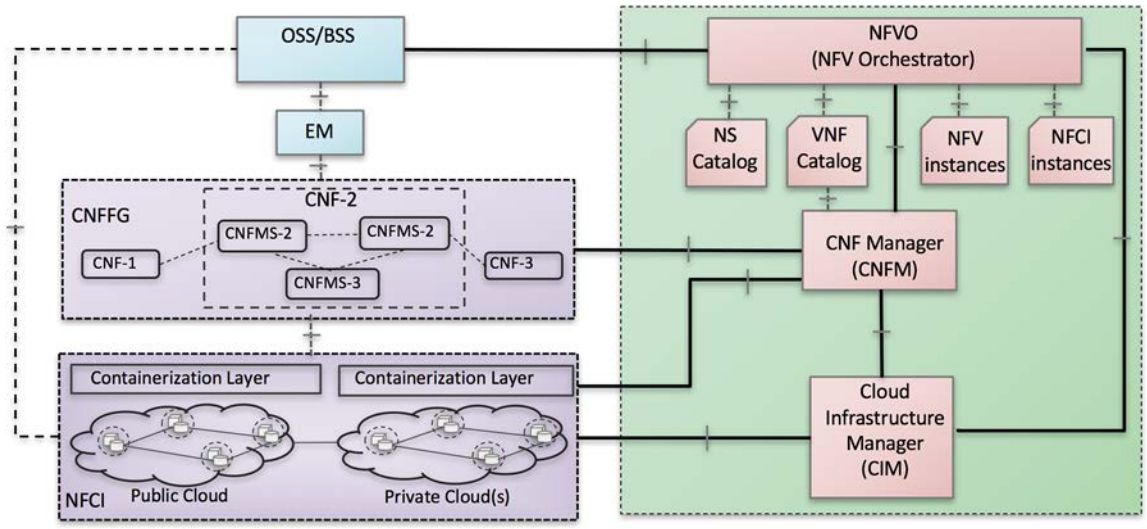

Figure 2 - Proposed ETSI MANO Reference Architecture Augementation for Cloud Native NFV

The detailed description of each functional block and component in Figure 2 is as follows:

- **Cloud Network Function MicroService (CNFMS)**

CNFMSs are the microservice containers from which a Cloud Network Function is composed.

- **Cloud Network Function (CNF)**

CNFs are the network functions deployed in the cloud as microservices, usually in container format.

- **Network Function Cloud Infrastructure (NFCI)**

NFCI provides the underline physical infrastructure for the network functions. This includes the hardware equipment for the computer, networking, storage, as well as the containerization layer on top of the hardware platform. CNFs are deployed on top of the NFCI.

- **Cloud Infrastructure Manager (CIM)**

CIM is responsible for controlling and managing the NFCI compute, storage and network resources, as well as scheduling the microservice containers in the cloud. It manages the lifecycle of the containers in the cloud.

CIM also collects performance measurements in the infrastructure including container level, and makes the data available for other functional blocks for monitoring purposes.

Other responsibilities of CIM include virtual networking control and management, as well as the southbound integration with various network controllers to achieve the physical network control and management capabilities.

Examples of the CIMs available in the market are Kubernetes, AWS ECS, and Docker Swarm.

- **Cloud Network Function Manager (CNFM)**

CNFM focuses on the life cycle management of individual CNF instances. In the cloud native architecture, a CNF is usually composed of a set of containers that implement a network function. A CNF manager takes the responsibility of the management of the multiple container instances of the same network function. To control the lifecycle of the CNFs, CNFM works closely with CIM, which manages the lifecycle of the individual container of the CNF.

CNFM also serves as an overall coordination and adaptation role for configuration and event reporting between the CIM and the EM systems of traditional operator architectures.

- **NFV Orchestrator (NFVO)**

The NFVO continues serving the responsibility of on-boarding a new Network Service (NS) composed of multiple CNFs, CNF forwarding graph, Virtual Links, and, as an option, Physical Network Functions (PNFs). The orchestrator also controls the life cycle of the Network Service including instantiation, scale-in/out or up/down, performance measurements, event correlation and termination. Further key operational functions are global resource management, validation and authorization of NFCI resource request, as well as policy management of Network Service instances.

- **Cloud Network Function Forwarding Graph (CNFFG)**

The CNFFG contains a list of CNFs and the virtual links among the CNFs and the physical endpoints.

Cloud Native NFV MANO Software Architecture

As a standard specification, ETSI focuses on high level architecture, development guidelines, and interoperability among systems produced from different vendors. In order for the industry to generate real products in microservices container architecture, we further refine and develop the functional blocks into micro services, helping to implement and realize the functionality mentioned in the previous section. We also realized that a real NFV MANO product needs to address more operational issues and challenges than what ETSI has specified. This includes the design phase support, network control in the cloud native environment, data collection, monitoring, and analytics. [4]

Figure 3 illustrates a pragmatic NFV MANO architecture using the cloud native approach.

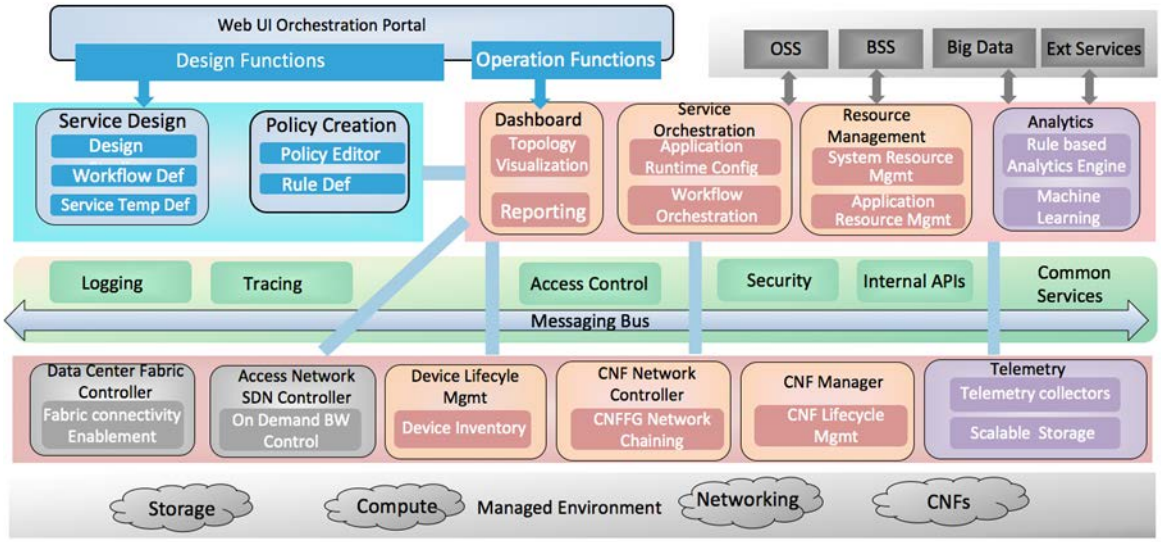

Figure 3 - A pragmatic NFV Software Architecture in the Cloud Native Environment

Figure 3 contains the microservices for the NFV Management and Orchestration in both the Design Phase and the Runtime Execution Phase. The design phase generates the network services model to be deployed onto the target private or public cloud(s). The Runtime execution phase contains the microservices for the network service deployment, orchestration, lifecycle management, network control, monitoring, and analytics. Besides the functional services, there are a set of common infrastructure services for all the microservices containers in the cloud native architecture environment.

1. Network Service Design Phase

One of the key functionality of NFV MANO is the Network Service Orchestration. Before the MANO can orchestrate the virtual and physical network services, we first need to design the services with rules and policies to indicate how the services are being deployed at run-time.

The Service Design microservice provides interfaces, usually in a visual studio way, for the user to design and model the network services and store them into a network service catalog (NS Catalog). The definition of the services need to specify how and when the CNFs are realized in a target environment. In particular, the definition would need to include the logical catalog items, together with selected workflows and instance configuration data, completely defines how the deployment, activation, and life-cycle management of CNFs are accomplished.

To facilitate the portability of the service design, it is desired to use a standard modeling language to describe the service definition. With such definition, any NFV orchestrator that is compliant with the standard modeling language can take the service specification and deploy into the target cloud environment.

2. Run-time Execution Phase

Various microservices work together during run-time to realize the functionality of the network service deployment, configuration, monitoring, and data analytics.

Taking the generated service definition as the Network Service Descriptor (NSD) and Cloud Network Function Descriptor (CNFD), the Service Orchestrator executes the workflow specified in NSD and works with other microservices to control the lifecycle of the network service; the Dashboard visualizes the topology of the network with basic monitoring of the health of the system; the Global Resource Manager allocates both the system resources and the network application resources; the CNF Manager deploys and manages the lifecycle of the Cloud Network Functions; the CNF Network Controller focuses on the control and management of the virtual networking among the CNF instances; and the Device Life Cycle Manager manages the physical life cycle of the devices.

Container networking in the cloud native environment is important to NFV applications. Since the CIM is realized by third party container orchestration tools, and most of the tools lack the full-fledged networking feature including policies and security support across different compute nodes, we would need to plug in various types of Network Controllers to augment the capability needed for the network functions deployed in the cloud.

The CNF Network Controller stitches the network connectivity between microservice containers. Usually there are two types of network traffic going through between CNFs: the control traffic and the data traffic. The control traffic usually requires lower bandwidth with relatively longer latency. The data traffic requires higher bandwidth with lower latency. Most container orchestration tools contain sufficient support for the control traffic. However, for data traffic, specialized CNF controllers will work with high speed virtual routers to realize the data plane acceleration in the cloud native environment.

The Data Center Network Controller and Access Network Controller are southbound plugins to the CIM to provide the physical networking to virtual networking mapping in the NFV system.

Telemetry and Analytics microservices work together to realize the collection, streaming, storage, and analytics of the operational data collected from the network.

Besides the functional services to achieve the NFV MANO capabilities, there are a set of common services that are particularly important in microservice container architecture.

The messaging bus enables the loosely coupled integration architecture in microservice container environment using publish/subscribe way of the inter container communication. Another important role of the messaging bus is to support large amount of telemetry data pushed from the network functions. The scalability and performance of the messaging bus is critical to the success of the cloud native architecture. Currently Kafka is one of the most popular messaging tools that are widely used in the microservices container environment.

Logging and tracing help with efficient and effective troubleshooting across distributed microservies. Open source tools such as fluentd, ELK, open tracing, and zipkin are popular ones to enable centralized logging and cross service tracing capabilities.

Service Design and Deployment using TOSCA Modeling Language

Network Service design and deployment is fundamental to NFV MANO functionalities. As the result from the service design, network service descriptor is generated. At run time, the NFV Orchestrator deploys the network service as CNF instances in the cloud native environment.

1. Service Design and Deployment Process Flow

The process flow of the Network Service Design and Deployment is described in Figure 4. The first step is to design the service using visualization tools in the Design Studio. The Design Studio is a graphical interface for the user to define the network services that contain the network nodes and the relations among them as links. A typical graphical Design Studio provides a set of drag and drop tools to help make the modeling process easy and intuitive to the user. During this process, the user enters the workflow definition, service template definition, and policy description as the input. As the result from this step, a Network Service Descriptor (NSD) will be generated and stored in a database called NS Catalog. At run time, as part of the NFVO, a NS Deployer would read the NSD from the catalog. The NS Deployer decomposes, translates, and converts the NSD into CNFD and stores it into the database called CNFD Catalog. The CNFD contains the specifications of the Cloud Network Functions that are ready for being deployed in the cloud. After that, the CNF Deployer reads the CNFD from the catalog and converts it into the executable artifacts accepted by the target cloud provider. Finally, the CNF Deployer communicates with the CIM to deploy the workload into the cloud.

Figure 4 describes the process flow of the Network Service Design and Deployment.

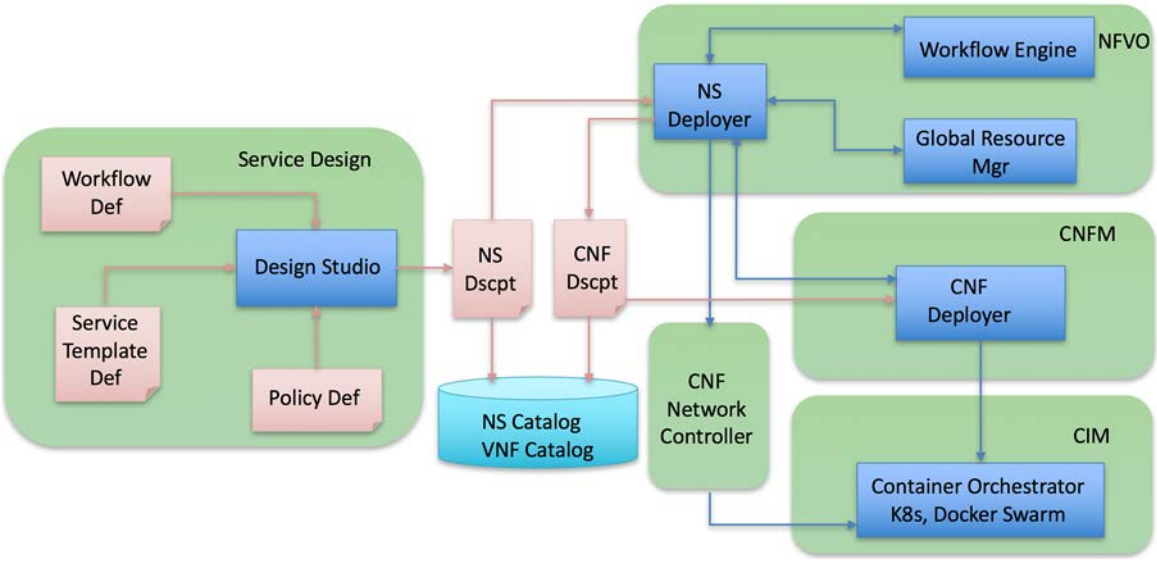

Figure 4 - Service Design and Deployment Process Flow

In the process described above, if the descriptor of the Network Service (NSD) and Cloud Network Function (CNFD) is specified in a standard modeling language, any NFV MANO that is compliant with

such standard language will be able to deploy the workload into the cloud. This has been the goal of TOSCA modeling language.

2. Network Service Modeling using TOSCA

Topology and Orchestration Specification for Cloud Applications (TOSCA) is a standard modeling language managed by industry group OASIS that can be used to orchestrate NFV services and applications. TOSCA delivers a declarative description of the application topology for a network or cloud environment that includes all its components, which may include the need for load balancing, networking, computing resources, and other software. It can also be used to define the workflows that need to be automated in the cloud.

The TOSCA modeling language includes concepts such as nodes and relationships, whereby a node is an infrastructure such as network, subnet, or a server software component. TOSCA can help define how these nodes and services work together. TOSCA uses templates to automate the configuration of these relationships.

TOSCA facilitates high levels of service portability, making services portable to any cloud or application that is TOSCA compatible. The data model also enables easier migration of applications. It is inherently infrastructure-agnostic, and thus is extensible to enable the automation of software-defined networks, in combination with NFV and clouds, to simplify end-to-end service orchestration for cloud and telco operators.

Figure 5 illustrates how to use TOSCA modeling language to model NFV Network Services and achieve the portability of deploying the same services in different cloud providers. [6] As described in the diagram, a CNFFG defined in TOSCA language can be deployed by a TOSCA compliant orchestration engine to different types of Cloud Providers such as AWS, Kubernetes, or Docker Swarm. The TOSCA compliant orchestration engine applies the necessary automatic matching, translation and optimization between the application requirements and the NFV infrastructure capabilities provided by the target cloud providers to achieve the portability.

TOSCA supports XML, JSON, and YAML implementation of the data model. With the industry trend moving to microservices container architecture, more and more TOSCA implementation products started the effort in deploying TOSCA defined services as microservice containers in the cloud native environment.

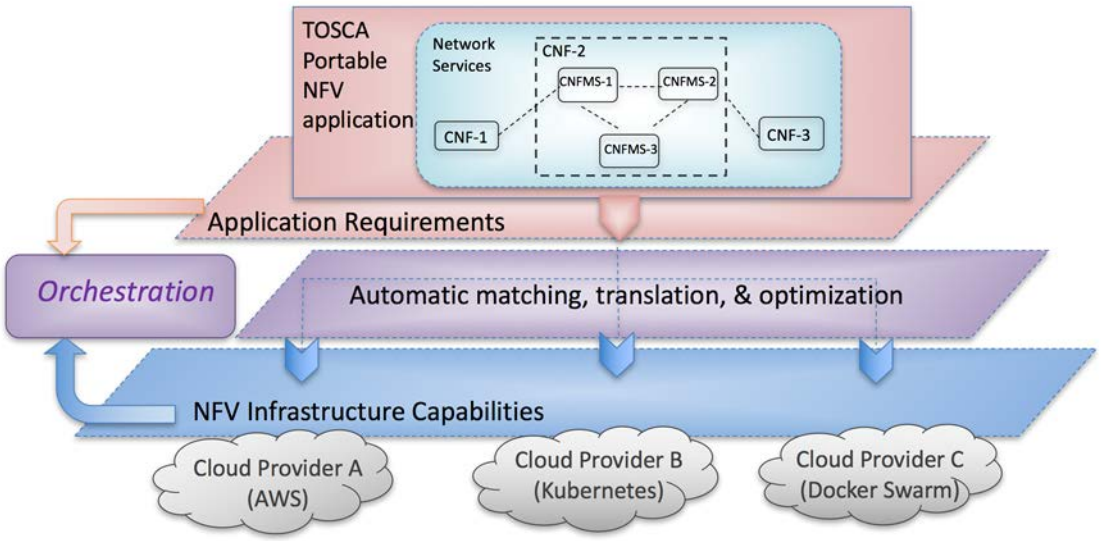

Figure 5 - TOSCA to support portable NFV applications[6]

3. TOSCA and YANG

YANG is a data modeling language used to describe configuration and state information. It was published by Internet Engineering Task Force (IETF) in 2010. YANG has been used to model networking devices and services – i.e., an object and its attributes. YANG defines the data models that are manipulated through the NETCONF protocol.

There has been a battle between using TOSCA or YANG modeling languages in the NFV context. These two modeling languages are not competitors but complementary to each other in different perspectives. TOSCA focuses more on the network topology, cloud workload, workflow representation, and deployment artifacts specification of the network services. The goal of TOSCA is to prepare a declarative specification for the workload being deployed in the cloud. YANG focuses more on the configuration of the network functions in the cloud. YANG provides the ability to easily configure network devices in a human readable fashion.

Because of YANG's strength is in configuring networking devices while TOSCA's strength is orchestration, we suggest using TOSCA in NFVO and CNFM, while using YANG for the configuration of CNFs and PNFs in NFV MANO architectural framework. [5] Figure 6 illustrates the idea of using both TOSCA and YANG in different functional blocks of the ETSI NFV MANO reference architecture.

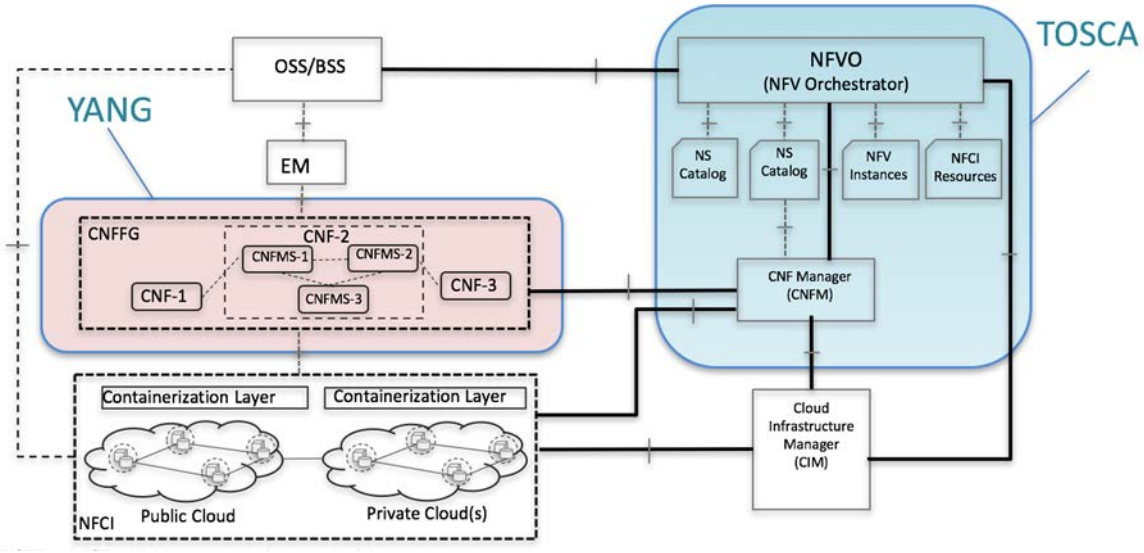

Figure 6 - Using TOSCA and YANG in different perspectives of NFV applications[5]

4. TOSCA for Cloud Native NFV

The TOSCA metamodel uses the concept of service templates to describe cloud workloads as a topology template, which is a graph of node templates modeling the components that the workload is made up of, and as relationship templates modeling the relations between those components. TOSCA further provides a type system of node types to describe the possible building blocks for constructing a service template, as well as relationship types to describe possible kinds of relations. Both node and relationship types may define lifecycle operations to implement the behavior an orchestration engine can invoke when instantiating a service template.

An orchestration engine processing a TOSCA service template uses the mentioned lifecycle operations to instantiate single components at runtime, and it uses the relationship between components to derive the order of component instantiation. Network Service workflow can be modeled using TOSCA leveraging the relationship modeling capabilities from TOSCA.

The TOSCA simple profile defines a number of base node types and relationship types to be supported by each compliant environment. Furthermore, it is envisioned that a large number of additional types for use in service templates will be defined by a community over time. At the same time, TOSCA is highly customizable with type inheritance capabilities built in the language. Specialized TOSCA engines can build the support for customized node types and relationship types to satisfy the needs.

TOSCA has been popular in the network service modeling in NFV. OASIS published the TOSCA Simple Profile for Network Functions Virtualization (NFV) Version 1.0 in May 2017. [8] However, the specification is VM based without the microservices container support. There is a need for extending the TOSCA specifications to support the Cloud Native NFV. With the strong growth of the Cloud Native NFV in the Telco space, this could be the next step from OASIS in the near future.

Before we have a set of TOSCA base node types and base relationships types defined by OASIS for the Cloud Native NFV, we can leverage the generic TOSCA base types with customization. Recently, there

has been research looking into how to use customized types in TOSCA to model generic applications deployed as Docker containers in the cloud [9][10]. We can take the same approach while adding additional custom types needed in the Cloud Native NFV context.

5. Example of using TOSCA Modeling Language

In this section, we will exercise an example using TOSCA modeling language to design and deploy a network service in a CMTS orchestration system. The goal is to illustrate the approach of using TOSCA to model the Network Service that contains sufficient information for the NFVO to deploy in a cloud native environment.

In this example, we will model a Physical Network Function called Remote PHY Device (RPD), which connects to a Cloud Network Function called Cloud Cable Modem Termination Service (CCMTS) to consume the CMTS services. To make it simple, we assume the CCMTS contains only one microservice container. We also assume that the CCMTS will need to be deployed as a Docker container in the cloud. The lifecycle of CCMTS and RPD needs to be specified in the service model.

To achieve the above goal, after analysis, we decided to add four node types and two artifact types to the existing TOSCA base type definition included in the TOSCA Simple Profile in YAML Verion 1.0 [7]. The current TOSCA relationship types defined in the referred document are sufficient to support the relation definition between the nodes.

Table 1 describes the custom types that we need to add to the TOSCA base type definition.

Table 1 - TOSCA custom types to support network services modeling in an example CMTS System Ochestrator

Node Types	Extends		Artifact Types	
Container	tosca.nodes.Root		Image	tosca.artifacts.Root
Container.Executable	cabu.nodes.Container		Dockerfile	tosca.artifacts.Root
Software	tosca.nodes.Root			
Volume	tosca.nodes.Root			
Phyendpoint	tosca.nodes.Root			

In the table, all the custom types extend the root Data Type in the standard TOSCA specification. The detailed definition of the tosca.nodes.Root can be found in [7]. The Container data type defines the basic meta data of the Docker container; The Docker executable container defines more data when the container is deployed in an actual cloud provider environment. The Software data type defines the actual software that runs inside a container; The Volume specifies the storage attached to the container; The Dockerfile and the Image are the two artifacts that the container uses for the software packaging and deployment; The Phyendpoint is the data type used for modeling a physical device.

Figure 7 describes the attributes of each Node Type and Artifact Type in the above table.

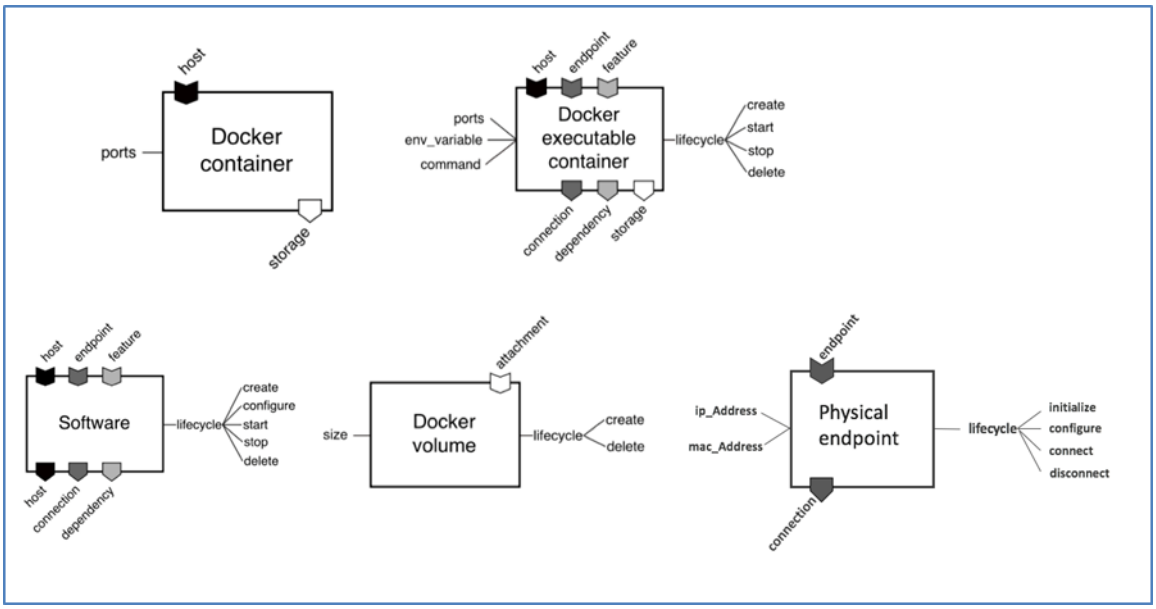

Figure 7 - Visulized TOSCA custom types for CMTS System Orchestrator

The above custom types help with the modeling of the CCMTS instances in the cloud. The Docker container and Docker executable container provide the attributes for the user to specify the ports, environment variables, and commands need to be executed when starting the container. Since the container is transient and the data needs to be stored in a Volume, the storage property of the Docker container and Docker executable container will allow the user to specify the Volume to be attached to the container. The standard lifecycle operation interfaces allow the user to plugin customized scripts to run during the lifecycle of the containers.

The Physical endpoint helps with the modeling of RPDs. Although it is not part of the workload being deployed in the cloud, we need this entity in the data model to specify the relationship between the RPDs and the CCMTS instances. The attributes and the lifecycle operations of the Physical endpoint will allow the user to uniquely identify this physical network function, and to insert customized workflow operations during the deployment of the RPD. Appendix 2 specifies the detailed TOSCA YAML definition of these custom types.

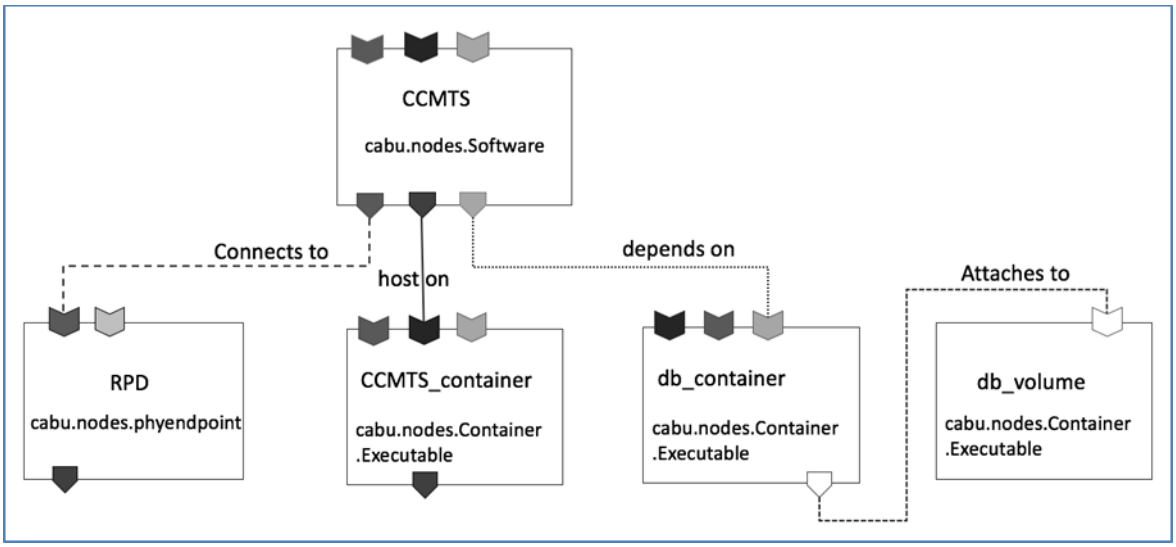

Figure 8 - The TOSCA model of an example CMTS System Orchestrator

Using the custom types, we can model the relationship between the RPDs and CCMTS instances, which is illustrated in Figure 8. In the diagram, CCMTS instance is a type of software node hosted on a CCMTS_container node with the sufficient information to spin up a microservice container in the cloud. This includes the Docker image file location for downloading during the deployment time, the runtime parameters passed to the container, the volume to store the data, which are modeled as different node and artifact instances using the custom types defined earlier. Moreover, the above data model specifies the relation between the nodes using the standard TOSCA relationship types.

The following YAML snippets give an example of the TOSCA definition of the above model. For the complete definition, please refer to Appendix 1 for more details.


```
node_templates:
  ccmts:
      type: cabu.nodes.software
      requirements:
          - host: ccmts_container
          - connection: RPD1
      interfaces:
          standard:
              create:
                  implementation: scripts/api/install.sh
                  inputs:
                      repo:    <git_repo_of_the_scripts>
                      branch:  {get_input: api_branch}
               configure:
                  implementation: scripts/api/configure.sh
               start:
                  implementation: scripts/api/start.sh
               delete:
                  implementation: scripts/api/uninstall.sh

  ccmts_container:
      type: cabu.nodes.Container.Executable
      artifacts:
          ccmts_image:
              file: ccmts:1.0
              type: cabu.artifacts.Image
              repository: <cabu_docker_hub_url>
          requirements:
              - storage:
                      node: ccmts_volume
                      relationship:
                          type: tosca.relationships.AttachesTo
                          properties:
                               location: /data/ccmts/db
  ccmts_volume:
      type: cabu.nodes.Volume
```

Figure 9 - TOSCA definition of the RPD to the CCMTS Connection in the Cloud Native Environment

After we generate the TOSCA definition of the CMTS service, we can use a TOSCA compliant orchestration engine to deploy the service into the microservices container environment. For example, if the target deployment cloud is Kubernetes managed microservices container environment, the TOSCA engine converts the service definition into Kubernetes deployment scripts and instruct Kubernetes to deploy the specified network services into the cloud.

The example described in this section is extremely simple. A real world NFV service is much more complex. For example, a CCMTS instance in the cloud will contain a set of microservice containers to realize the CNF functionality.

In this case, there are two solutions for the deployment. The first one is to model the service at CNF level without getting into the details of the microservice containers that CNF is composed of. In the data model, we rely on the life cycle operations of the CCMTS to run a script that spin up a set of microservice containers required by the CCMTS instance in the cloud.

Another solution is to rely on NFVO to convert the high level network service data model into the detailed model that contains all the microservice container definitions deployable in the cloud. Then the NFVO sends the generated detailed data model to the CNF Manager, which converts the service definition of the CCMTS into the deployment script that the target cloud understands. Then CNFM instructs the CIM to deploy the CCMTS into the cloud.

In the CNF Manager, we can either incorporate a third party TOSCA compliant orchestration engine or develop a TOSCA orchestration engine from scratch to parse the TOSCA definition based on the predefined node types and relationship types. There are open source TOSCA engines and TOSCA parsers available in the market including Cloudify, Tacker, and Open-O. [11] [12] [13]

Conclusion

Cloud Native NFV is the next wave in the telecommunication and network function virtualization space. At this early stage of Cloud Native NFV, we observe the gaps between the ETSI NFV references and this newly introduced approach in NFV. This includes:

1. ETSI NFV specification focuses on virtual appliances based solutions, and lacks the information and guidelines to support the cloud native architecture and environment.
2. ETSI NFV specification focuses on the service deployment and orchestration. A pragmatic NFV application needs to address other perspectives including service design, modeling, monitoring, and analytics.
3. Network Service design and modeling needs a standard modeling language. Current ETSI NFV compliant service design tool TOSCA is a good candidate but lacks the support for Cloud Native NFV.

This paper helps to bridge the gap between ETSI NFV and the Cloud Native NFV by identifying elements in the ETSI specification that needs to be augmented to support the microservices container environment. We also proposed a pragmatic software architecture to illustrate a Cloud Native NFV MANO system. Because using a standard modeling language for Network Service design is critical to the portability and interoperability of NFV MANO systems, we proposed using TOSCA modeling language and explained how to use its customization capabilities to support the Cloud Native NFV.

Cloud Native NFV is still in its infant stage. The purpose of this paper is to share the observations, practices, and examples in the related areas to help with the building of actual products using this approach. We believe with more and more NFV applications using the cloud native approach, more and more tools to support the NFV applications in the related areas will appear. More and more best practices discussions, guidelines, and specifications will be generated towards the eventual interoperability and standardization in the Cloud Native NFV.

Abbreviations

CIM	Cloud Infrastructure Manager
CCMTS	Cloud Cable Modem Termination System
CMTS	Cable Modem Termination System
CNF	Cloud Network Function
CNFMS	Cloud Network Function MicroService
CNFM	Cloud Network Function Manager
CNFFG	Cloud Network Function Forwarding Graph
EM	Element Manager
ETSI	European Telecommunications Standards Institute
ETSI ISG	ETSI Industry Specification Group
ETSI ISG GS	ETSI ISG Group Specification
IETF	Internet Engineering Task Force
MANO	Management and Orchestration
NFCI	Network Function Cloud Infrastructure
NFV	Network Function Virtualization
NFVI	Network Function Virtualization Infrastructure
NFVO	Network Function Virtualization Orchestrator
NS	Network Service
NSD	Network Service Descriptor
PNF	Physical Network Function
RPD	Remote PHY Device
TOSCA	Topology and Orchestration Specification for Cloud Applications
VIM	Virtualized Infrastructure Manager
VM	Virtual Machine
VNF	Virtualized Network Function
VNFD	Virtualized Network Function Descriptor
VNFM	Virtualized Network Function Manager
VNFFG	Virtualized Network Function Forwarding Graph
AP	access point
bps	bits per second
FEC	forward error correction
HFC	hybrid fiber-coax
HD	high definition
Hz	hertz
ISBE	International Society of Broadband Experts
SCTE	Society of Cable Telecommunications Engineers

Bibliography & References

[1] *Microservices architecture in the Telco Cloud*, SDX Central; Available from https://www.sdxcentral.com/nfv/definitions/microservices-architecture-telco-cloud/

[2] Network Function Virtualization – Introductory White Paper; Available at https://portal.etsi.org/nfv/nfv_white_paper.pdf

[3] *Network Function Virtualization Management and Orchestration Group Specification;* European Telecommunications Standards Institute (ETSI) Group Specification;2014; Available from http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf

[4] *OpenNetwork Automation Platform*, ONAP Wiki; Available from

https://wiki.onap.org/display/DW/Architecture

[5] *TOSCA vs. Netconf – a Comparison*, SDX Central; Available from https://www.sdxcentral.com/nfv/definitions/tosca-vs-netconf-comparison/

[6] *Making TOSCA Truly Portable*, Nati Shalom, May 12, 2016; Available from http://cloudify.co/2016/05/12/making-tosca-truly-portable-openstack-cloud-nfv-open-source-orchestration.html

[7] *TOSCA Simple Profile in YAML Verion 1.0*; Available from

http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/csprd02/TOSCA-Simple-Profile-YAML-v1.0-csprd02.html

[8] *TOSCA Simple Profile for Network Functions Virtualization (NFV) version 1.0*; Available from http://docs.oasis-open.org/tosca/tosca-nfv/v1.0/tosca-nfv-v1.0.html

[9] *Orchestrating applications with TOSCA and Docker*, Luca Rinaldi; Available from

https://core.ac.uk/download/pdf/79623650.pdf

[10] *Docker.io*; Available from https://www.docker.com/what-docker

[11] *Cloudify.co*; Available from http://cloudify.co/

[12] *Tacker – OpenStack NFV Orchestration*; Available at https://wiki.openstack.org/wiki/Tacker

[13] *OpenO.org*; https://www.open-o.org/

Appendix 1. Cloud Native CMTS System Orchestrator TOSCA Model

```
tosca_definitions_version: tosca_simple_yaml_1_0

description: TOSCA description of the CCMTS and RPD Orchestration
application.

repositories:
  docker_hub: <cabu_docker_hub_url>

imports:
    -cabu_tosca:<cabu_tosca_type_def_url>

topology_template:
  ccmts_port:
      type: integer
      default: 8080
      description: REST port

node_templates:
  ccmts:
      type: cabu.nodes.software
      requirements:
          - host: ccmts_container
          - connection: RPD1
      interfaces:
          standard:
              create:
                  implementation: scripts/api/install.sh
                  inputs:
                      repo: <git_repo_of_the_scripts>
                      branch:  {get_input: api_branch}
               configure:
                  implementation: scripts/api/configure.sh
               start:
                  implementation: scripts/api/start.sh
               delete:
                  implementation: scripts/api/uninstall.sh
```

```
ccmts_container:
    type: cabu.nodes.Container.Executable
    artifacts:
        ccmts_image:
            file: ccmts:1.0
            type: cabu.artifacts.Image
            repository: <cabu_docker_hub_url>
        requirements:
            - storage:
                    node: ccmts_volume
                    relationship:
                        type: tosca.relationships.AttachesTo
                        properties:
                             location: /data/ccmts/db
ccmts_volume:
    type: cabu.nodes.Volume

rpd1:
    type: cabu.nodes.phyendpoint
    requirements:
        -connection: ccmts
    properties:
        -macAddress: 40:00:00:00:00:04
    interfaces:
        Phyendpoint:
            initialize:
                implementation: scripts/api/initialize.sh
                inputs:
                    repo: <git_repo_of_the_scripts>
                    branch:  {get_input: api_branch}
             configure:
                implementation: scripts/api/configure.sh
             connect:
                implementation: scripts/api/connect.sh
             disconnect:
                implementation: scripts/api/disconnect.sh
```

Appendix 2. Cloud Native CMTS System Orchestrator TOSCA Custom Types

```
Tosca_definitions_version: tosca_simple_yaml_1_0

Description: Definition of the custom types of cmts orchestrator

Node_types:
  cabu.nodes.Container:
      Derived_from: tosca.nodes.Root

      Attributes:
          Id:
              Type: string
          Private_address:
              Type: string
          Public_address:
              Type: string
          Ports:
              Type: map

      Properties:
           Ports:
              Type: map
              Required: false

       Requirements:
            -storage:
                  capability: tosca.capabilities.Attachment
                  occurences: [0, UNBOUNDED]
                  node: cabu.nodes.Volume
                  relationship: tosca.relationships.AttachesTo

       capabilities:
           host:
               type: tosca.capabilities.Container
               valid_source_types: [cabu.nodes.Software]
               occurances: [0, UNBOUNDED]
```

```
cabu.nodes.phyendpoint:
  derived_from: tosca.nodes.Root

  attributes:
    mac_address:
      type: string
    ip_address::
      type: string

  properties:
     mac_address:
      type: string
      required: yes
     ip_address:
      type: string
      required: false

   requirements:
     -connection:
        capability: tosca.capabilities.Endpoint
        occurences: [0, UNBOUNDED]
        node: tosca.nodes.Root
        relationship: tosca.relationships.ConnectsTo

   capabilities:
```

```
  cabu.nodes.phyendpoint:
      derived_from: tosca.nodes.Root

      attributes:
          mac_address:
              type: string
          ip_address::
              type: string

      properties:
           mac_address:
              type: string
              required: yes
           ip_address:
              type: string
              required: false

       requirements:
            -connection:
                  capability: tosca.capabilities.Endpoint
                  occurences: [0, UNBOUNDED]
                  node: tosca.nodes.Root
                  relationship: tosca.relationships.ConnectsTo

       capabilities:
               endpoint:
                   type: tosca.capabilities.Endpoint
                   valid_source_types: [cabu.nodes.Software,
cabu.nodes.Container.Executable]]
                    occurances: [0, UNBOUNDED]

        interfaces:
               cabu.interfaces.node.lifecycle.phyendpoint:
               description:
                  this interface defines the lifecycle operations related to
the physical endpoints
               initialize:
                   description: lifecycle initialization operation for
physical endpoints
               configure:
                   description: lifecycle configuration operation for
physical endpoints
               connect:
                   description: lifecycle connect operation for physical
endpoints
               disconnect:
                   description: lifecycle disconnect operation for physical
endpoints
```

