POLYTECHNIC OF TURIN

Master Degree Thesis in Computer Engineering

Prototyping a Network Service based on the ONAP Platform

Supervisors
Prof. Fulvio Giovanni Ottavio Risso
PhD. Francesco Lucrezia

Candidate
Tommaso Guario

Academic Year 2018-2019

Summary

In recent years, the telecommunications sector is witnessing an exponential growth in the number of devices constantly connected to the network and in the systematic usage of cloud computing by the Network Service Providers (NSP) for reducing costs while at the same time providing better and/or new services; this requires the need for a new way to manage networks. The main approach that is emerged in last years is to exploit new technologies such as Software Defined Networking (SDN) and Network Function Virtualization (NFV) to build a dynamic, flexible and above all reliable network model that allow the management, configuration and automation of highly-available and scalable network services. The SDN is an architecture for creation of telecommunications networks in which the network control plane and the data transport plane are logically separated. This logical separation allows on the one hand the possibility to manage the entire network via a single software controller, thus ensuring greater scalability, higher standards of reliability and network security, and on the other hand the possibility to use devices, produced by multiple vendors, that no longer contain the management functions within them, thus allowing the emergence of a dynamic network that is no longer linked to the very large number of different protocols currently used. Instead, NFV is the process that aims to virtualize the network functionalities performed by physical devices in elementary blocks that can be interconnected to implement communication services. A virtualized network function (VNF) consists of one or more virtual machines that manage different software and processes on standard servers, memory devices or even on a cloud infrastructure, instead of using different hardware devices for each network function. The NFV is important both to reduce the cost of network nodes (due to the lower complexity required or in extreme cases by eliminating the need for the physical node itself) and to integrate it into an SDN-type network management, a context in which NFVs can be used together.

Despite the rapid adoption of these cloud technologies, there is still lack of a single system able to design and offer on-demand cloud services, in a simple and fast way, and also manage the whole network infrastructure, from physical devices to datacenters, cloud environments and domain controllers. For these reasons, in 2017 a new project,

ONAP, is born with the purpose of being a comprehensive platform for real-time, policy-driven orchestration and automation of physical and virtual network elements that will enable software, network, IT and cloud providers and developers to rapidly automate new services and support complete lifecycle management.

The ONAP platform is of great interest to service providers for both capabilities and simplification that it introduces in the network infrastructure management and for the cost reduction that it would introduce in the provisioning of services to customers. ONAP exploits SDN and NFV to orchestrate physical/virtual network functions on a global scale (multi-site and multi-VIM) and instantiates network elements and services dynamically in closed loop processes able to receive external events and reacting in real time. In essence, ONAP is the platform above the infrastructure layer that automates the network. ONAP allows end-users to connect products and services through the infrastructure, deployments of VNFs and scaling of the network in a fully automated manner. The high-level architecture of ONAP contains different software subsystems that are part of a design-time environment, as well as an execution-time environment to provide automated instantiation of a network service when needed and managing service demands in a dynamic way.

In this dissertation we present the work done to create a prototype network service using the features offered by the ONAP platform. In particular, first we analyze the internal ONAP architecture, the main components used and the way in which they interact together to offer ONAP services, and the several installation procedures tested to setting up ONAP in a cloud environment; then we describe in detail our service prototype built on ONAP, which can be deployed as often as a customer requires it. This service is designed to provision end-to-end connectivity in real networks composed by heterogeneous devices and links within a single authoritative domain: it offers a layer 2 connection between a customer, which could be a single host as well as a branch office, and a service of any type (such as a firewall, a load balancer, etc.) hosted in a datacenter, usually located in the Service Provider's Central Offices. Our service enables the management of the whole network infrastructure, both by configuring the physical devices that provide the connection between the datacenter up to the customer premise and by instantiating, inside datacenter, the VNF that will run the service to which the customer will access.

Acknowledgements

A very demanding journey ends, which I hope will give me great satisfaction in the future.

I would like to thank my supervisor Prof. Risso who gave me the right advices to reach the goal of the Master's degree.

I thank my second supervisor PhD. Francesco, Ing. Aniello and the TIM team with whom I worked closely for the realization of this thesis project.

Finally, a special thanks goes to my family and friends who supported me, in the good and in the bad, during this adventure.

Contents

List of Figures

Chapter 1

Introduction

1.1 Context and Motivation

New technologies are transforming the way in which service providers and businesses develop, deploy and scale their next-generation networks and services. The new approach that is maturing in recent years is to adopt a dynamic, flexible and above all reliable network model, able to adapt to the changes of the future without requiring major maintenance efforts or the installation of additional hardware by the network operators. A network with these features can be developed thanks to an innovative architectural model such as Software Defined Networking (SDN) and a new way to exploit the functionality of devices such as Network Function Virtualization (NFV). These two concepts are closely linked to each other and can entail particular advantages if applied simultaneously, but are in themselves independent.

- *Software Defined Networking (SDN)* [1] is an innovative approach to design and develop a telecommunications network. It is proposed to simplify the traditional work of IP and Ethernet networks, replacing the distributed control logic with a centralized control logic. The SDN splits the network control plan from the forwarding plan, removing the first from the network devices and centralizing it. Network devices will maintain forwarding functions and can be properly programmed by a central network controller, using common languages and APIs. The network controller, known as an SDN controller, also features northbound APIs for programming different applications based on their respective communication methods. This software-driven approach has the effect of simplifying network devices, which can be implemented with common, low-cost hardware tools and, at the same time,

[1] Software-Defined Networking: A Comprehensive Survey [1]

allows greater flexibility and ease of network management

- *Network Function Virtualization (NFV)* [2] is the process of virtualizing network functionalities performed by physical telecommunications equipment. The major advantages that the network operator can derive from the NFV usage derive substantially from the fact that they are no longer bound to the hardware (switches, servers, storage devices, etc.) necessary to introduce new network services. This involves a whole series of benefits in economic terms and the time reduction to market thanks to the virtualization process

The synergy of SDN and NFV solutions allows the network to achieve the best performance. In fact, the SDN provides the NFV with the advantages of a programmable connection between virtualized network functions; the NFV, on the other hand, makes available to the SDN the possibility of implementing the network functions through software on COTS (Commercial off-the-shelf) servers. Thus, it is possible to virtualize the SDN controller by implementing it on a cloud that can be easily migrated to any location based on the needs of the network.

In this context, operators of large networks are challenged to keep up with the size and costs of the manual changes necessary to implement new service offerings. Many are trying to exploit SDN and NFV to improve service speed, simplify interoperability and equipment integration, and reduce the overall costs of CapEx and OpEx.

1.2 Scenario

Nowadays, datacenters and cloud computing are at the center of modern software technology, providing more flexibility, better performances and the capabilities to store, manage, and process data. Within datacenters, network infrastructure and compute resources can be virtualized, eliminating the need to purchase and maintain expensive hardware devices, and offered as cloud services to datacenter users. For these reasons several new frameworks are born to simplify the management and the networking of datacenters such as Openstack and Kubernetes. Openstack is useful for datacenter orchestration and cloud resources deployment while Kubernetes aims to simplify deployment and scalability of containerized applications into clustered hosts, on which containers are executed. These technologies have different purpose but can be used together to provide a scalable, efficient and easy to manage cloud environment.

[2] Network Function Virtualization: State-of-the-art and Research Challenges [2]

Despite the increasing usage of cloud technologies, from service provider's perspective, there is still shortage of a single system able to design and offer on-demand cloud services, in a simple and fast way, and also manage the whole network infrastructure, from physical devices to datacenters, cloud environments and domain controllers. For these reasons, in the last few years a new project, ONAP, is born with the aim of presenting itself as solution to this lack, providing a comprehensive platform for real-time, policy-driven orchestration and automation of physical and virtual network elements that will enable software, network, IT and cloud providers and developers to rapidly automate new services and support complete lifecycle management.

The ONAP platform is of great interest to service providers both for capabilities and simplification that it proposes to offer in the network infrastructure management and for the cost reduction that it would introduce in the workplace. In fact, all work described in this thesis is carry out in collaboration with TIM company that is working on the ONAP project and has made its testbed environment available for lab trials.

1.3 Goals

Thesis's main purpose is to study the ONAP platform in order to understand its architecture and capabilities offered to service providers and then, over ONAP, create a network service that can be deployed on the fly to customers who request it.

ONAP is a complex framework composed of many components that interact together to allow the entire lifecycle management of a service, from the design to the distribution and execution phases. It can be partially/completely installed in a cloud environment following different installation methods, depending on the available testbed and the necessary requirements. Furthermore, the open-source nature of the ONAP project makes it possible to have free access to all the code and to customize the platform and its components to achieve specific objectives.

The service developed on ONAP is a prototype of an EVPL (Ethernet Virtual Private Line) service that offers, for customers who request it, a layer 2 connection to a service of any type (such as a firewall, a load balancer, etc.) hosted in a datacenter, usually located in the Service Provider's Central Offices. This EVPL service is an extension of a use-case previously developed in TIM's labs, called E-CORD. The latter creates on-demand Ethernet Virtual Circuits between different locations over metro and wide area networks by using a hierarchy of SDN controllers for the management and configuration of the physical network infrastructure. For this we have integrated the E-CORD architecture

within ONAP so that it becomes part of the EVPL service and provides a communication channel between provider's and customer's sites. Moreover, the EVPL service also manages the datacenter in which the service requested by the customer will run and the related network infrastructure.

The main advantage of using ONAP as a global orchestration manager consist of having a single entry point to (1) design, compose and deploy EVPL service instances on demand, (2) manage, scale and monitor the service instances lifecycle and (3) control the underlying network infrastructure from customer to service provider by using ONAP's default modules or by adding custom software modules that perform the required logic.

1.4 Thesis Organization

The thesis's structure is organized as follows:

- **Chapter 1 Introduction** in the first chapter we analyze the scenario in the cloud environment technologies and the birth of new platform, ONAP, designed specifically for service providers and network infrastructure management
- **Chapter 2 ONAP: Open Network Automation Platform** in chapter two we describe in detail the ONAP project, its architecture and the main components that compose the platform
- **Chapter 3 ONAP workflow** in chapter three we present how the ONAP platform works, from the design and composition of a generic service to the execution phase, and the main interactions between components
- **Chapter 4 ONAP installation** in chapter four we discuss two different approch for installing ONAP and all the technologies and tools studied and used
- **Chapter 5 EVPL service implementation in E-CORD** in chapter five we present the E-CORD service, its architecture and the service implementation in TIM testbed
- **Chapter 6 EVPL Service implementation using ONAP** in chapter six we discuss the EVPL service implementation based on the ONAP platform describing all components that make up the service
- **Chapter 7 ONAP current & future directions** in chapter seven we discuss the future developments of the ONAP project and the work evolution in TIM labs

- **Chapter 8 Conclusion and Future work** in chapter eight we discuss the future work of the EVPL Service implementated using ONAP and its future updates

Chapter 2

ONAP: Open Network Automation Platform

2.1 Overview

The Open Network Automation Platform (ONAP) project [3] is an initiative that aims to offer the ability to design, create, orchestrate and manage physical and virtual network services bringing greater flexibility and lower costs. ONAP is an open source networking project of the Linux Foundation born in March 2017. It is the result of the union of the two main orchestrating and networking projects, open source ECOMP (Enhanced Control, Orchestration, Management & Policy) of AT&T and OPEN-O (Open Orchestrator).

ONAP has the capability to orchestrate physical and virtual network functions on a global scale (multi-site and multi-VIM). It facilitates the usability of the platform by providing a set of open and interoperable Northbound REST APIs and by supporting the modeling of YANG [1] and TOSCA [2] data. Its modular and stratified nature improves interoperability and simplifies integration, enabling to support multiple environments for the management of VNF by integrating with different VIM, VNFM, SDN orchestrators and even common equipment.

The ONAP platform allows to instantiate network elements and services dynamically in closed loop processes capable of receiving external events and reacting in real time. This approach is based on the use of three main logical components:

- *a design framework* that allows to define a service in every aspect, from the modeling

[1] YANG data model, Data Modeling Language for the Network Configuration Protocol (NETCONF). More detail in section 3.1.3

[2] TOSCA is a data model standard that can be used to orchestrate network functions virtualization (NFV) services and applications. More detail in section 3.1.2

of resources and relationships to the specification of the rules that guide the behavior of the service by defining the applications, analyzes and closed-loop events necessary for a dynamic management of the service

- *an orchestration and control framework* that allows the instantiation and management of new services when necessary
- *an analysis framework* that monitors the behavior of the service during its life cycle, based on the specifications defined in the design phase, and communicates with the orchestration and control framework for the resources redefinition related to the service

To achieve these objectives ONAP integrates a portal from which it is possible to manage all the network infrastructure, from the definition of the customers to whom the services will be associated to the modeling and distribution phase of a service and its components.

ONAP is in continuous development, thanks to the work of the community of collaborators working on the project, and releases a new version of the framework every six months. For the work done in this thesis the first version was used, Amsterdam, which allows to interact only with one of the main cloud management software, Openstack.

2.2 Architecture

ONAP is composed of many software subsystems, which are part of the two main architecture frameworks:

- **Design-time framework:** an environment for defining, designing and programming the platform
- **Run-time framework:** an environment for executing the logic defined in the design phase

In the next sections we give an overall view of the ONAP's component but for the project developed in this thesis only some of the components have been used. The figure below shows the ONAP Amsterdam architecture and the components, highlighted in yellow, actually used.

Access to Design-time and Run-time frameworks is provided by the graphical interface of the ONAP portal and by the command line (CLI).

The ONAP portal allows centralized management to access the framework through multiple accounts depending on the role to play (designer, tester, governor, operational,

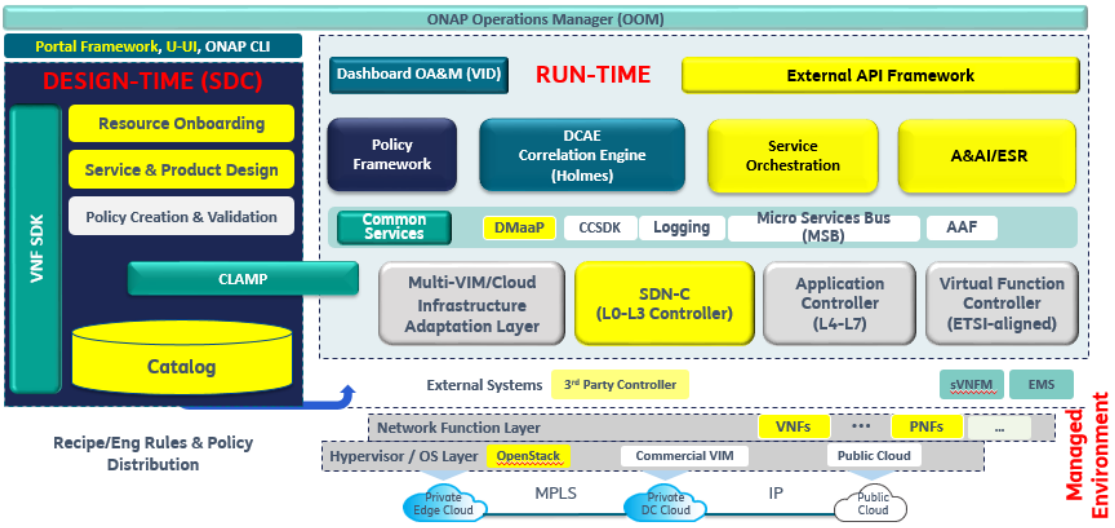


Figure 2.1: Amsterdam architecture

admin), which can be configured within the portal itself. ONAP user interfaces are intended for users in variety of roles:

- System and network administrators who need to instantiate, manage, and monitor Resources, Services, and Products on an existing ONAP system
- ONAP administrators who create user accounts, assign roles, and install applications within ONAP
- Providers of the several assets managed by ONAP :
  - Vendors who need to create and integrate ("onboard") low-level Resources, such as VNFs (virtual network functions) or other single-purpose functions
  - Service designers who need to compose complex Services from Resources
  - Testers/Approvers who need to test and certify Resources and Services before they are added to the ONAP catalog
  - Product managers who need to define Products from Services (Products include billing and customer support definitions for external Business Support Systems)

In this way it is possible to separate all the phases of design, development, testing and release, from the definition of the customer and the license associated with a service to its design, verification, approval and distribution. From the portal, the administrator can load and manage applications and widgets, and manage user access; users can access

applications already in the framework such as SDC, Policy, A&AI UI, CLI.
In addition, the portal provides an SDK to facilitate the development of new applications by exploiting the systems in the framework (services, APIs, widgets).

2.3 Design-time framework

The design framework is a development environment composed of applications, operating modes, repositories for the description and definition of resources, services and products.

The design time framework facilitates reuse of models, further improving efficiency as more and more models become available. Resources, services, products, and their management and control functions can all be modeled using a common set of specifications and policies for controlling behavior and process execution. Process specifications automatically sequence instantiation, delivery and lifecycle management for resources, services, products and the ONAP platform components themselves. The design framework consists of the following subsystems:

- **SDC:** Service Design and Creation is the graphical application of ONAP for modeling and design
- **Policy Creation:** is a subsystem that maintains, distributes, and operates on the set of rules that underlie ONAP's control, orchestration, and management functions
- **CLAMP:** Closed Loop Automation Management Platform, is a platform for designing and managing control loops. It is used to design a closed loop, configure it with specific parameters for a particular network service, then deploying and undeploying it. Once deployed, the user can also update the loop with new parameters during runtime, as well as suspending and restarting it
- **VNF SDK:** VNF Software Development Kit, is a development tool designing and uploading new VNFs inside the SDC

2.3.1 Service Design and Creation

SDC is the ONAP visual modeling and design tool. It creates internal metadata that describes assets used by all ONAP components, both at design time and run time.

The SDC manages the content of a catalog, and logical assemblies of selected catalog items to completely define how and when VNFs are realized in a target environment. A complete virtual assembly of specific catalog items, together with selected workflows and instance configuration data, completely defines how the deployment, activation, and life-cycle management of VNFs are accomplished. Selected sub-assemblies may also be

represented in the catalog and may be combined with other catalog items, including other sub-assemblies.

In the context of a catalog containing TOSCA nodes that are assembled to form a TOSCA blueprint within the SDC, it is expected that such nodes convey (as properties) all of the workflows and/or workflow fragments needed to realize the node in some specific target environment. In this way, any assembly of such nodes, defined in a blueprint, can be used to define the end-to-end workflow needed to realize the VNF associated with the TOSCA blueprint. All TOSCA informations provided to the SDC are divided into different entities and stored as vertices of a graph; the relationships between these entities, and their logical connections are stored as links. The SDC uses Titan Graph DB to create the graph while persistence is provided by the Cassandra database.

SDC manages two levels of assets:

- **Resource:** implemented either entirely in software, or as software that interacts with a hardware device. Each Resource is a combination of one or more Virtual Function Components (VFCs), along with all the information necessary to instantiate, update, delete, and manage the Resource. A Resource also includes license-related information. There are three kinds of Resource: *Infrastructure* (the Cloud resources, e.g., Compute, Storage); *Network* (network connectivity functions and elements); *Application* (features and capabilities of a software application, such as a load-balancing function).

- **Service:** a well formed object comprising one or more Resources. Service Designers create Services from Resources, and include all of the information about the Service needed to instantiate, update, delete, and manage the Service

The definitions of assets include Information Artifacts and Deployment Artifacts. Information Artifacts are provided by the vendor of an asset such as a VNF; they describe characteristics of the asset. Some of these artifacts are supporting documents intended for human readers only, whereas others contain data that will be imported into the ONAP environment when the asset is onboarded.

Once assets are on-boarded, the information provided by the vendor is translated into SDC internal resource models. The service provider will use SDC to further enrich the resource model to meet the provider's environment, and additionally compose resources into service models. The model includes not only the description of the asset but also references to ONAP functions needed for lifecycle management of the asset. The tested models will then be distributed to the ONAP execution environment as Deployment Artifacts.

The Deployment Artifacts include the asset definition (a Resource or Service) with instructions to ONAP for creation and management of an instance of the asset in the network. Currently, SDC imports and retains information from Heat Templates for cloud infrastructure creation, YANG XML files for state data manipulated by the Network Configuration Protocol, TOSCA files for specifying cloud infrastructure, and certain vendor provided scripts. In the future, SDC may import BPMN [3] flows files for specifying business processes and their interconnections in a service-oriented architecture.

The SDC consists of three main components:

- **Catalog:** is the repository for resources and services. These elements are added to the catalog using Design Studio
- **Design Studio:** is used to create, modify, and add resources and services to the Catalog
- **Distribution Studio:** is used to distribute information about resource and service models to the several components of the execution environment

In addition, the SDC integrates Jersey, a RESTful Web Services framework, to expose a set of APIs to the outside used by ONAP components.

Distribution Flow

For the deployment phase the SDC communicates with the execution environment using the DMaaP component, Data Movement as a Platform. The SDC defines two objects (topic) within the DMaaP, one to publish notifications and one to read notifications about the status of an event. It also provides a client (SDC Distribution Client) that allows the management of communications with the DMaaP and perform several operations: register to a topic, receive notifications about a topic, send notifications about the status of an event, define and download only the artifacts of interest, delete registration from a topic. The components that integrate the distribution client are: A& AI, SDN-C, MSO, DCAE and APP-C. Once a service is deployed, the SDC sends a notification to a topic with information about the artifacts related to the service, including the name, type, URL for the download and a checksum to validate the integrity once downloaded . All components registered to that topic receive a notification, identifying which artifacts

[3] Business Process Model and Notation (BPMN) is a graphical representation for specifying business processes in a business process model. More detail in section 3.1.5

to download based on their type, and contact the SDC for download via REST API. Once the artifacts are downloaded, the application sends a status notification to inform the SDC.

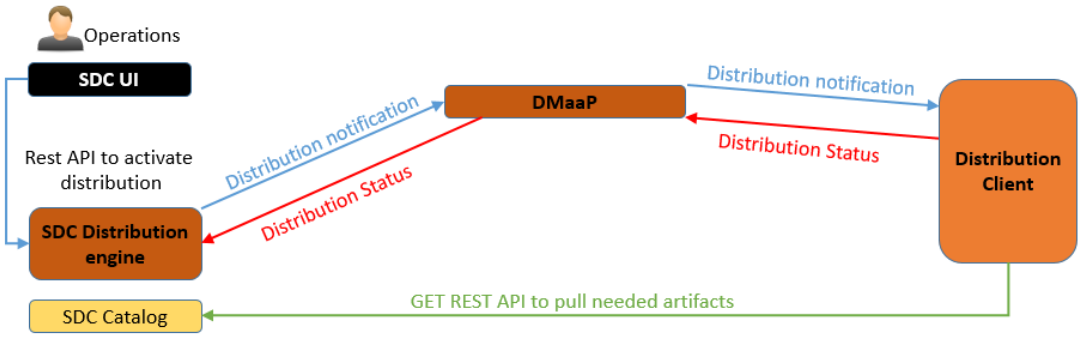


Figure 2.2: Distribution Flow

Runtime components can choose whether to receive specific types of artifacts or to receive the whole TOSCA package (CSAR [4] file). In the case of single artifacts each application implements its own logic to read and manage them, while for the CSAR file there is a specific module. This module, the SDC TOSCA parser, analyzes the CSAR and retrieves the artifacts contained in it, required by the application.

2.4 Run-time framework

The Run-time framework executes the rules and policies defined and distributed by the Design-time framework. This allows the distribution of criteria and models among the ONAP modules such as the Master Service Orchestrator (MSO), the Controllers (SDN-C, APP-C, VF-C), the DCAE (Data Collection, Analytics and Events) and the A&AI (Active and Available Inventory). These components use a set of common services that provide support for logging, access control and data management. The VID (Virtual Infrastructure Deployment) application allows users to instantiate services, with their components, and to perform infrastructural change operations such as resizing and updating software of existing VNF instances. In addition, the External API framework provides a standard interface between the Business Support System (BSS) and the several ONAP components (MSO, A&AI, SDC, etc.). This provides an abstract view of

[4] Cloud Service Archive: is an archive defined by the OASIS TOSCA standard. It is a compressed file that includes a TOSCA model of a network service and all the necessary scripts or files that a VNF needs for its lifecycle from creation to termination.

the platform, facilitating the integration of an operator's existing BSS/OSS environment without excessive costs.

2.4.1 Master Service Orchestrator

The Master Service Orchestrator (MSO) component executes the specified processes by automating sequences of activities, tasks, rules and policies needed for on-demand creation, modification or removal of network, application or infrastructure services and resources. MSO provides orchestration at a very high level, with an end-to-end view of the infrastructure, network, applications, and facilitates additional orchestration that takes place within underlying controllers. It also marshals data between the several controllers so that the process steps and components required for execution of a task or service are available when needed. The MSO's primary function is the automation of end-to-end service instance provisioning activities. MSO is responsible for the instantiation and release, and subsequent migration and relocation of VNFs in support of overall end-to-end service instantiation, operations and management. MSO executes well-defined processes to complete its objectives and is typically triggered by the receipt of service requests generated by other ONAP components, by external APIs or by BSS/OSS. The orchestration procedure is obtained from the Service Design and Creation (SDC) component, where all service designs are created and exposed/distributed for consumption. MSO runs autonomously within ONAP and the orchestration engine is a reusable service.
Any component of the architecture can execute process workflows. The service model maintains consistency and reusability across all orchestration activities and ensures consistent methods, structure and version of the workflow execution environment. Orchestration processes interact with other platform components or external systems via standard and well-defined APIs. Controllers (Network and Application) participate in service instantiation and are the primary players in ongoing service management; for example, control loop actions, service migration and scaling, service configuration, and service management activities. Each controller instance supports some form of orchestration to manage operations within its scope.
In future releases Orchestration process flows will be defined in the Service Design and Creation subsystem (SDC). These process flows start with a template that may include common functions such as homing determination, selection of Infrastructure, network and application controllers, consultation of policies and interrogation of Active and Available Inventory (AAI) to obtain information needed to guide the process flows. MSO does not provide any process-based functionality without a workflow for the requested activity: in the Amsterdam release the process flows are designed directly in MSO using BPMN.

MSO interrogates AAI to obtain information regarding existing Network and Application Controllers to support a service request. AAI provides the addresses of candidate controllers that are able to support the service request.

MSO may then interrogate the controller to validate its continued available capacity. MSO and the controllers report reference information back to AAI upon completion of a service request to be used in subsequent operations. As previously stated, orchestration is performed by several components, primarily the MSO and the Application and Network Controllers. Each will perform orchestration for: service delivery or changes to an existing service; service scaling, optimization, or migration; capacity management. Regardless of the focus of the orchestration, all workflows must include steps to update AAI with configuration information, identifiers and IP Addresses.

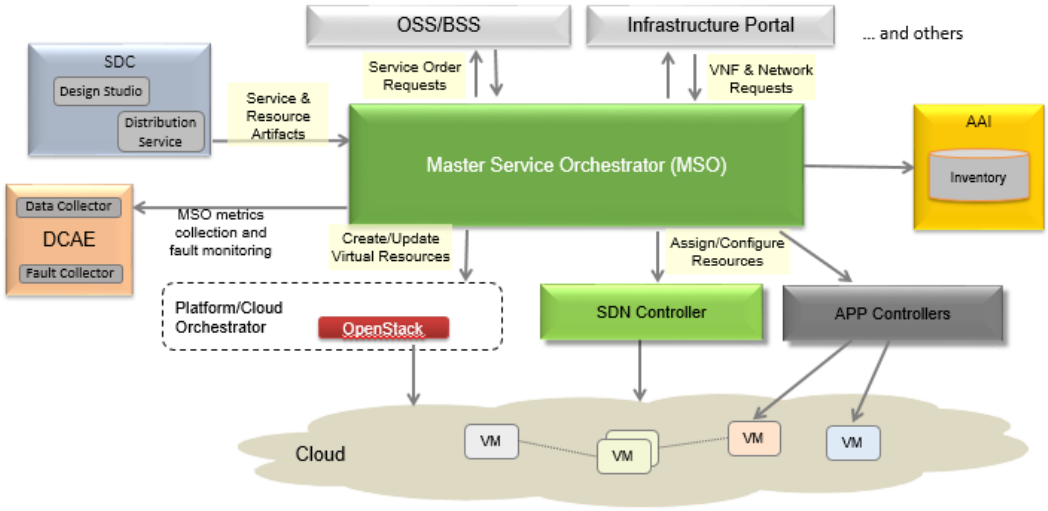


Figure 2.3: High Level Architecture and Interfaces

Furthermore MSO interacts with Openstack cloud platform for instantion of virtual resources using well-defined Openstack APIs, primarily to communicate with Heat and Keystone components.

Network Controller Orchestration

MSO obtains compatible Network Controller information from AAI and in turn requests LAN or WAN connectivity to be established and configured. This may be done by requesting the Network Controller to obtain its resource procedure from SDC. It is the responsibility of MSO to request (virtual) network connectivity between the components and to ensure that the selected Network Controller successfully completes the network

configuration workflow. A service may have LAN, WAN and access requirements, each of which must be included in the procedure and configured to meet the instance specific customer or service requirements at each level. Physical access might need to be provisioned in the legacy provisioning systems prior to requesting MSO to instantiate the service.

Application Controller Orchestration

MSO sends requests to Application Controllers to obtain the application-specific component of the service procedure from SDC and execute the orchestration workflow. MSO ensures that the Application Controller successfully completes its resource configuration as defined by the procedure. As with Network Controllers, all workflows, whether focused on instantiation, configuration or scaling, will be obtained or originate from SDC. In addition, workflows also report their actions to AAI as well as to MSO.

Moreover not all changes in network or service behavior are the result of orchestration. Policies and rules (in the Policy subsystem) inform the Controller such that it can enable service behavior changes.

2.4.2 Active and Available Inventory

Active and Available Inventory (AAI) is the ONAP subsystem that provides real-time views of system's resources, services, products and their relationships with each other. AAI (sometimes referred to as A&AI) not only forms a registry of active, available, and assigned assets, it also maintains up-to-date views of the multidimensional relationships among these assets, including their relevance to different components of ONAP.

In addition to inventory and topology management, AAI provides the ability to do inventory administration. Data in AAI is continually updated in real-time by the controllers as they make changes in the network environment. Because AAI is metadata-driven, new resources and services can be added quickly with Service Design and Creation (SDC) catalog definitions, using the AAI model loader, thus eliminating the need for lengthy development cycles. In addition, new inventory item types can be added quickly through schema configuration files.

AAI provides standard APIs to enable queries for inventory and topology. Queries can be supported for a specific asset or a collection of assets.

The AAI subsystem uses graph data technology to store relationships between inventory items. Graph traversals can then be used to identify chains of dependencies between items. Relationships captured by AAI include "top-to-bottom" relationships such as those

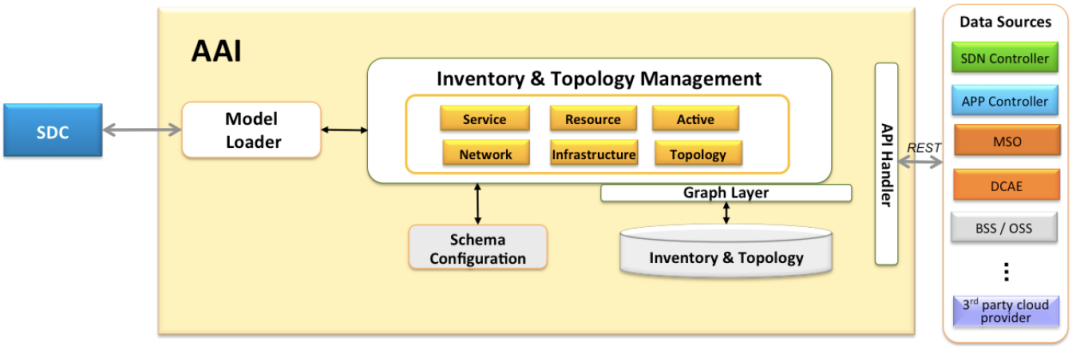


Figure 2.4: AAI functional diagram

defined in SDC when products are composed of services, and services are composed of resources. It also includes "side-to-side" relationships such as end-to-end connectivity of virtualized functions to form service chains.

AAI's data model is composed of vertexes and edges with their attributes. There are two kind of edge: parent/child and cousin. Parent/child edge refers to nesting of node types and re-use of node types, which means a node type can belong to more than one parent node type; while cousin edge means that several types can be related to each other in multiple ways.

AAI Provides a UI front-end, nick-named "Sparky", which allows users to view the graph of actual instance objects (generic-vnfs, service instances, pnfs, l3-networks, etc) and analyze data. This model visualization exposes the model to designers or operators to display graphically the set of node types and the relationships between them.

2.4.3 Network Controller

The SDN Controller (SDN-C) is the entry point for network management and control in ONAP. It manages the state of a network resource, in terms of configuration and instantiation, and is the primary agent in ongoing management, such as control loop actions, migration, and scaling. SDNC knows the network resource type and its related properties from the TOSCA CSAR file created in SDC. Based on those information it will retrieve network parameters from external component(s).

SDNC is based on Opendaylight [5] controller framework that supports a model driven service abstraction layer (MD-SAL), api handlers, operational and configuration trees,

[5] OpenDaylight (ODL) is a modular open platform for customizing and automating networks of any size and scale [4]

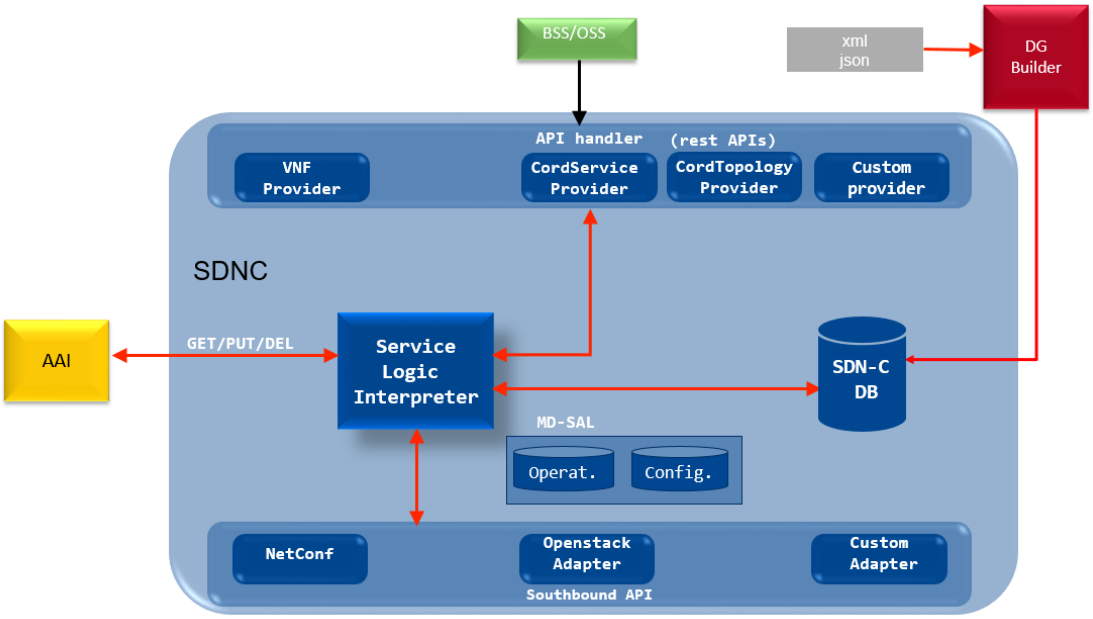


Figure 2.5: SDNC Network Controller

and an adapter framework for integrating with controlled devices, virtual functions, and cloud infrastructure. Within this framework the Service Logic Interpreter (SLI) addition provides an extensible scripting language for expressing service logic through a Directed Graph (DG) builder. SLI can be extended by adding Java classes that can be called as a node in a DG to support frequent complex operations. SLI is in charge to check if a DG exists, get it from the SDNC Database and execute it. All DGs are stored in SDNC DB.

The northbound interface is composed of several providers that expose a set of RESTCONF APIs. The API handlers is the component responsible to handle received API requests and delegate the execution workflow to SLI. Those APIs are used to interact with different components: DMaaP, for event notifications; SDC, for the distributions phase (receive and consume TOSCA models); MSO, for orchestration workflow logic; control loop applications. Northbound APIs are modelled via YANG and YANG-generated Java classes are available to parse xml files. These files are stored in the MD-SAL LogicalDataStore.

Southbound interface is made up of Java plugins (adapters) to interact with the underlying domains/networks with different protocol/mechanism. The DG workflow defines the application logic and one or more plugins can be called. Provider and plugin are uploaded in SDNC as bundle in the Opendaylight environment.

In addition, SDNC reports the status of each workflow execution to both the Active and Available Inventory and the Master Service Orchestrator.

2.4.4 Application Controller

The Application Controller (APP-C) is responsible for handling the Life Cycle Management (LCM) of Virtual Network Functions and is based, like SDNC, on Opendaylight controller framework. APP-C performs actions such as controlling, modifying, starting or stopping virtual applications and/or their components. A virtual application is composed of a maximum of four layers: Service, Virtual Network Function (VNF), Virtual Network Function Component (VNFC), Virtual Machine (VM). A Life Cycle Management command may affect any number of these layers.

The APP-C Provider module exposes the endpoints for each action supported by APP-C. This module uses the YANG model to define the YANG Remote Processing Call (RPC) and data model, in other words, the input and output parameters for each action. The Provider module is responsible for validating the RPC input and for rejecting any malformed input. After successful validation, the APP-C Provider calls the Dispatcher to continue the request processing. The APP-C Dispatcher component processes requests received by the API Request Handler from other ONAP components such as MSO, DCAE, and the Portal. The Dispatcher checks the conditions are sufficient for performing the request and selects the correct Direct Graph (DG) workflow for execution, or rejects the request. When the DG execution is complete, the Dispatching function is responsible for notifying the initiator of the operation with the request execution result (Success/Error) and updates the VNF state in Active and Available Inventory. The SLI framework is responsible for executing Directed Graphs (DGs). The Dispatcher invokes the SLI framework to execute a specific DG based on the input action. The SLI executes the DG and returns a success or failure response to the caller. APP-C can use several adapters to connect to VNFs. The IAAS adapter is provided with the ODL platforms and is the southbound adapter for APP-C. It connects with the OpenDaylight controller to perform several operations on VNFs such as restart, migrate, rebuild etc. The IAAS Adapter is effectively used as a DG plugin in that the services exposed by the adapter are called from DGs.

2.4.5 Data Movement as a Platform

Data Movement as a Platform (DMaaP) is a platform for high performing and cost effective data movement services that transports and processes data from any source to any target with the format, quality, security, and concurrency required to serve the business and customer needs.
DMaaP has four components:

- **Message Router (MR):** is a reliable, high-volume publisher/subscriber messaging

service with a RESTful HTTP API. The service is built over Apache Kafka [5].

- **Data Router (DR):** is a common framework by which data producers can make data available to data consumers and a way for potential consumers to find feeds with the data they require. The interface to DR is exposed as a RESTful web service known as the DR Publishing and Delivery API.
- **Data Movement Director (DMD):** is a client to DMaaP platform to publish and subscribe data.
- **Data Bus Controller:** provides API to create topics and grant the associated pub/sub permissions.

In ONAP Amsterdam release, only unauthenticated topics are supported. Unauthenticated topics are created upon first message publish transaction. Therefore, the only step needed is for the publisher and subscriber to agree on the topic name.

Producer components implements an HTTP client which publishes on a topic using the MR Producer API. While consumer components implements an HTTP server to receive notification messages related on topics on which they are subscribed.

2.4.6 Data Collection Analytics Events

The Data Collection, Analytics, and Events (DCAE) is the ONAP subsystem that supports closed loop control and higher-level correlation for business and operations activities. DCAE collects performance, usage, and configuration data; provides computation of analytics; aids in trouble-shooting and management; and publishes event, data, and analytics to the rest of the ONAP system for FCAPS (Fault Configuration Accounting Performance Security) functionality.

The primary functions of the DCAE subsystem are:

- collect, ingest, transform and store data as necessary for analysis
- provide a framework for development of analytics

These functions enable closed-loop responses by several ONAP components to events or other conditions in the network.

DCAE provides the ability to detect anomalous conditions in the network. Such conditions, might be, for example, fault conditions that need healing or capacity conditions that require resource scaling. DCAE gathers performance, usage, and configuration data about the managed environment, such as about virtual network functions and their underlying infrastructure. This data is then distributed to several analytic micro-services,

and if anomalies or significant events are detected, the results trigger appropriate actions. In addition, the micro-services might persist the data (or some transformations of the data). In addition to supporting closed-loop control, DCAE also makes the data and events available for higher-level correlation by business and operations activities, including BSS/OSS.

The DCAE Platform consists of several functional components: Collection Framework, Data Movement, Storage Lakes, Analytic Framework, and Analytic Applications. In large scale deployments, DCAE components are generally distributed in multiple sites that are organized hierarchically. For example, to provide DCAE function for a large scale ONAP system that covers multiple sites spanning across a large geographical area, there will be edge DCAE sites, central DCAE sites, and so on. Edge sites are physically close to the network functions under collection, for reasons such as processing latency, data transport, and security, but often have limited computing and communications resources. On the other hand, central sites generally have more processing capacity and better connectivity to the rest of the ONAP system. This hierarchical organization offers better flexibility, performance, resilience, and security.

2.4.7 Multi-VIM/Cloud

Multi-VIM/Cloud is the component that has the goal to enable ONAP to deploy, run and manage network services and VNFs on multiple virtualized infrastructure environments, for example OpenStack, Kubernetes, VMware and so on. It decouples the evolution of ONAP platform from the evolution of underlying cloud infrastructure, and minimizes the impact on the deployed ONAP while upgrading the underlying cloud infrastructures independently.

Multi-VIM/Cloud is a plugable and extensible framework that provides a Cloud Mediation Layer which includes the following functional modules:

- **Provider Registry:** to register infrastructure site/location/region and their attributes and capabilities in A&AI
- **Infra Resource:** to manage resource request (compute, storage and memory) from MSO, DCAE, or other ONAP components, so as to get VM created and VNF instantiated at the right infrastructure
- **SDN Overlay:** to configure overlay network via local SDN controllers for the corresponding cloud infrastructure
- **VNF Resource LCM:** to perform VM lifecycle management as requested by VNFM (APP-C or VF-C)

- **FCAPS:** to report infrastructure resource metrics (utilization, availability, health, performance) to DCAE Collectors

Multi-VIM/Cloud exposes a common northbound interface (NBI) of the functional modules to be consumed by other ONAP components (MSO, SDN-C, APP-C, VF-C, DCAE etc). In addition, it provides the ability to generate or extend NBI based on the functional model of underlying infrastructure. In Amsterdam release Multi-VIM/Cloud is not used and the interaction wih underlying VIM (Openstack) is handled by MSO.

Chapter 3

ONAP Workflow

ONAP is a complex platform composed of many subsystems that interact to allow new service creation and network infrastructure management. This chapter describe the steps necessary to offer these functionalities, starting from the preliminary stages of environment configuration to then move on to the service design, distribution and execution.

3.1 Service design

First of all this phase involves the definition of data structures, templates and additional software modules so that the service is properly distributed. It is important to note that ONAP is an open source framework and all the documentation and code are available. This allows service providers to use the default features that ONAP provides or to customize the platform for achieving their goals. Service design implies to define what types of data, modules and interactions are required and how modify the ONAP's framework if it is necessary. The services can be built in different ways:

- a service composed only of resources representing external network modules (physical or virtual)
- a service composed only of virtual resources in cloud environments (VIM) managed by ONAP
- a service composed of both virtual resources and resources representing external network modules

Based on service type the performed operations can be different; for example, a service composed only of virtual resources requires the loading of templates that describe their structure while a service composed only of resources representing external network modules requires to modify the SDNC with additional code so that it interacts with external

modules. ONAP provides default data structures and functionalities but it may be necessary to provide additional files and templates in the Design-time framework or modify other ONAP components by adding software modules to reach the required functionalities.

The operations, taken into consideration in this thesis, for service creation are the following:

- define Heat Orchestration Templates (HOT) to create network resources and virtual machines in Openstack
- add TOSCA models in SDC to describe new kinds of resources
- define YANG models to describe custom RPC and data models required by SDNC workflows
- preload manually network parameters into SDNC for cloud resources creation or customize SDNC to retrieve this parameters autonomously from external components
- create new BPMNs and upload them inside MSO to define custom workflow logic
- prepare APIs to manage AAI and trigger workflows in MSO and SDNC

3.1.1 Heat Orchestration Template

Heat is an Openstack service to create and orchestrate composite cloud applications using a declarative template format through an OpenStack-native REST API. A Heat template describes the infrastructure for a cloud application in text files which are readable and writable by humans, and can be managed by version control tools. Templates specify the relationships between resources (e.g. this volume is connected to this server) and this enables Heat to call out to the OpenStack APIs to create all of your infrastructure in the correct order to completely launch your application. The templates allow creation of most OpenStack resource types (such as instances, floating ips, volumes, security groups, users, etc), as well as some more advanced functionality such as instance high availability, instance autoscaling, and nested stacks. In Heat templates it is possible to specify the input parameters, defined in the parameters section of a HOT template, required from resources. This helps to make a template more easily reusable by avoiding hardcoded assumptions and allows users to customize a template during deployment.

From an ONAP's perspective, the Heat templates are used to model different kinds of VNFs that will compose a service. Each VNF can be composed from one or more cloud resourses, Virtual Function (VF) modules, and each VF module is described by its own

HOT. In this manner we can model VNFs, such as router, network and virtual machine, and provide network input paramaters during the Runtime phase.

3.1.2 TOSCA model

TOSCA [6] is a data model that can be used for creating templates or data descriptions of applications and infrastructure for cloud services. It can also be used to define the relationships among these services, as well as their operational behavior. This can happen independently of the supplier creating the service or the technology infrastructure used to deliver it. TOSCA abstracts configuration data away from specific hardware or services to make cloud services more interoperable and portable and to enable the automation of software-defined networks, in combination with NFV and clouds, to simplify end-to-end service orchestration.

TOSCA can deliver 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.

In ONAP, TOSCA is used by SDC to describe services, resources and their relationships. SDC offers the possibility of adding new TOSCA models to define new types of resources through APIs, by UI or by running custom scripts. After uploading TOSCA models, SDC stores these new resource types in the Catalog and make them available to designers.

3.1.3 YANG model

YANG [7] is a data modeling language for NETCONF and RESTCONF configuration management protocols. Together, NETCONF/RESTCONF and YANG provide the tools that network administrators need to automate configuration tasks across heterogeneous devices in a software-defined network. The YANG data modeling language provides descriptions of a network's nodes and their interactions. Each YANG module defines a hierarchy of data that can be used for configuration, state data, Remote Procedure Calls and notifications. Modules can import data from other external modules and include data from sub-modules.

RESTCONF is a REST like protocol running over HTTP for accessing data defined in YANG using datastores defined in NETCONF. It uses HTTP methods to provide CRUD (Create, Read, Update, Delete) operations on a conceptual datastore containing YANG-defined data. Request and response data can be in XML or JSON format.

In ONAP, the RESTCONF protocol is used to describe and call providers in the SDNC northbound interface from other components, like BSS/OSS and MSO. The provider Java

classes are generated from YANG model using compile tools (e.g. Maven [8]) and they are responsible to get and set xml data into the MD-SAL datastore.

3.1.4 SDNC network parameters

In the ONAP infrastructure logic, the component responsible for network management is SDNC. This implies that MSO, when executes its workflow, has to (1) call SDNC APIs to retrieve network parameters for cloud resources or (2) delegate the interaction with external network modules.

In the first case, the API to call is the preload provider's API that allow to specify network parameters (gateway address, DHCP property, ip-version, IP address, and so on) for cloud resources, such as virtual networks and VNFs. It is also possible to define a custom provider that takes care of retrieving network parameters without having to preload them manually. SDNC stores these informations in its datastore and sends them to MSO when it is required.

In the second case, the API to call is the provider's API that allow interactions with external modules. Since these modules has proprietary logics and operating modes, it is necessary to define the entire SDNC workflow logic, composed by northbound providers, DGs and plugins.

3.1.5 BPMN workflow definition

Business Process Model and Notation (BPMN) [9] is a standard for business process modeling that provides a graphical notation for specifying business processes in a Business Process Diagram, based on a flowcharting technique very similar to activity diagrams from Unified Modeling Language (UML). The objective of BPMN is to support business process management, for both technical users and business users, by providing a notation that is intuitive to business users, yet able to represent complex process semantics. Inside BPMN diagram is possible to specify the references to external scripts for execution of custom tasks. There are many system that integrate with BPMN standard for workflow and process automation, one of the most used is Camunda [10]. The latter is a Java-based framework composed of the following main components:

- **Process Engine:** a Java library responsible for executing BPMN processes
- **Camunda Modeler:** a modeling tool for BPMN diagrams
- **Camunda Cockpit:** a web application for process monitoring and operations that allows to search for process instances, inspect their state and repair broken instances

Within ONAP, BPMN and Camunda are used from MSO to execute process instances. It is also possible to model new BPMN diagrams and upload them inside MSO to perform custom workflows.

3.1.6 APIs definition

ONAP provides a large number of API interfaces to command several components, like SDC, MSO, AAI, SDNC and so on. Each component exposes a set of REST-based APIs documented in the ONAP wiki and this makes possible to perform any operation or array of operations using BSS/OSS.

Automation of a service instantiation is still missing in community ONAP Amsterdam since many APIs are to be called from an external component even for very simple services. Inside this thesis the used APIs are all standard APIs belonging to the Amsterdam release documentation.

3.2 Onboarding and Distribution

This section describe how SDC works to define and distribute a service model to the Runtime framework. The main steps to do for service creation are: VNF creation, customer definition and service composition, distribution. The VNF creation is necessary only for description and instantiation of custom cloud resources in Openstack because a service can be composed also using TOSCA nodes already present in SDC.

3.2.1 VNF creation

VNFs are composed from one or more modules (VNFCs) and these work together to perform the actions of the VNFs. A module is a subset of the resources of a VNF described in a Heat template. Each module is described in a separate Heat template and an Incremental Module is a growth or scaling unit.

In ONAP, a single VNF should be composed from one or more Heat Orchestration Templates, each of which represents a subset of the overall VNF. These component parts are referred to as “VF Modules”. During orchestration, these modules are deployed incrementally to create the complete VNF and additional incremental modules may be deployed at different times to scale portions of the VNF. All VNFs must have one base VNF module template and that module is the first one deployed: the base template. This base module must include all the shared resources of the VNF including private networks, server groups and security groups. It must also expose all shared resources by their UUID (universally unique identifier). The base module may include an initial set

of VMs and may be operational as a stand-alone minimum configuration of the VNF. A VNF may also have one or more incremental modules which define additional resources that may be added to an existing VNF and each module, base or incremental, must be described by a complete Heat template. These incremental modules should define logical growth units of the VNF.

A well-defined VNF, with all Heat templates, can be onboarded in SDC from GUI as Vendor Software Product (VSP). A VSP is transformed in a TOSCA blueprint by SDC engine and then it can be used as TOSCA node type during the service composition.

The virtual infrastructure manager used by ONAP to handle the creation of cloud resources is Openstack. The latter exposes a set of APIs that allow creation of most OpenStack resource types and this resources, once created, are referred to as stacks.

3.2.2 Service creation

Service creation consists of service attributes definition and service distribution through GUI of SDC. This GUI allows to design entirely the service accross different user account with different roles (such as designer, tester, governor). Each service is thought to be

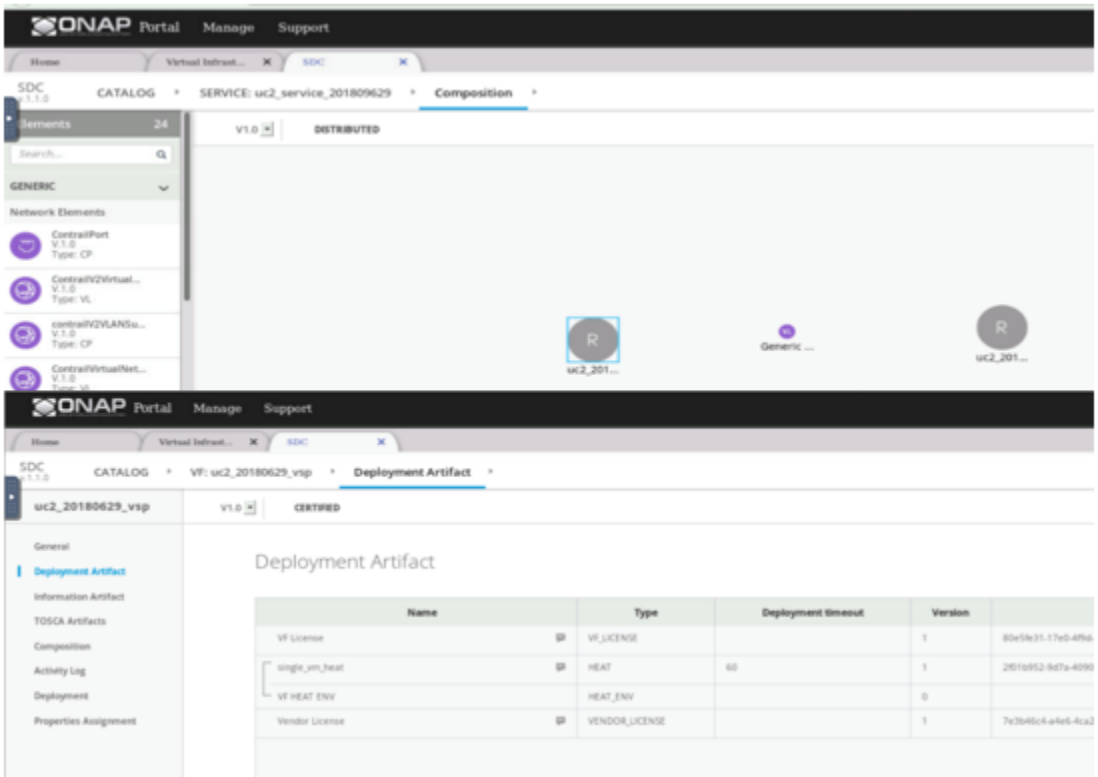


Figure 3.1: SDC Service design

associated with additional informations (such as vendor name, description, and so on) and a Vendor Licence Model (VLM) must to be specified. After license creation, we can onboard all VNFs required by the service as VSPs and import them inside SDC's Catalog as TOSCA node types.

A service is composed from the GUI by adding resources and relationships among VNFs, PNFs and Networks; these resources can be both default and custom TOSCA node types and for each of them is possible to customize Tosca properties and specify the input parameters necessary during execution.

Service distribution

At this point the service definition is complete and its model can be distributed to the Runtime framework. SDC provides an interface for distributing the modeled service, TOSCA artifacts and CSAR file, to SDNC, AAI and MSO using DMaaP notitications. Once a service artifact is downloaded/deployed, a consumer application publishes a status notification; in this way SDC knows the state of the distribution accross different components.

From the Tosca artifacts, MSO stores in its Catalog the models of the resource types, dependencies, parameters and Heat templates that compose the service. This gives MSO the knowledge to handle the orchestration for service's instance creation.

3.3 Execution-time phase

Runtime Framework is the responsible for the service instantiation and provides APIs to command the several components. The core of orchestration is inside MSO that executes BPMN workflows and handles interactions with AAI and SDNC. By default MSO provides a large number of APIs to start the execution of predefined BPMN workflows, which are responsible for distributing the resources belonging to the service one at a time. In addition MSO provides the capabilities to personalize the instantiation of a resource using custom BPMN workflows. One advantage of this approach is that the service must not be instantiate entirely but we can deploy only desired resources.

In Amsterdam there is no help for automation of a service instantiation therefore a service is deployed sending an ordered sequence of APIs from BSS/OSS and each API has the job to trigger a precise phase for service instantiation.
Automation can be provided only for custom services mainly in two ways:

- **Global BPMN:** is a BPMN diagram that contain all the logic to instantiate, entirely or partially, a service

- **API Handler modification:** modify the MSO API Handler by adding new APIs with their own logic for service deployment

During execution, MSO and SDNC perform their own workflows and interact with AAI to store and retrieve information about resource instances and relationships.

3.3.1 MSO workflow

MSO uses two database during the orchestration for service instantiation: Request DB and Catalog DB. Request DB is used to track open and completed requests while Catalog DB is populated via SDC adapter with the informations contained in the TOSCA artifacts. After a service distribution, MSO contains in Catalog DB different types of data:

- **Heat templates:** are used from Network and VNF adapters to instantiate cloud resources in Openstack
- **Resource models:** are the several modules that compose the service; they can be of four types: service, network, VNF, VF modules. The service model represents the root resource of the service and has relationships with other resource models belonging to the service
- **Resource recipes:** represent the mapping between resource models and a BPMN workflow. Each resource model has its own recipe table where a resource model name is associated with a BPMN diagram name; in this way is possible to change the BPMN workflow executed for a resource instantiation

When an API request arrives, API Handler is in charge to manage the request and parse the body to retrieve model information about the resource to be instantiated. Using the model name of the resource, API handler queries recipe table to knows the name of the BPMN process to execute. The Camunda Execution Engine exposes a REST endpoint to which the API Handler send requests for BPMN execution. The message sent by the API Handler to this endpoint is a JSON wrapper, containing all information about incoming request, and the connection is kept open until the main process flow sends back a response.

The BPMN workflow describes the orchestration logic; it includes either calls to nested BPMNs or execution of scripts, written in Java or Groovy code, that interacts with other ONAP components. During flow execution, BPMN performs the following main tasks:

- stores and retrieves informations from AAI related to resource instances

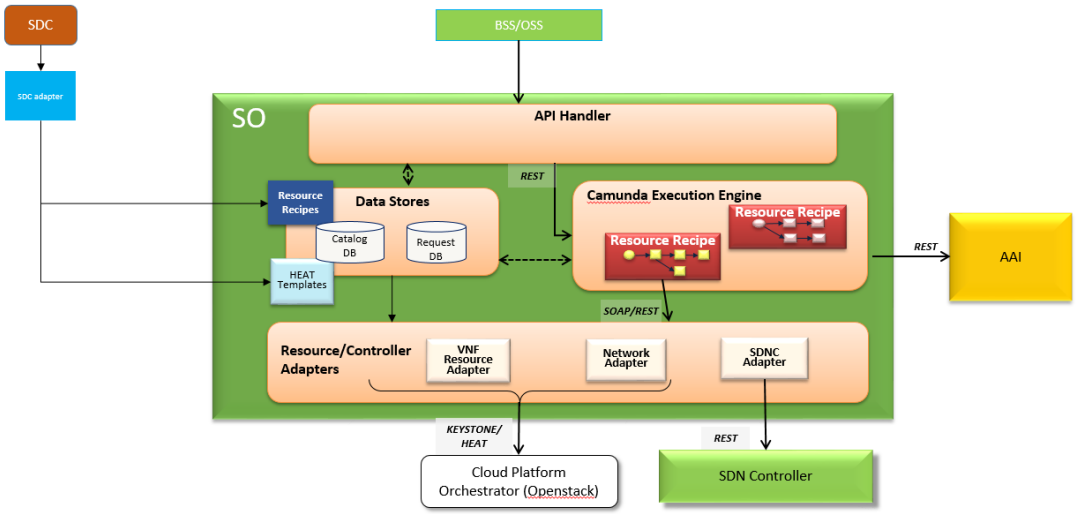


Figure 3.2: MSO Structure

- interacts with SDNC providers through SDNC adapter to request and configure network resources
- interacts with Network Adapter and VNF Adapter to instantiate virtual networks and VMs in Openstack. These adapters use Heat templates stored in Catalog DB to create a stack via Heat REST APIs

3.3.2 APIs dependencies

Any type of service requires additional informations before being deployed, so we have to populate AAI via REST APIs with the following data:

- **Customer:** identy the client who requested the service
- **CloudRegion:** specify id and name of the Openstack Tenant where the cloud resources (VMs, virtual networks) will be instantiated
- **Service subscription:** this is just a label and maps to the service-type we associate with the customer entry and used as part of instantiation phase

Moreover there are some principles to keep in mind for service instantiation:

- *service model* must be instantiated before other resource models belonging to the service. This is because each resource model (network, VNF or VF module) to be implemented requires a relationships with a service instance

- *VNF model* must be instantiated before VF modules of which it is composed. This is because each VF module to be deployed requires a relationships with a VNF instance
- *VF module and virtual network model* require network parameters before being instantiated. This parameters can be provided either with preload API or by adding custom logic inside SDNC

3.3.3 Interactions with SDNC and AAI

The interactions between MSO, SDNC and AAI are not predefined at all but they are described inside BPMN workflows, so each BPMN execution will perform different kind of interactions with SDNC and AAI.

To give a better idea of how these components work together, let's suppose to instantiate a simple service composed only of a virtual network. We can distiguish three distinct steps: service instance creation, preload network parameters, virtual network creation.

Service instance creation

From BSS/OSS we send an API request to MSO with a body containing information data about service model to be instantiated. The API handler takes in charge the request and, based on "service_recipe" table, invokes BPMN workflow associated with service model name. This BPMN has the main task to create an entry in AAI representing a service instance to which other resource instances, belonging to the same service, will be correlated. In this way AAI creates a hierarchical tree of relationships among instances for describing the service.

Preload network parameters

From BSS/OSS we send an API request to SDNC for the Preload Network provider with a body containing network parameters for the virtual network. Then the provider calls the Service Logic Interpreter that checks if a Directed Graph exists for the requested API, gets it from the SDNC Database and executes it. In this case the DG saves network parameters into the MDSAL Datastore.

Virtual network creation

From BSS/OSS we send an API request to MSO with a body containing information data about network model to be instantiated. The API handler invokes BPMN workflow, associated with network model name, that queries the SDNC to know which network

to use for the requested network type. SDNC saves in AAI the network parameters, stored previously in MDSAL, with a unique identifier (UUID) and returns it to MSO. At this point BPMN workflow queries the AAI using the UUID got from SDN-C to get the network parameters and invokes the Network Adapter that retrieves Heat template of virtual network from Catalog DB and merges it with network parameters got from the AAI. Finally Network Adapter uses Openstack APIs to create a stack, sending Heat template with network parameters.

Chapter 4

ONAP Installation

This chapter describes how to setting up the ONAP platform in a cloud environment using two different installation modes based on Openstack.
Furthermore, in the first section, we also give a description of the tools required to prepare the underlying cloud infrastructure needed before launching the ONAP installation.

4.1 Tools required for installation

4.1.1 Openstack

The OpenStack (OS) [11] project is an open source cloud computing platform for all types of clouds, which aims to be simple to implement, massively scalable, and feature rich. OpenStack lets users deploy virtual machines and other instances that handle different tasks for managing a cloud environment. It makes horizontal scaling easy, which means that tasks that benefit from running concurrently can easily serve more or fewer users on the fly by just spinning up more instances.

OpenStack provides an Infrastructure-as-a-Service (IaaS) solution through a set of interrelated services. Each service offers an application programming interface (API) that facilitates this integration. Depending on the needs, it is possible to install some or all services. The following list describes the main core services that make up the OpenStack architecture:

- **Compute (Nova):** manages the lifecycle of compute instances in an OpenStack environment. It is used for deploying and managing large numbers of virtual machines and other instances on demand to handle computing tasks
- **Network (Neutron):** enables Network-Connectivity-as-a-Service for other OpenStack services. It provides an API for users to define networks and the attachments

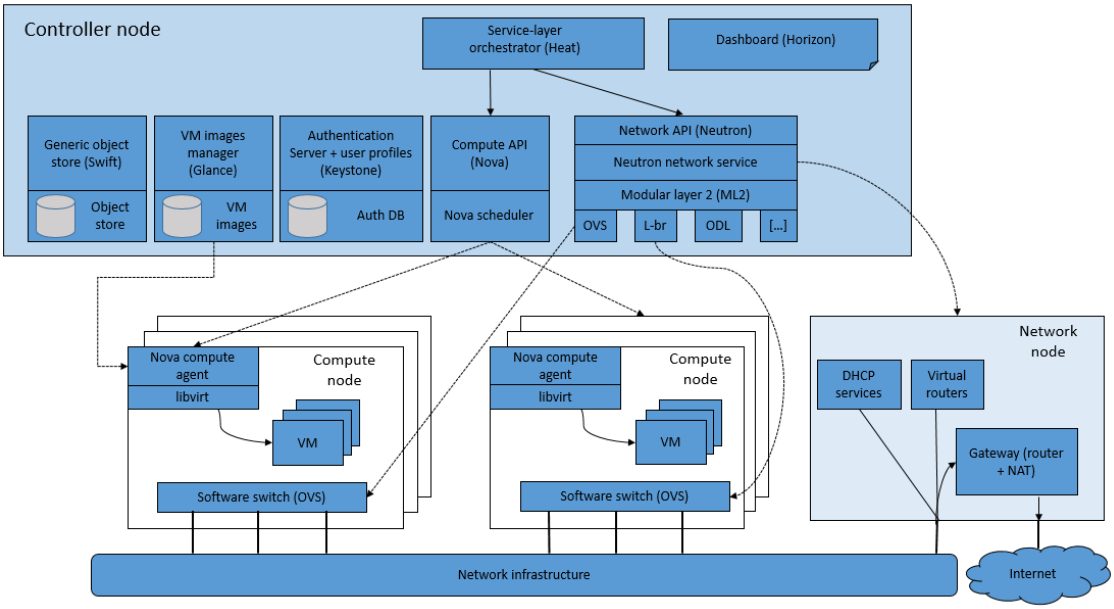


Figure 4.1: Openstack Logical Architecture

into them. It ensures that each of the components of an OpenStack deployment can communicate with one another quickly and efficiently

- **Image storage (Glance):** provides image services to OpenStack. In this case, "images" refers to images (or virtual copies) of hard disks. Glance allows these images to be used as templates when deploying new virtual machine instances
- **Object storage (Swift):** is a storage system for objects and files. Rather than the traditional idea of a referring to files by their location on a disk drive, developers can instead refer to a unique identifier referring to the file or piece of information and let OpenStack decide where to store this information. This makes scaling easy, as developers don't have the worry about the capacity on a single system behind the software
- **Block Storage (Cinder):** provides persistent block storage to running instances
- **Identity (Keystone):** provides an authentication and authorization service for other OpenStack services. It is essentially a central list of all of the users of the OpenStack cloud, mapped against all of the services provided by the cloud, which they have permission to use
- **Orchestration (Heat):** orchestrates multiple composite cloud applications by using either the native HOT template format or the AWS CloudFormation template

format, through both an OpenStack-native REST API and a CloudFormation-compatible Query API. In this way, it allows developers to store the requirements of a cloud application in a file that defines what resources are necessary for that application

- **Dashboard (Horizon):** is a web-based self-service portal to interact with underlying OpenStack services, such as launching an instance, assigning IP addresses and configuring access controls. Developers can access all of the components of OpenStack individually through an API, but the dashboard provides system administrators a look at what is going on in the cloud, and to manage it as needed
- **DNS (Designate):** is a multi-tenant DNS as a Service (DNSaaS) for OpenStack. It provides a standard, open API that can be used to program DNS with integrated Keystone authentication

A standard network architecture design includes a cloud controller host, a network gateway host, and a number of hypervisors for hosting virtual machines. The cloud controller and network gateway can be on the same host.

Networking architecture

The OpenStack Networking service resides on the Controller node and provides an API that allows users to set up and define network connectivity and addressing in the cloud. OpenStack Networking handles the creation and management of a virtual networking infrastructure, including networks, switches, subnets, and routers for devices managed by the OpenStack Compute service. It consists of the neutron-server, a database for persistent storage, and any number of plugin agents. A wide choice of plugins are available. For example, the open vSwitch and linuxbridge plugins utilize native Linux networking mechanisms, while other plugins interface with external devices or SDN controllers. The Modular Layer 2 (ML2) plugin is a framework allowing OpenStack Neutron to simultaneously utilize the many layer 2 networking technologies found in complex real-world datacenters. It cleanly separates management of network types from the mechanisms for accessing those networks (e.g., VLANs, VxLAN, GRE, etc.).

The Network node handles the majority of the networking workload. It hosts the DHCP agent, the Layer-3 (L3) agent and the Layer-2 (L2) agent. In addition to plugins that require an agent, it runs an instance of the plugin agent to perform local networking configuration. Both the Open vSwitch and Linux Bridge mechanism drivers include an agent.

The Compute node hosts the compute instances themselves. To connect compute instances to the networking services, Compute nodes must run the L2 agent. Like the

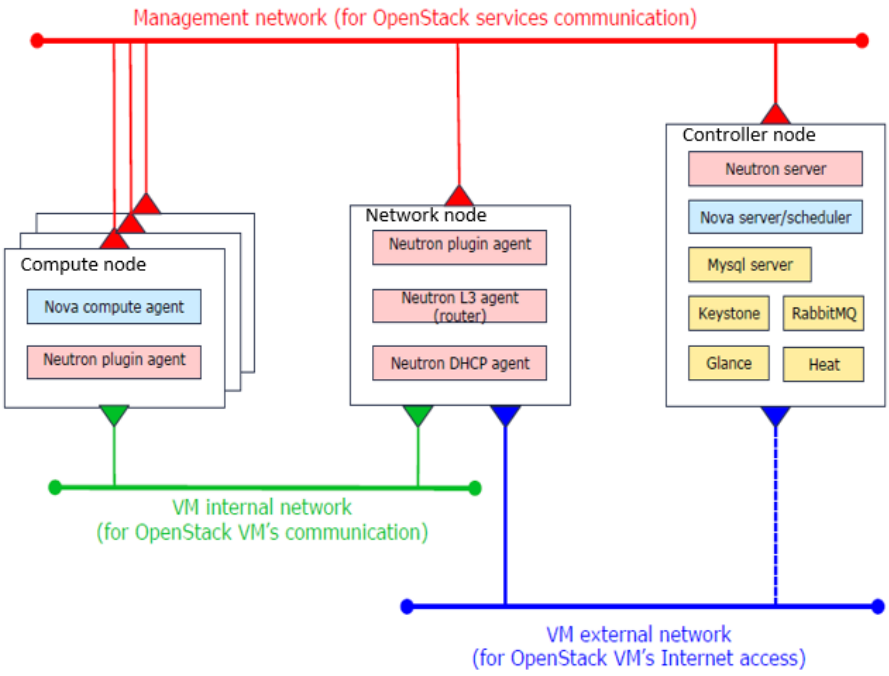


Figure 4.2: Openstack Networks

Network node that handle data packets it must also run an instance of the plugin agent for local networking configuration.

Openstack allows users to create tenant networks for connectivity within projects; they are fully isolated by default and are not shared with other projects. Networking supports a range of tenant network types:

- *Flat:* All instances reside on the same network, which can also be shared with the hosts. No VLAN tagging or other network segregation takes place
- *Local:* Instances reside on the local compute host and are effectively isolated from any external networks.
- *VLAN:* OpenStack Networking enables to create multiple provider or tenant networks using VLAN IDs (802.1Q tagged) that correspond to VLANs present in the physical network. This allows instances to communicate with each other across the environment.
- *VXLAN and GRE:* they use network overlays to support private communication between instances. Tunneling encapsulates network traffic between physical Networking hosts and allows VLANs to span multiple physical hosts. Instances communicate

as if they share the same layer 2 network. Open vSwitch supports tunneling with the VXLAN and GRE encapsulation protocols. A Networking router is required to enable traffic to traverse outside of the GRE or VXLAN tenant network.

4.1.2 Kubernetes

Kubernetes [12] (commonly stylized as K8s) is an open source system for orchestration and container management. It was developed by the Google team and is now maintained by the Cloud Native Computing Foundation (CNCF). Kubernetes facilitates the deployment and scalability of containerized applications and easily and efficiently manages clustered hosts on which containers are executed. It works with a range of container tools, including Docker.

Kubernetes follows a master-slave architecture; it consists of a Master controller and a set nodes. A node is a worker machine, previously known as a minion. A node may be a VM or physical machine, depending on the cluster. Each node contains the services necessary to run pods and is managed by the master components. The services on a node include the container runtime, kubelet and kube-proxy.

The minion nodes pool together their resources to form a more powerful machine. When you deploy programs onto the cluster, it intelligently handles distributing work to the individual nodes for you. If any nodes are added or removed, the cluster will shift around work as necessary.

Kubernetes is composed of several parts that are designed to be loosely coupled and extensible to meet different workloads. The main components are the following:

- **Pod:** the pods are the atomic unit on the Kubernetes platform. Each pod wraps one or more containers into a higher-level structure and is tied to the node where it is scheduled. Any containers in a pod share the same resources and the same local network. Containers run in a shared context on the same node while maintaining a level of isolation from the others

- **Kubelet:** is responsible for the running state of each node, ensuring that all containers on the node are healthy, and it takes care of starting, stopping, and maintaining application containers

- **API Server:** is the main management point of the entire cluster. It serves up the Kubernetes API and is the front-end for the control plane

- **Controller Manager:** is a daemon that incorporates the control core loops supplied with Kubernetes. In practice, a controller checks the status of the cluster

using the API Server monitoring function and, when notified, makes the necessary changes to move the current state to the desired state

- **Scheduler:** looks for unscheduled pods and binds them to nodes, based on resource availability, quality of service requirements, and other constraints
- **Etcd:** is a distributed, consistent, and highly available key-value repository that stores the configuration data of the cluster. It is used for configuration management, service discovery, and coordinating distributed work

Moreover Kubernetes provides a partitioning of the resources that it manages into non-overlapping sets called namespaces. Namespaces are a way to divide cluster resources between multiple users.

Services

Applications running in a Kubernetes cluster find and communicate with each other, and the outside world, through the Services. A Kubernetes Service is an abstraction which defines a logical set of Pods and a policy by which to access them - sometimes called a micro-service. Services can be exposed in different ways by specifying a type that determine accessibility from inside and outside of cluster. There are several types of Services:

- *ClusterIP:* exposes a service on an internal IP in the cluster, which makes the service only reachable from any container (even from different pods) within the same cluster
- *NodePort:* is a ClusterIP service with an additional capability, it is reachable on each Node's IP at a static port (the NodePort). A ClusterIP service, to which the NodePort service will route, is automatically created. In thi way it is possible to contact the NodePort service from outside the cluster
- *LoadBalancer:* combines the capabilities of a NodePort with the ability to setup a complete ingress path. It exposes the service externally using a cloud provider's load balancer. NodePort and ClusterIP services, to which the external load balancer will route, are automatically created.

4.1.3 Rancher and Helm

Rancher [13] is a container management platform for controlling multiple Kubernetes clusters running anywhere, on any provider. It also allows to manage cluster nodes, adding, removing, deploying applications through a unique web interface. Rancher uses

Docker as the underlying container runtime and coordinate running containers between multiple physical/virtual nodes. Rancher also includes modular infrastructure services including networking and load balancing.

Rancher adds significant value on top of Kubernetes, primarily by centralizing role-based access control for all clusters and giving global admins the ability to control cluster access from one location. It then enables detailed monitoring and alerting for clusters and their resources and integrates directly with Helm.

Helm [14] is an application package manager running atop Kubernetes. It allows describing the application structure through convenient helm-charts and managing it with simple commands. Helm Charts enable to define, install, and upgrade even complex Kubernetes application and simplifies the management of microservices. The main benefit of this approach is the ability to consider scalability from the start. For example an application composed of clearly defined microservices can scale only the ones we need to scale, adding more Kubernetes nodes and pods to the cluster.

4.2 Testbed environment

ONAP can be deployed in different ways, depending on the requirements of the service provider. In this chapter we describe two types of installations we have tested: ONAP on Openstack and Onap on Kubernetes on Openstack. Both of them require an existing Openstack installation with the following base components deployed in the infrastructure: Cinder, Glance, Heat, Horizon, Keystone, Neutron, Nova, Designate. For these ONAP installations we used the OpenStack Ocata release but it is possible to use several Cloud providers offering OpenStack based solutions.

The figure below shows the structure and the available resources of Openstack Ocata used for the ONAP installation.

The Openstack installation is composed of one server (OSC) that acts as Cloud controller and six server (OSA) that act as Compute nodes. In this case the Controller node and the Network node are on the same host. The Cloud controller exposes all OpenStack APIs, including the network API, to tenants and anyone on the Internet via the API network. Compute nodes and Cloud controller are connected together via Management Network for Openstack services and VM's communication.

Moroever the virtual router, within Cloud controller, connects tenant networks with External Network and so to the Internet. It also provides the ability to connect to instances directly from an external network using floating IP addresses.

Resources	Configured
vCPU	200
RAM	572 GB
Ephemeral HD	16 TB
Persistent HD	1,4 TB
Floating IPs	240

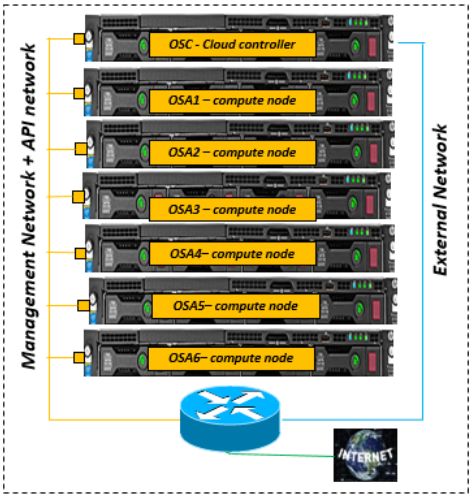


Figure 4.3: Openstack testbed

4.3 ONAP on Openstack

This ONAP installation is made directly on Openstack using a predefined Heat template that spins up all the ONAP components. The Heat template refers to an environment file in which all the default parameter values are defined and it can be deployed via Horizon dashboard or Command Line (using the OpenStack Heat service). By modifying the template resources, we can customize the ONAP installation so that only the desired components are distributed.

When the Heat template is executed, the OpenStack Heat engine creates a new stack with the resources defined in the template, based on the parameters values defined in the environment file. Each ONAP component is represented by one or more VM.

4.3.1 Requirements

ONAP on Openstack installation requires the following resources:

- 29 VM
- 148 vCPU
- 336 GB RAM
- 3 TB Ephemeral HD
- 29 floating IP addresses

Most of the resources are required by the DCAE deployment, in fact it takes 15 VMs. In addition the following artifacts must be deployed on the OpenStack infrastructure:

- a public SSH key to access the several VMs
- private SSH key and public SSH key to be used between DCAE VMs
- three images: Ubuntu 14.04, Ubuntu 16.04 and CentOS 7
- a set of flavors: small, medium, large, medium, large, xlarge, xxlarge

The default installation assumes that the Default security group of Openstack is configured to enable full access between VMs representing the ONAP components.

The OpenStack infrastructure must enable Internet access and it is necessary to have an External network already configured properly. The External network ID will have to be provided in the Heat environment file.

4.3.2 Heat template and parameters

The Heat template is composed of two sections: parameters and resources. The parameter section contains the declaration and description of the parameters that will be used to setting up ONAP. The resource section contains the definition of:

- Operation And Management (OAM) private network, which ONAP components use to communicate with each other and with VNFs
- ONAP Virtual Machines (VMs)
- virtual interfaces towards the OAM network
- disk volumes

ONAP VMs have a private IP address in the OAM private network space and use floating IP addresses for remote access and connection to repositories. A router that connects the ONAP Private Management Network to the External network is also created. Furthermore each VM runs a post-instantiation script that downloads and installs software dependencies (e.g. Java JDK, gcc, make, Python, ...), ONAP software packages and Docker containers from remote repositories.

ONAP installs a DNS server used to resolve IP addresses in the ONAP OAM private network. ONAP Amsterdam release also requires OpenStack Designate DNS support for the DCAE platform, so as to allow IP address discovery and communication among DCAE elements. This is required because the ONAP Heat template only installs the

DCAE bootstrap container, which will in turn install the entire DCAE platform. As such, at installation time, the IP addresses of the DCAE components are unknown.

In the environment file it is necessary to customize some parameters that need to be set depending on the user's environment:

- *Openstack parameters:* ID of the External network, ID and credentials about tenant on which VMs will be deployed, URL endpoints (like Keystone and Horizon), images and flavors, public SSH key to access VMs
- *network parameters:* CIDR of OAM network, DNS IPs, private IP addresses of VMs
- *DCAE parameters:* informations about DCAE VMs (like SSH key pair and additional VM image IDs/names), configuration parameters relate to DNSaaS support provided by Designate

4.4 ONAP on Kubernetes on Openstack

This ONAP installation is based on a Kubernetes cluster created on an OpenStack environment. ONAP is deployed using the ONAP Operations Manager (OOM) which provides the ability to manage the entire life-cycle of an ONAP installation, from the initial deployment to final decommissioning. OOM can be deployed on a private set of physical hosts or VMs (or even a combination of the two) and it uses the Kubernetes/Helm system as a complete ONAP management system to drive all user driven life-cycle management operations:

- **Deploy:** a comprehensive set of Helm charts describe the composition of each of the ONAP components and the relationships within and between components. Using this model Helm is able to deploy all or partially the ONAP platform
- **Configure:** each project within ONAP has its own configuration data generally consisting of environment variables, configuration files, and database initial values. It is possible to modify Helm charts to customize the ONAP installation
- **Monitor:** ONAP includes mechanisms to monitor the real-time health of its components
- **Scale:** many of the ONAP components are horizontally scalable which allows them to adapt to expected offered load

- **Heal:** the Helm charts implement automatic recoverability of ONAP components when individual components fail. This mechanism ensures that, after a failure, all of the ONAP components restart successfully
- **Upgrade:** Helm has built-in capabilities to enable the upgrade of pods without causing a loss of the service being provided by that pod or pods
- **Delete:** existing deployments can be partially or fully removed once they are no longer needed

4.4.1 Requirements

Onap is installed on an underlying cloud infrastructure composed of: 1 VM running Rancher and one or more VM representing Kubernetes nodes. The following minimal resources are required for a full ONAP deployment (all components including DCAE):

	Number VM	vCPUs	RAM (GB)	Disk (GB)	Floating IPs
Rancher	1	2	4	40	-
Kubernetes	1	8	80-128	100	-
DCAE	15	44	88	880	15
Total	17	54	156-220	1020	15

As in the previous installation the DCAE takes most of the resources; customizing ONAP to deploy only components that are needed drastically reduce the requirements. For our installation we haven't installed the DCAE and we have changed the number and the flavor of VMs allocated for Kubernetes nodes to host ONAP components. We have created 5 VMs, each of one with floating IP: 1 VM for Rancher and 4 VMs for Kubernetes nodes. ONAP VMs have a private IP address to communicate with each other and use floating IP addresses for remote access and connection to repositories. Also in this case an External network and a router enable Internet access from all VMs and the Default security group of Openstack is configured to allow full access between VMs.
Moreover on the OpenStack infrastructure we have deployed the following artifacts:

- a SSH keypair to access the several VMs

- one image, Ubuntu 16.04, used for all VMs
- two flavors: one for Rancher VM (4 vCPUs, 4 GB RAM, 80 GB Disk) and one for Kubernetes VMs (6 vCPUs, 30 GB RAM, 80 GB Disk)

4.4.2 Cloud infrastructure

This section describes the steps for the installation of Kubernetes on an OpenStack environment with Rancher. Firstly we have to create a VM that acts as Master node in which runs Rancher, Helm, Docker and a NFS (Network File System) server; this node will not be used to host ONAP itself, it will be used exclusively by Rancher. Secondly we create the VMs for Kubernetes, each of which runs Docker, Helm agents and a NFS common.

At this point we can access the Rancher UI from Master node to create the Kubernetes environment and then add VMs representing the Kubernetes nodes. After a Kubernetes environment has been created, the infrastructure services will not be started until at least one host is added. The process of adding hosts is the same steps for all container orchestration types. Once the first host has been added, Rancher will automatically start the deployment of the infrastructure services including the Kubernetes services (i.e. master, kubelet, etcd, proxy, etc.). The Master Node runs Rancher and Helm clients and connects to all the Kubernetes nodes in the cluster. Kubernetes nodes, in turn, run Kubernetes and Helm agents, which receive, execute, and respond to commands issued by the Master Node (e.g. kubectl [1] or helm operations).

Furthermore deploying applications to a cluster requires Kubernetes nodes to share a common, distributed filesystem. In this case the Master node plays the role of NFS Master while all the other cluster nodes play the role of NFS slaves.

4.4.3 OOM deployment

OOM deploys and manages ONAP on a pre-established Kubernetes cluster but the lifecycle of this cluster is independent of the life-cycle of the ONAP components themselves. In fact we can clone the OOM repository, from ONAP Gerrit for the desidered release (Amsterdam in our case), in the Master node and then start the OOM deployment. Much like an OpenStack environment, the Kubernetes environment may be used for an extended period of time, possibly spanning multiple ONAP releases.

The Helm model of ONAP used by OOM is composed of a set of hierarchical Helm charts that define the structure of the ONAP components, the configuration of these

[1] kubectl: Kubernetes command-line tool for deploying and managing applications on Kubernetes.

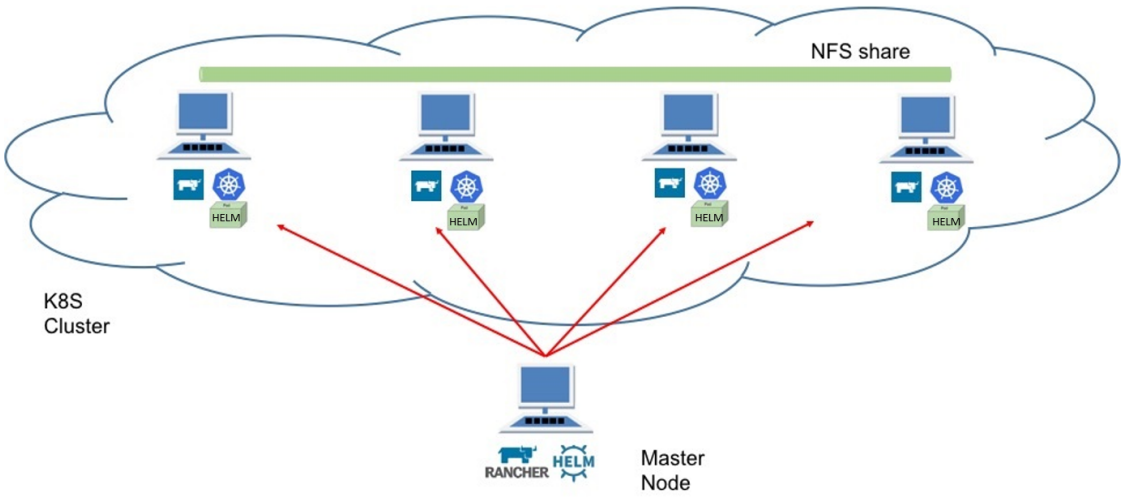


Figure 4.4: K8s cluster with Rancher

components and the related set of Kubernetes resources. These Helm charts describe the desired state of an ONAP deployment and instruct the Kubernetes container manager as to how to maintain the deployment in this state. Furthermore these dependencies dictate the order in which the containers are started for the first time so that such dependencies are always met without arbitrary sleep times between container startups. When an initial deployment of ONAP is requested the current state of the system is empty so ONAP is deployed by the Kubernetes manager as a set of Docker containers on one or more predetermined hosts. When deploying on virtual machines the resulting system will be very similar to “Heat” based deployments, i.e. Docker containers running within a set of VMs, the primary difference being that the allocation of containers to VMs is done dynamically with OOM and statically with “Heat”.

Each ONAP component consists of a group of containers with shared storage and networking that are grouped together into a set of Kubernetes pods. In Amsterdam release, pods are mapped one-to-one to docker containers and a namespace is created for each of the ONAP components. The Kubernetes namespace concept allows for multiple instances of a component (such as all of ONAP) to coexist with other components in the same Kubernetes cluster by isolating them entirely. In addition these namespaces expose services that provide external connectivity to pods; OOM uses the Kubernetes service abstraction to provide a consistent access point for each component independent of the pod or container architecture of that component. For example, the SDNC component may introduce OpenDaylight clustering and change the number of pods in this component but this change will be isolated from the other ONAP components by the service abstraction.

A service can include a load balancer on its ingress to distribute traffic between the pods and even react to dynamic changes in the number of pods.

During the OOM deployment it is possible to follow the progress of the ONAP installation and the pods creation using kubectl or the Rancher GUI.

4.5 Evaluation

In this chapter we have discussed two different ways of installing the ONAP platform. Both configure ONAP and allow access to the services exposed by the several components but they have advantages and disadvantages in terms of resource usage, flexibility and complexity.

Comparing the two installations, we have decided to use ONAP on Kubernetes on Openstack as the environment on which develop the service described in Chapter 6. Despite the greater complexity of installation and infrastructure management this choice offers important benefits:

- **limited resource usage:** as opposed to VMs that require a guest operating system be deployed along with the application, containers provide similar application encapsulation with neither the computing, memory and storage overhead
- **lifecycle management:** Kubernetes and Rancher compose a comprehensive system for managing the lifecycle of containerized applications. Their use as a platform manager ease the deployment of ONAP, provide fault tolerance and horizontal scalability, and enable seamless upgrades
- **rapid deployment:** eliminating the guest operating system results in containers coming into service much faster than a VM equivalent

Chapter 5

EVPL service implementation in ECORD

In this chapter we discuss in detail the E-CORD use-case developed in TIM labs describing its architecture, implementation and execution phase. We focus on this service because it will be integrated and used within ONAP to build the EVPL service presented in chapter 6.

5.1 Overview

Enterprise CORD (E-CORD) is a use-case that offers enterprise connectivity services over metro and wide area networks and it is based on the architecture of another project, CORD, which aims to bring datacenter economics and cloud agility to service providers. Through E-CORD we can create on-demand EVPL[1] services by using a hierarchical approach of SDN controllers that manage the several portions of the network and interact with physical devices.

In the following sections we first analyze the CORD project, the features it proposes and its architecture, and then move on to the description of the E-CORD use-case and its actual implementation within TIM labs.

[1]Ethernet virtual private line (EVPL) is a data service, defined by the MEF, which connects two Ethernet ports on a WAN and provides a point-to-point or point-to-multipoint connection between a pair of UNI.

5.2 CORD

CORD (Central Office Re-architected as a Datacenter) [15] is an architecture for the Telco Central Office that combines SDN, NFV, and elastic cloud services - all running on commodity hardware - to build cost-effective, agile networks with significantly lower CAPEX/OPEX and to enable rapid service creation and monetization. The goal of CORD is not only to replace today's purpose-built hardware devices with their more agile software-based counterparts, but also to make the Central Office an integral part of every Telco's larger cloud strategy, enabling them to offer more valuable services.

A reference implementation of CORD consists of a collection of commodity servers, interconnected by a fabric constructed from white-box switches, and disaggregated access technologies with open source software to provide an extensible service delivery platform. This gives network operators the means to configure, control, and extend CORD to meet their operational and business objectives. As shown in Fig. 5.1, the switching fabric is organized in a leaf-spine topology to optimize for traffic flowing east-to-west - between the access network that connects customers to the Central Office and the upstream links that connect the Central Office to the operator's backbone. The NFV and SDN control plane is composed by ONOS (described in next section), Openstack and XoS [16] as orchestrator. All controller entities run on Docker containers along with the deployed VNFs. On top of the software infrastructure different use-case domains leveraged on CORD: Residential, Mobile and Enterprise.

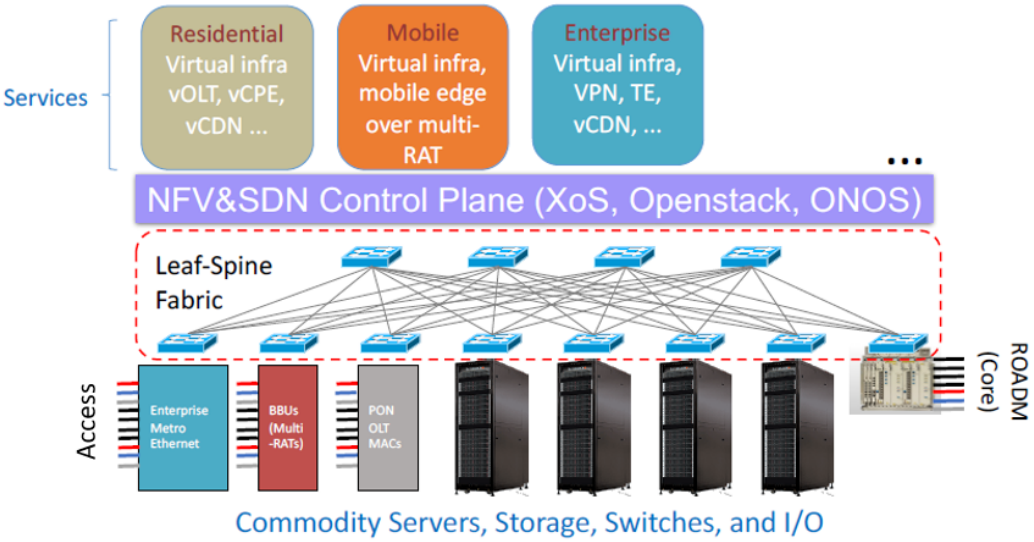


Figure 5.1: CORD infrastructure

CORD offers an open-source reference platform for unified network resource orchestration from a centralised vantage point and fine-grained bandwidth and connectivity service on demand. As End-to-end connectivity involves the control of multiple heterogeneous underlying networks, the CORD infrastructure separates the controllers into a domain-agnostic one, the global orchestrator, and multiple domain-specific controllers. This way it is easier to maintain the platform up and running during temporary down times due to failures or software releases.

5.2.1 ONOS

ONOS (Open Network Operating System) [17] is an SDN operating system for network operators that is designed for scalability, high performance and high availability and to make it easy to create apps and services. It can run as a distributed system across multiple servers, running multiple ONOS instances that are identical in terms of their software stack. The ONOS kernel and core services, as well as ONOS applications, are written in Java as bundles that are managed by the Apache Karaf OSGi container [18]. OSGi is a component system for Java that allows modules to be installed and run dynamically in a single JVM. Since ONOS runs in the JVM, it can run on several underlying OS platforms. ONOS allows to build carrier-grade solutions that leverage the economics of white box merchant silicon hardware while offering the flexibility to create and deploy new dynamic network services without the need to alter the dataplane systems. It provides the control plane by managing the entire network rather than a single device and eliminating the need to run routing and switching control protocols inside the network fabric. However, for each device a single controller instance acts as a master, while the others are ready to step in if a failure occurs. With these mechanisms in place, ONOS achieves scalability and resiliency.

ONOS maintains a global network view to manage and share network state across ONOS servers in a cluster. This abstraction provides a graph model of the network which corresponds to the underlying network structure. Network topology and state discovered by each ONOS instance such as switch, port, link, and host information is used to construct the global network view. Applications then read from the global network view to make forwarding and policy decisions, which are in turn written to the network view. As applications update and annotate the view, these changes are sent to the ONOS southbound modules and programmed on the appropriate physical device. The southbound modules manage the physical topology, react to network events and program/configure the devices leveraging on different protocols. The ONOS platform is not directly tied to a closed set of protocols, but it provides its own set of high-level abstractions and models, which it exposes to the application programmers, that enable model generation and, by

extension, code/API generation from models. This generalisation can overlay any specific modelling language although YANG has emerged as the data modelling language for the networking domain.

5.3 Architecture

The CORD architecture is composed of a two layer hierarchy of controllers, the root global controller and the leafs local controllers. Each controller is represented by an ONOS instance able to store and distribute the state between the instances of a cluster. The global controller creates, updates and maintains an abstract global view of the network. It interacts with the underlying ONOS controllers to gain knowledge of the network topology and manage the virtual devices of which it is aware of. On the other hand, leaf controllers handle the real network infrastructure by interacting directly with physical devices and each of them maintains its own abstract topology view of the network portion it manages.

The global node has three main logical components:

- **Service orchestrator**: application that exports northbound APIs to outside of the platform and splits service requests into instructions targeting the local controllers
- **Virtual Provider**: receives notifications from the underlying domains about devices, ports and inter-connection links between devices
- **HTTP-Channel**: communication channel to talk with the underlying domain controllers

While in each local controller we have:

- **BigSwitch Service**: topology aggregation mechanism to aggregate topology elements into virtual topology data structures
- **HTTP-Channel**: communication channel to talk with the global controller to notify topology elements and receives network provisioning requests
- **Network Application**: domain-specific application that implements the network provisioning (forwarding, filtering rules, policing, etc.). It interacts with the whole local topology to fully exploit physical device capabilities

The communication channel for bidirectional data exchange between controllers is implemented as a client/server REST channel. In the global controller we have a server to sense topology events from the underlying domains, and a client to propagate service

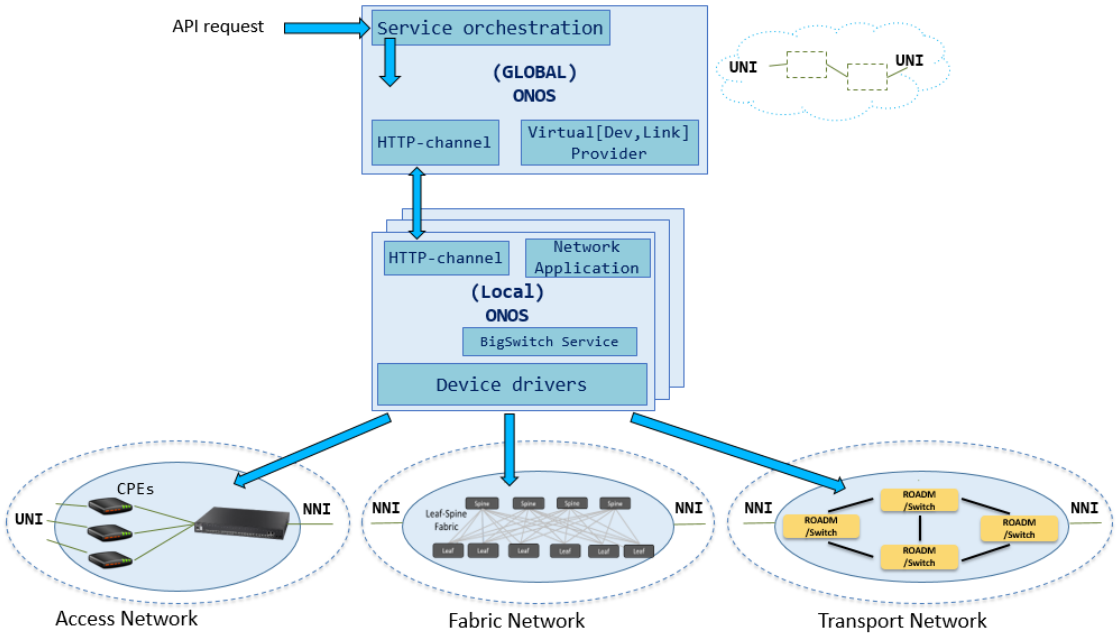


Figure 5.2: ECORD architecture

requests to remote domains. While in local controllers we have a server to receive service requests and a client to send the topology events published by the BigSwitch service. The HTTP client component implements as many Java APIs as the number of specific services offered by the platform and registers itself as a listener object. When a service request arrives at the global node, the request is generally divided into several instructions targeting the virtual domain devices and, for each device, the orchestration service calls all the registered listeners among which only the one that implements the communication with the domain the device belongs to process instructions for that device. The endpoints of the remote domain, the IP, the port and the credentials are provided by the configuration.

Within each local controller the Network Application component translates incoming service requests from the global node into some actions on the network. The implementation of this component encapsulates specific logic and code to define and manage the several device drivers, which can use YANG/NETCONF, SNMP, REST, Openflow and any other protocol to interact with physical devices. Fig. 5.2 illustrates an example of three network domain under the control of local ONOS controllers: an access network with Customer Premise Equipments (CPEs) connected to an Ethernet Edge device; a leaf-spine fabric composed of white-box switches; a transport optical network composed of ROADM switches.

5.3.1 Topology abstraction

The global node maintains an abstract view of the underlying topology to improve scalability and to separate domain-specific and domain-agnostic concerns. For each local controller, a BigSwitchService component exposes one abstract device to the global node: it represents an aggregation of the real network elements that compose the topology of a local site. In this way, the global ONOS has fewer devices and link data structures to deal with. Path computation will involve only these aggregated items, while the actual network provisioning will be achieved by the local site controllers. The relevant topology information for the global node are the connect points representing the demarcation line between a Service Provider and its customers network, the connect points between two Service Provider networks, and relative ports characterization relevant to the services deployed at the global level. The local controller's BigSwitchService aggregates the physical devices into a single device data structure with related relevant connect points to expose to the global controller via an HTTP channel. The connect points are marked as UNI (User-to-Network Interface) for those facing the customer side and NNI (Network-to-Network Interfaces) for those neighboring with an external network, following the terminology adopted by the MEF consortium [19]. The BigSwitchService is responsible to apply the one-to-one mapping between physical and virtual connect points and to notify the global about those changes in the local topology that would affect the aggregated virtual topology; it listens for events of the local topology and propagates events related to the virtual topology.

5.4 Enterprise CORD

E-CORD [20] builds on the same CORD infrastructure to support enterprise customers, alongside residential and mobile customers, and provides enterprise connectivity services (L2 and L3VPN). In addition, service providers can offer services that go far beyond simple connectivity services, as they can include Virtual Network Functions and service composition to support cloud-based enterprise services. In turn, enterprise customers can use E-CORD to rapidly create on-demand networks between any number of endpoints or company branches. These networks are dynamically configurable, implying connection attributes and SLAs can be specified and provisioned on the fly. Furthermore, enterprise customers may choose to run network functions such as firewalls, WAN accelerators, traffic analytic tools, virtual routers, etc. as on-demand services that are provisioned and maintained inside the service provider network.

The E-CORD architecture is composed as follows:

- **multiple Central Office/Local POD**: identified also as E-CORD sites, they are standard CORD sites equipped with specific access equipment, such as an Ethernet edge switch, and are usually located in the Service Providers' Central Offices. A CORD site is used to connect the enterprise user to the service provider network and run value added user services at the edge of the network; it comprises one or more compute nodes, and one or more fabric switches. Upstream, the POD connects to the service provider metro/transport network

- **a Transport Network**: provides connectivity between the several CORD sites. It can be almost anything, from an optical network to a single packet switch, and can be composed of white-boxes, legacy equipment, or a mix of both

- **a Global node**: it is a single machine running in the Service Provider's network, used as general orchestrator that coordinates between all the local PODs of the E-CORD deployment. It is composed by an instance of XoS, for NFV orchestration, and one ONOS instance, for SDN control plane

5.4.1 Testbed

Based on E-CORD architecture, we have chosen to take a hierarchical approach where there is a WAN orchestration layer that acts as SDN control plane for connecting multiple CORD sites together via Carrier Ethernet circuits established on-demand.

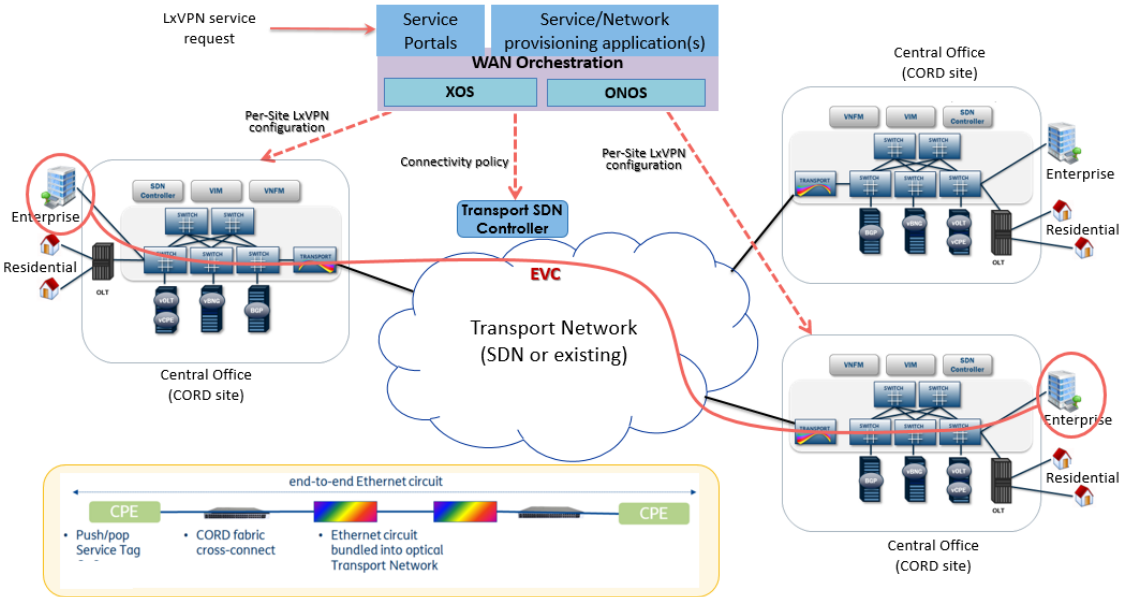
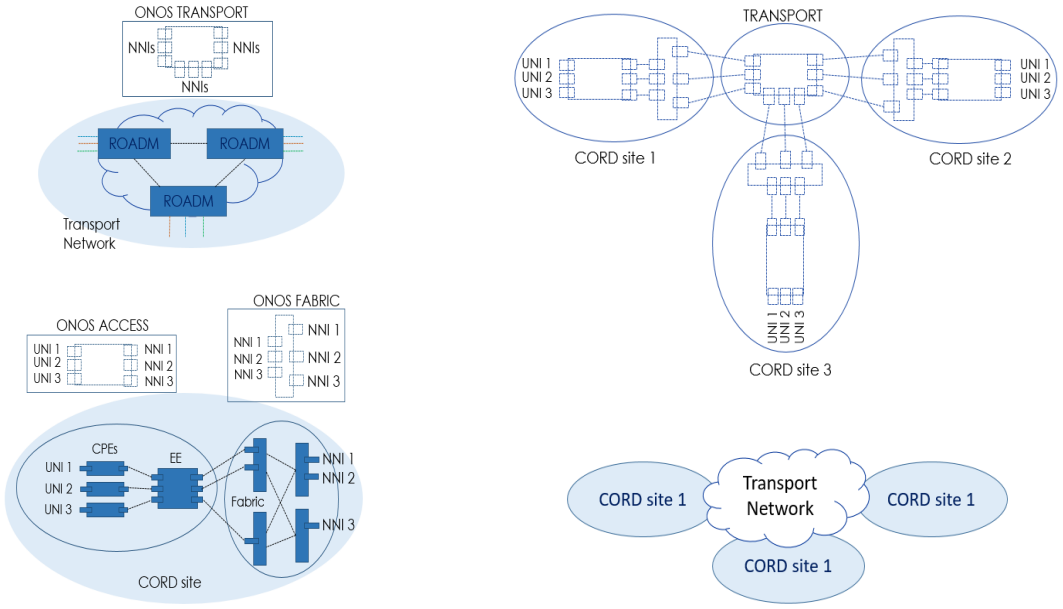


Figure 5.3: E-CORD Point-to-point Carrier Ethernet Service

The Service Orchestrator is the Carrier Ethernet application that exports APIs to setup, tear down and update Ethernet Virtual Circuits (EVCs) spanning multiple sites. An EVC is identified by a service tag (outer vlan tag of the 802.1ad protocol), one or more customer vlan tags (802.1q) mapped to the service tag, a bandwidth profile and a set of UNI ports among which we want to create the layer-2 VPN based on Ethernet. The EVC request is split by the Service Orchestrator into as many forwarding constructs as the number of virtual devices along the path between the UNIs. The forwarding constructs are sent to the local controllers which are responsible to allocate the appropriate network resources.

Each CORD site uses two ONOS controllers to manage the physical network: (1) *ONOS Access* runs the application that controls the edge network, including the CPE devices and the Ethernet Edge (EE) devices, and (2) *ONOS Fabric* runs the application configuring the cross connections within the fabric of CORD to bridge the CPEs to the transport network and eventually to the remote sites; alternatively, it bridges customer's traffic to a chain of VNFs before being routed to the Internet gateway. While the transport network is managed by a single local ONOS instance (*ONOS Transport*) that runs the application for configuring an optical network to bridge several CORD sites.



(a) ONOS-Bigswitch Abstraction (b) ONOS-Global Abstracted view

Figure 5.4: E-CORD topology abstraction

The global ONOS controller builds the global network view from the abstract devices

received by underlying controllers: one abstract device, representing the transport network, is exposed by *ONOS Transport* and two abstract devices, representing the CPEs and the fabric, are exposed by *ONOS Access* and *ONOS Fabric* for each CORD site. Fig. 5.4a shows the device abstraction performed by the Bigswitch Service of the local controllers for CORD sites and transport network, while Fig. 5.4b shows an example of the abstracted topology seen by global node of a network composed of three CORD sites interconnected by the transport network.

In our implementation of E-CORD the CPEs are directly connected to the switching fabric without the EE devices, while the transport network is composed by ROADM switches, one for each CORD site, connected to the leaf switch of the fabric to simulate the optical transport network. The southbound protocols used to control the devices are Netconf for the CPE, Openflow 1.3 for the fabric switches and OpenFlow 1.3 + Optical Transport Protocol Extensions (ONF TS-022) for the ROADMs. The CPE in use is a custom SFP of Microsemi, the ea1000 featured with an embedded Linux operating system and a FPGA board programmable via Yang/Netconf. The fabric whitebox switches are EdgeCore 5712 and the ROADM are custom disaggregated appliances provided by TIM.

5.4.2 Environment details

In order to provision end-to-end connectivity, users can request for a Point-to-point Carrier Ethernet Service through the Service Portal running on the Service Orchestrator. In response to that request, WAN orchestration layer will determine which sites needs to be involved, which path to go through the transport network and what kind of services and policies needs to be applied (such as bandwidth profiles). Then, high-level description of what actions each CORD sites and transport network must take will be sent down to each ONOS controller.

In the following sections we describe (1) the primary initialization phase to configure the E-CORD infrastructure and (2) the instructions sent by global ONOS to manage an incoming user request for EVC creation.

Infrastructure configuration

The ONOS controllers must be initialized so that they are aware of the network infrastructure and can interact together. To do this, through the ONOS API, we perform the following operations:

- **CPE device creation**: each *ONOS Access* receives information on the CPEs it

has to manage. For each CPE it is specified the protocol to use, the IP and the port to which connect, and hardware information

- **ONOS Global node initialization**: it receives endpoint information about underlying domains (CORD sites and transport network). This allows to use the HTTP-Channel for communication with the several ONOS controllers
- **Local controllers initialization**: each ONOS controllers receive (1) information on the global ONOS endpoint for HTTP-Channel communications, (2) domain-specific information regarding network ports, devices and applications, and (3) information on MEF connect points marked as UNI, ENNI (Egress NNI), INNI (Ingress NNI) such that the BigSwitch Service is able to abstract a single device to be sent to the global ONOS

EVC creation

An API request for EVC creation contains a pair of UNI endpoints, the bandwidth profile to be applied and a list of customer vlan tags (c-tags) enabled to use such EVC. Then the global node generate a service vlan tag (s-tag) and sends a set of APIs, via HTTP-Channel, to all the ONOS controllers involved in the path among UNI endpoints received. Fig. 5.5 illustrates the sequence of APIs sent to create an EVC between two CORD sites, both represented by an *ONOS Access* and an *ONOS Fabric*, connected by the transport network.

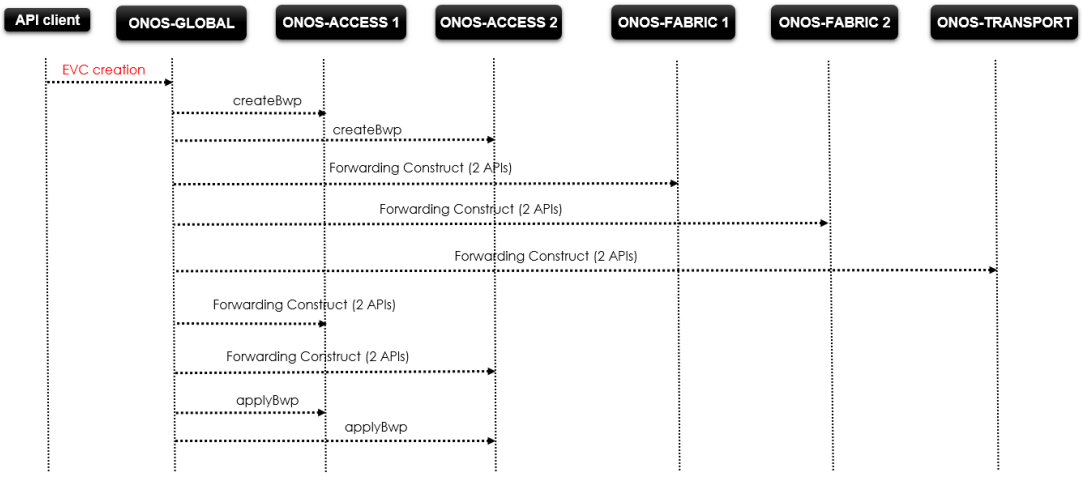


Figure 5.5: EVC creation

The forwarding constructs (FCs) instruct ONOS controllers to install flow rules into the physical devices. Each FC specifies: the s-tag, to identify the EVC, and a pair of MEF

ports, related to the abstract device of the ONOS controller to which the FC is directed. These MEF ports represent the demarcation points of a specific domain among which a local controller must create the cross connection; as the EE device is not present the FC on the *ONOS Access* domain is pointing to the same port (both a UNI and NNI port). Furthermore, since the EVC is a bidirectional channel, for each forwarding construct the global node sends two APIs, each of which represents one direction.

In addition to forwarding constructs each *ONOS Access* receives two APIs: one (*createBwp*) to create the bandwidth profile within the CPE for a given list of c-tags and one (*applyBwp*) to enable the EVC once all forwarding constructs have been sent.

Chapter 6

EVPL Service implementation using ONAP

In the following chapter we first give a description of how we have used ONAP to create an EVPL service and provide a Layer 2 link between two endpoints; then we discuss the components developed and the steps necessary to configure and offer this service.

6.1 Overview

The main idea behind our EVPL service implementation is to take advantage from E-CORD use-case to create a point-to-point EVC between two endpoints and offer a full Layer 2 communication. The endpoints can be of any type, such as datacenters located in different sites that require communication among them or a customer who is offered access to a service hosted in a datacenter.

In this context, the ONAP platform is used as a global orchestration manager to perform the following operations:

- integrate E-CORD into ONAP so that it manages multiple CORD sites and the Transport network between them
- design the EVPL service model in the SDC and distribute it to the Runtime framework
- create a global BPMN that orchestrates the EVPL service creation/deletion, managing all the interactions between ONAP components and external devices (such as Openstack and CORD domains). This approach simplifies and speeds up the service deployment by allowing the use of a single API to the MSO to trigger execution of the global BPMN

- manage the Openstack environment to deploy a virtual machine when a new request to create the EVPL service arrives. This VM acts as one of the endpoints connected through the EVC. It is equipped with a floating IP to enable access from outside the datacenter, in this case from the customer connected to the other side of the EVC

As shown in Fig. 6.1, ONAP knows and manages the whole network topology up to the customer's CPE. This allows us to calculate the best route between the customer's CPE and the nearest provider's CPE, where the datacenter that will host the virtual machine is connected. Another important aspect concerns the network infrastructure that connects the datacenters to the CPEs and how this is managed. In section 6.3 we discuss the design choices made to connect Openstack to a CORD site while maintaining a Layer 2 communications.

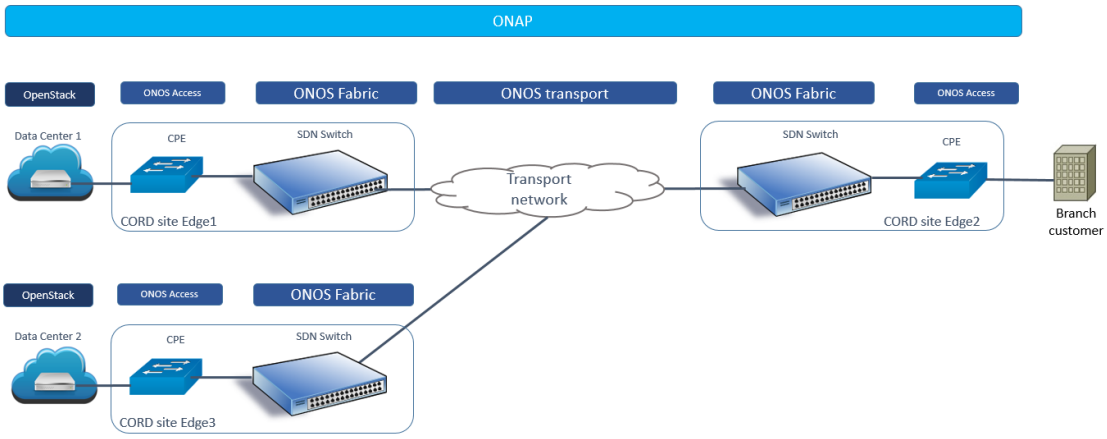


Figure 6.1: ONAP-EVPL infrastructure

The main advantages of adopting ONAP as a global orchestrator compared to an ECORD-type solution based only on the ONOS network controller are the following:

- it is possible to design a service by composing it with TOSCA resources defined in the SDC. This allows us to create complex services that include the management and control of both external network modules, such as ONOS leaf controllers or any other configurable device, and cloud resources such as VNFs
- the TOSCA modeling allows us to describe E-CORD as a reusable resource within the SDC. In this way we can either replicate exactly the E-CORD use-case by deploying a service composed only of this resource or use this resource as a component of a more complex service

6.2 ONAP-ECORD integration

ONAP replaces the ONOS global node of E-CORD becoming the orchestration manager of the leaf ONOS controllers. Using ONAP as a global orchestrator we have exploited the platform capabilities to model an EVC as a TOSCA network resource (called *ECORD EVC*) within the SDC. The properties associated with the TOSCA node will be assigned as input attributes in the resource model and used by the orchestration API. This resource model will be used to guide the orchestration in the MSO, which processes API requests and interacts with the SDNC. The latter is the component responsible to communicate with underlying ONOS controllers; based on the parameters received from the MSO, SDNC sends instructions, via the HTTP-Channel, to the several CORD sites and to the optical transport network for EVCs management.

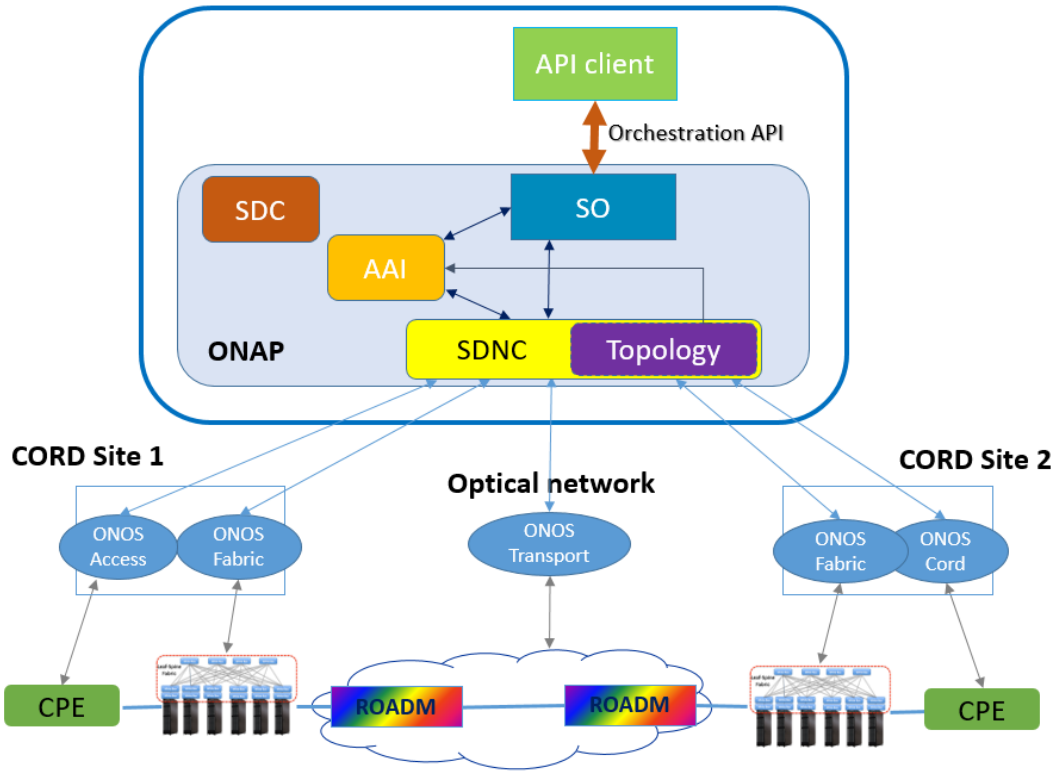


Figure 6.2: ONAP-ECORD infrastructure

Moreover ONAP must be aware of the underlying topology before being able to deploy an EVC. This goal is achieved by using a new component (the Topology service) inside SDNC that is in charge to get the topology from ONOS controllers and store it into the AAI. Through a script execution all ONOS controllers are configured with endpoint

information about new global controller (in this case ONAP) and with underlying devices modellization; in this way, for each ONOS controller the BigSwitch Service can represent its own topology with a single abstract device and send it to Topology service.

The AAI acts as a common database between the components; it is used by the Topology service, which will store the resources representing the network infrastructure, and by SO/SDNC, which will consume the AAI topology and will insert/update service-level information for each EVC addition.

Now we discuss of modellization and changes made on ONAP components (MSO, AAI and SDNC) to support and integrate the E-CORD use-case.

6.2.1 MSO

In order to accomplish the incoming API requests we have designed a BPMN, composed of sub-BPMNs and Groovy scripts, be able to orchestrate the creation/deletion of an EVC. It is associated with the *ECORD EVC* resource model in the Catalog DB such that the API Handler can trigger its execution.

The BPMN workflow performs the following main operations:

- parses informations sent through orchestration API related to the EVC (resource model and network parameters) and checks if it is associated with a service instance in the AAI
- sends a request with EVC service parameters, via the SDNC adapter, to the SDNC provider responsible for the EVC creation/deletion. This request is formatted according to the provider's YANG model
- updates the request status in the Catalog DB and returns a success/failure response to the API client

6.2.2 AAI

The AAI allows us to build an abstract global view of the E-CORD infrastructure by providing predefined data structures that can be used to fully describe the resources and relationships that make up a network infrastructure. Furthermore the AAI provides a set of well-defined APIs through which the SDNC is able to represent the physical topology and the EVCs.

The physical topology is composed of the ONOS Controllers (*ONOS Access*, *ONOS Fabric*, *ONOS Transport*), each of which is modeled using the following AAI data structures:

- **pnf**: a physical network function represents a physical node of the network, in this case an ONOS controller, providing a description of it. A pnf is connected with a complex, an esr, and one or more p-interfaces

- **p-interfaces**: represent physical ports belonging to an ONOS controller. These interfaces are characterized by a name-identifier (in this case a port number), a role (UNI, INNI, ENNI) and other attributes

- **physical link**: is the object used to link two p-interfaces; it describe the physical connection between ONOS controllers

- **complex**: associates a domain to a physical location. A complex has relationships with all pnfs composing such domain

- **esr**: an external system register is used to store connectivity information for a given ONOS controller such as name, url endpoint, credentials and so on. An esr is always associated with a pnf while topology creation

Since each ONOS Controller can handle multiple EVCs, we have modeled each EVC over physical topology as follows:

- **logical link**: is the object that abstract the Cross Connections and used Bandwidth Profile, for an EVC, related to an ONOS controller

- **l-interfaces**: represent the two sides of a Forward-Construct for a given EVC on a given domain; each l-interface is associated with a p-interface, a logical link, and a vlan

- **vlan**: represent the c-tag/s-tag combination related to an EVC

Moreover the service instance, to which an EVC belongs, will be linked to all Forwarding-Constructs across the domains used along the path; this is represented by the relationship to the logical-links (each one is specific to a single EVC on that domain). There is also a configuration object which links to the MDSAL endpoint in SDNC where the EVC configuration is stored. This is purely symbolic as SDNC does not rely on this AAI information, but other components could.

6.2.3 SDNC

As mentioned above, the SDNC has the task of interacting with the several domains to know underlying network topology and manage EVCs. For this reason we have implemented two new services within the SDNC: (1) Topology service that receives, processes and stores the abstract topology seen by each ONOS controller and (2) ECORD-EVC service that handle MSO REST API call to create (activate)/delete (deactivate) an EVC.

Topology service

The Topology service is composed of a provider, described by YANG data modellization, and a Direct Graph, which defines the workflow. No plugins are required since the Topology service will only interact with AAI.

The Topology provider is defined by the following Remote Procedure Call:

```
rpc cord-topology-service {
     description "RPC to receive a cord topology service";
     input {
         list ports {
             leaf port {
                 type string;
                 description "Name that identifies the
                    p-interface, represented by a port
                    number";
                 example:value "3";
             }
             leaf type {
                 type string;
                 description "Indicates the physical
                    properties of the p-interface (e.g.
                    fiber, copper, OCh)";
                 example:value "fiber";
             }
             leaf domainId {
                 type string;
                 description "Unique name of physical
                    network function";
                 example:value "163.162.95.51-onos-cord";
```

```
                }
                leaf interlinkId {
                    type string;
                    description "Indicates the physical link
                       name to which the p-interface is
                       connected";
                    example:value "site1-cpe-fabric";
                }
                leaf mefPortType {
                    type string;
                    description "Role specification for
                       p-interface hardware (e.g. UNI, INNI,
                       ENNI)";
                    example:value "UNI":
                }
                uses annotations;
                description "annotations contains other port
                   details such as port speed, lambda
                   wavelength (for optical domain), etc.";
            }
        }
        output {
            uses response-common;
        }
    }
```

Listing 6.1: RPC for Topology service (YANG Data Model)

For each ONOS controller, the Bigswitch service abstracts a single device and for each device ports (UNI, INNI or ENNI) sends information according to this RPC. After, the provider's Java code calls the Service Logic Interpreter which checks if the DG related with the Topology provider exists, takes it from the SDNC DB and executes it.

The DG workflow performs several operations: (1) save all input data (domainId, portType, portName, etc.) related to a port, (2) creates the AAI modellization of the ONOS controller, to which the port belongs, with the proper data depending on whether the port is tied to a CORD site or to the optical transport network, (3) queries the AAI to know what informations it already contains regarding that specific domain, and stores via AAI APIs the missing data structures (pnf, complex, esr, physical link, p-interface).

ECORD-EVC service

The ECORD-EVC service, like the Topology service, is composed of a provider, a DG and also southbound plugins to the CORD/Transport domains.

The ECORD-EVC provider responds to the following RPC:

```
rpc cord-service-operation {
    description "RPC to create/delete an EVC using
       provided service information and other details
       given in payload";
    input {
        uses sdnc-request-header;
        uses request-information;
        uses service-information;
        uses model-information;
        uses evc-information;
    }
    output {
            uses response-common;
    }
}
```

Listing 6.2: RPC for ECORD-EVC service (YANG Data Model)

The data structures within the 'input' statement describe information received from the MSO (in order to focusing on EVC management, only the *evc-information* YANG model is reported):

- **sdnc-request-header**: identifies the request received from SDNC Adapter. It contains an identifier (*activate* or *deactivate*) representing the action for EVC and the MSO URL endpoint to which send success/failure response
- **request-information**: contains information on the initial orchestration API sent to the MSO
- **service-information**: contains data about the service instance (id, service type), to which the *ECORD EVC* resource model belongs, and the client who has requested the service

- **model-information**: represents the *ECORD EVC* resource model (id, name, etc.)
- **evc-information**: contains all information related to an EVC (Bandwidth Profile, c-tags list, EVC type, UNI endpoint list). This data structure is necessary only for EVC creation

```
container evc-information {
        leaf evc-type {
            type string;
            description "Name that identifies
               the EVC type";
            mandatory true;
            example: value "POINT_TO_POINT";
        }
        container ctags-list {
            description "Set of vlan c-tags to
               be associated with the EVC";
            leaf-list c-tag {
                 type string;
                 mandatory true;
                 example: value "100";
                }
        }
        container uni-endpoint-list {
                list uni-endpoint {
                    description "Pair of UNI
                       endpoint among which EVC
                       will be created"
                    key "endpoint";
                    leaf endpoint {
                        type string;
                        description "The
                           endpoint is defined
                           as
                           pnfName/portNumber";
                        mandatory true;
                        example: value
                           "163.162.95.51-onos-cord/3";
```

```
                                }
                                leaf role {
                                    type string;
                                    description
                                       "Indicates the
                                       user role used to
                                       access the CPE";
                                    mandatory true;
                                    example: value  "ROOT";
                                }
                            }
                        }
                    uses bw-profile;
                    description "bw-profile contains the
                       bandwidth profile for the EVC";
                    example: value  "CBS:100, EBS:10, CIR:10000,
                       EIR:1000";
                }
```

Listing 6.3: evc-information Data Type (YANG Data Model)

When an API request to create/delete an EVC comes from MSO, according to *cord-service-operation* RPC, the ECORD-EVC provider performs several main operations: (1) checks if the service instance is already associated with an EVC and if so retrieves EVC information from MDSAL, (2) calls SLI that checks if the relevant DG exists and executes it, (3) saves EVC service information into MDSAL; these data contain the service instance id, the action for EVC, the EVC information and the EVC service status once DG workflow is finished. The (1) is useful only for EVC deletion in fact through service instance id we can identify the EVC into MDSAL and what needs to be deleted; while for EVC creation the EVC information are provided by MSO (using *evc-information* data structure).

Depending on *action* parameter value inside *sdnc-request-header* data structure, the DG carry out different workflows:

- *action* = **activate**

  The DG fetches AAI physical topology of ONOS controllers, processes data and builds a graph that represent the network. This graph captures capacity availability

for all CORD sites and wavelength compatibility in optical domain; it is composed by a Vertices list and an Edges list: the first list contains the graph vertices, each identified as combination of pnf & p-interface, while the second list contains all links between the vertices (the cross connections between p-interfaces on the same pnf and the physical links between p-interfaces on different devices). After graph creation, a Dijkstra algorihtm is applied on the graph to find the best path among the pair of UNI endpoint received from MSO. If a path is found the DG generates a unique vlan s-tag accross pnfs involved in the path and configure, through southbound plugins, underlying ONOS controllers with bandwidth profile and forwarding constructs; the APIs, sent toward ONOS controllers to create the EVC, are the same APIs depicted in Fig. 5.5 but in this case SDNC plays the role of global ONOS. Finally, if EVC creation is successful, the DG updates AAI service-level information: l-interfaces, logical-links, vlans, service instance relationships and EVC configuration object (which links to MDSAL).

- *action* = **deactivate**

  The DG uses EVC information, retrieved by provider from MDSAL, to build and send the instructions toward ONOS controllers for deleting forwarding consructs and bandwith profile related to such EVC. Then, using the service instance id, it retrieves all EVC relationships from AAI, recalculates the capacity availability for each p-interface and restore vlan s-tag value among those avalaible. Finally the DG deletes all logical links, l-interfaces, vlans and EVC configuration object related to the service instance id.

6.3 Openstack-CPE connection

The connection between datacenter and CPE can be designed in different ways depending on service provider network infrastructure, but always maintaining a layer 2 communication required by the EVPL service. For lab trials we have used a Huawei switch (called gateway) to interconnect an Openstack instance to a CPE:

- the Openstack-gateway connection is made through a VxLAN tunnel so that it is possible to cross whatever layer 3 network among them. Within Openstack Network node we install VxLAN flow rules inside OVS switch (called br-ex) that represents the demarcation point between Openstack network and external network infrastructure
- on other side the gateway has a NIC directly connected to the customer-side port (c-port) of the CPE and this connection is configured manually using MAC forwarding

rules

In order to manage the Openstack-CPE connection using ONAP platform we have (1) modelled VxLAN tunnel as a TOSCA network resource with related properties within SDC, (2) created AAI modellization of physical network topology and logical connections (VxLAN tunnels), (3) added a VxLAN service within SDNC that is in charge for creation/deletion of VxLAN tunnels. The VxLAN resource model will be distributed to the Runtime framework and used by MSO during orchestration workflow to create/delete on-demand VxLAN tunnels.

6.3.1 AAI topology

Openstack-CPE network infrastructures and VxLAN tunnels can be represented via AAI APIs using the same AAI data structures seen in section 6.2.2. Pnfs, p-interfaces, physical links, esr and complexes are used to model the physical network view while logical links, l-interfaces and vlans describe VxLAN tunnels among Openstack instances and the gateways.

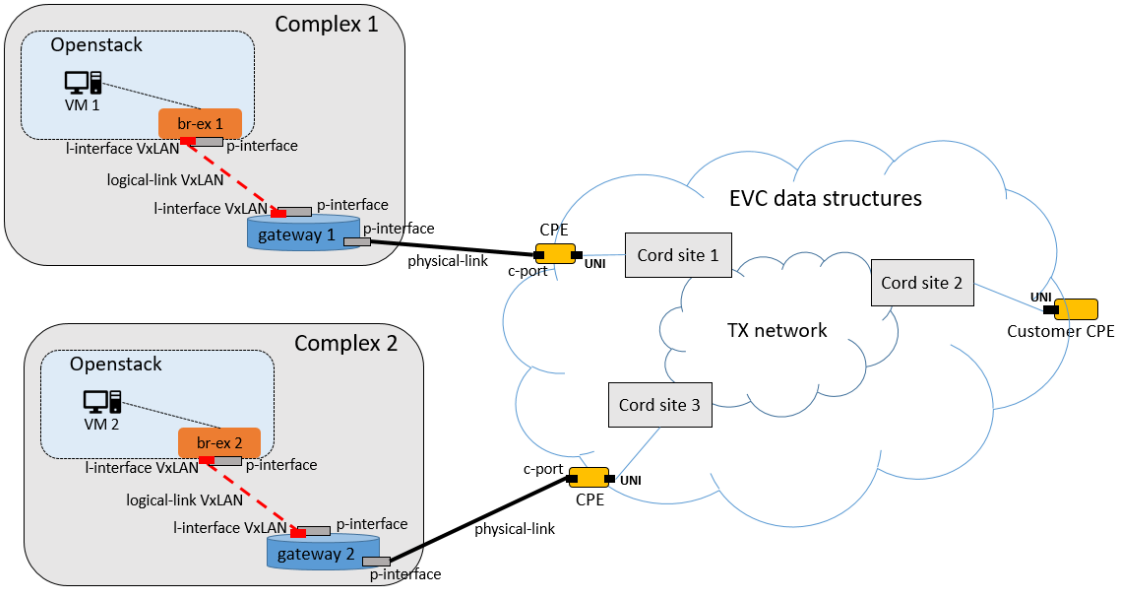


Figure 6.3: Openstack-CPE modellization

Fig. 6.3 illustrates the AAI modelizzation of two complexes connected with CPEs. Each complex represents a physical domain composed of (1) *br-ex* pnf that describes OVS switch within an Openstack instance and (2) *gateway* pnf that describes Huawei switch used to interconnect Openstack to CPE.

6.3.2 VxLAN service

The VxLAN service senses MSO API request for VxLAN tunnel management and interacts with physical devices (br-ex and gateway) of a complex to install/remove VxLAN flow rules. This service is composed of:

- **a VxLAN provider** that processes MSO request, retrieves VxLAN tunnel information (if any) from MDSAL, executes DG workflow and then saves VxLAN tunnel information into MDSAL. The YANG data modellization of the provider defines RPC, parameters and data structures required to create/delete a VxLAN tunnel. For VxLAN tunnel creation the provider receives information on: VxLAN resource model, service instance to which this resource belongs, pnf names among which create VxLAN tunnel, floating IP address of an Openstack VM whose traffic will flow through VxLAN tunnel
- **a DG** which defines workflows for creation/deletion of VxLAN tunnel. The DG retrieves AAI physical topology, selects the relevant complex on which create/delete VxLAN tunnel using pnf names received by MSO and then sends instructions via southbound plugins toward br-ex and gateway of such complex. Moreover, if the workflow execution is successful, the DG updates AAI service-level information (l-interfaces, logical-link, vlans, service instance relationships) related to the created/deleted VxLAN tunnel

6.4 Service design

Within SDC the EVPL service is modeled with custom/default TOSCA resource types and then distributed to the Runtime framework; this enables service model usage by BSS/OSS to create/delete on demand EVPL service instances. The TOSCA resources describe the several components of the EVPL service (cloud resources, EVC, VxLAN tunnel) and guide the orchestration workflow in the MSO.

In order to specify the datacenter that will host the virtual machine we can proceed in different ways: (1) design a single EVPL service model which uses a specific Openstack image and/or flavor in the VM's HOT template that will be triggered by customer service selection, (2) design a single EVPL service model which uses a set of images and/or flavors options in the VM's HOT template; the specific image/flavor will be selected in the portal/MSO according to customer choices, (3) design multiple EVPL service models; each service has the corresponding image and flavor in the associate VM's HOT template to be selected in the portal/MSO according to customer choices. In the latter case, the user will specify the model name of the preferred VNF via MSO API.

For lab trials we have chose the (1) option and composed the EVPL service model with five resources:

- **a virtual machine** that represents host in which will run the service to be offered to the customer. This VM is onboarded as VNF using a HEAT template that describes: static properties (such as public SSH key, image name, flavor, security group, floating IP) and input network attributes which will be provided during Runtime orchestration workflow.
- **a virtual router** connected on one side to Openstack external network and on the other to a private network. This router allows us to assign VM's floating IP and make it accessible from the outside. As for VM, the router is described by HEAT template and onboarded as VNF
- **a virtual private network** used to connect VM to virtual router. The SDC already provides TOSCA model and HEAT template that describe a generic Neutron net in Openstack, so it is not required to design and onboard this resource as VNF. As for VM and router, input network attributes will be assigned during Runtime execution
- **a *ECORD EVC* resource** that represents the EVC between customer's CPE and provider's CPE. The properties related to this TOSCA resource are: UNI endpoints (domains and ports), UNI roles, EVC type, c-tags list and bandwidth profile
- **a *VxLAN* resource** that represents the VxLAN tunnel between provider's gateway and br-ex hosted in Openstack. The properties related to this TOSCA resource are VxLAN endpoints and VM floating IP

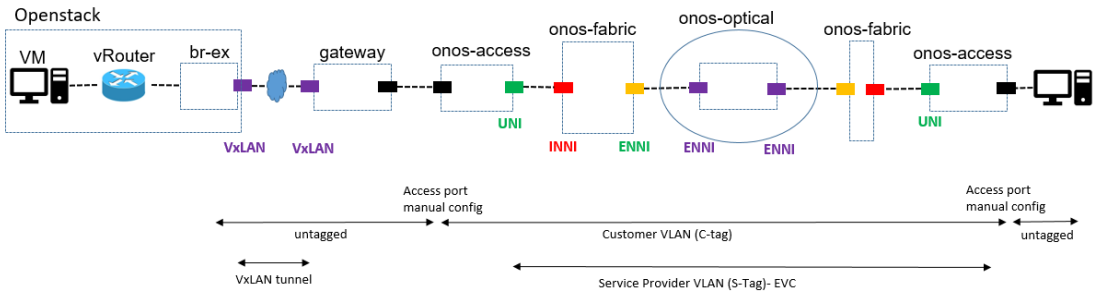


Figure 6.4: EVPL service logical view

After service distribution, the MSO Catalog DB contains all artifacts related to the EVPL service: (1) models of VNFs and VF modules that describe router and VM, (2) models of network resources for EVC, VxLAN tunnel and generic Neutron net and (3) the service model that represent the whole EVPL service and has relationships with all other resource models.

6.5 Orchestration workflow

The deployment of an EVPL service instance is made by running a global BPMN that is responsible of interacting with the several ONAP components and the external devices. This global BPMN defines the logic to select the best datacenter on which deploy cloud resources and the sequence of operations required to create the EVPL network infrastructure. The BPMN workflow is triggered by sending an API, to the MSO, which contains the information related to the EVPL service model and the network parameters to be used; this allows us to create EVPL service instances on the fly and assign them to customers who request the service.

6.5.1 Environment configuration

The global BPMN workflow relies on preloaded AAI information and SDNC providers endpoint to carry out the EVPL service instantiation. For a correct BPMN execution, the AAI must contain information on: (1) list of Openstack tenants, each of ones is related to a list of images, (2) EVPL service label, (3) customers related to the EVPL service subscription and related to the Openstack tenants available, (4) Openstack-CPE modellization for each Openstack tenant available, (5) ONOS controllers modellization. While for network infrastructure operations, the SDNC must contain the following services up and running: (1) the ECORD-EVC service for EVC creation, (2) the VxLAN service for VxLAN tunnel creation, and (3) a new service, called BESTDESTSEARCH, that selects the best complex to use for EVPL service deployment. The latter service receives the customer CPE endpoint, the bandwidth profile for the EVC and a list of gateway endpoint; then it computes all paths via Dijkstra algorithm between customer CPE and each gateway, selects the best path and returns the result to MSO.

6.5.2 BPMN execution

When an API to create EVPL service instance arrives at MSO the API Handler takes charge of the request and triggers the execution of the BPMN associated with the EVPL service model in the Catalog DB. The API request contains all information required to

guide the orchestration workflow: (1) the EVPL service model, (2) the customer UNI endpoint and the network parameters to be used for cloud resources and EVC and (3) the customer that requests EVPL service. These information will be passed to Camunda engine and used during global BPMN execution to manage the several parts of EVPL service deployment. Fig. 6.5 illustrates BPMN structure and main workflow phases; the workflow is composed of a sequence of functional blocks: those with the bold margin represent sub-BPMNs called during execution while the others runs functions defined in Groovy scripts.

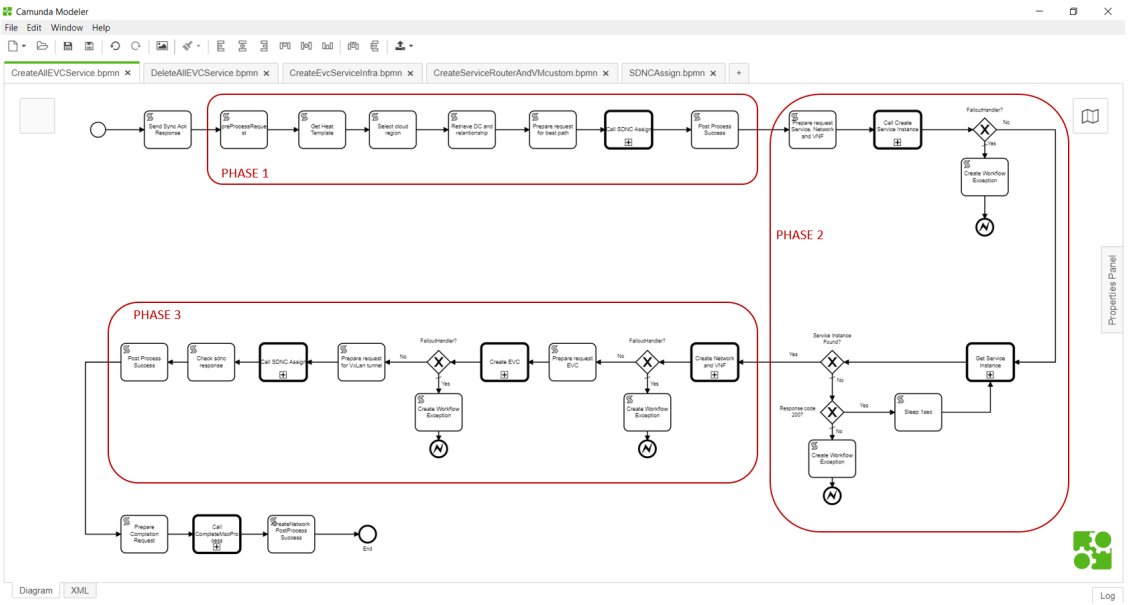


Figure 6.5: Global BPMN

Phase 1 : Best datacenter selection

In this phase the BPMN retrieves the HEAT template associated to VM's VNF from Catalog DB and uses the related image name to find, in the AAI, all Openstack datacenter that are able to manage VM deployment and also the related Openstack-CPE modellization. After, these information are used to send a request to the BESTDESTSEARCH service that computes the best path between the customer UNI and each gateway whose has a relationship with a candidate datacenter. We assume a distance of 0 between the gateway and the br-ex, so the path computation will be between the customer UNI and the interface of the gateway facing the provider network (the Microsemi CPE); path computation must also satisfies bandwidth constraint. The SDNC returns a single gateway as a result of path computation and from the gateway we now know which (1) complex

is the selected candidate to host the VM and (2) provider's CPE has to be used for the EVC creation. At this point all information required for EVPL network infrastructure deployment are avalaible.

Phase 2 : Cloud resources creation

The BPMN uses the EVPL service model to retrieve, from Catalog DB, the several resource models belonging to the service and to perform the following operations:

- create the EVPL service instance in the AAI; this AAI resource is the root node to which will be linked all other resource instances during execution
- create cloud resources (router, virtual LAN network and VM) into Openstack instance selected during phase 1. Firstly, the BPMN makes the network parameters preload into SDNC and instantiate the virtual network through Network Adapter. Secondly, it deploys recursively all VNFs related to the EVPL service model and, for each VNF, it carries out VF modules preload and instantiation through VNF Adapter.

At the end of this phase, the VM is connected to the router through virtual LAN network and equipped with a floating IP that will be used during VxLAN tunnel creation. The AAI will contain the service instance connected to two VNFs and to one network resource.

Phase 3 : EVC and VxLAN tunnel creation

Once cloud resources are created, the global BPMN calls the sub-BPMN responsible for the EVC creation (described in section 6.2.1). This sub-BPMN builds the request for the ECORD-EVC service within SDNC to create the EVC between customer CPE and the CPE (p-interface) plugged into the gateway that has a relationship with the datacenter complex selected previously. The EVC network parameters (customer's UNI endpoint, EVC type, c-tags list, bandwidth profile) to be used come from initial orchestration API while the provider's UNI endpoint comes from the selected provider's CPE in phase 1.

When the EVC creation is finished, the BPMN creates the VxLAN tunnel between br-ex of the selected complex and related gateway. To do this it sends request to VxLAN service inside SDNC with network parameters to setup a logical-link VxLAN tunnel; the parameters contains the endpoints for br-ex and gateway, identified as combination of pnf & p-interface, and the VM floating IP for installing flow rules into physical devices.

After the last phase the BPMN completes its workflow execution updating the EVPL request status in the Catalog DB and sending a success/failure response to the API client

that has generated the initial orchestration API. Furthermore the AAI will contain the tree representation of the EVPL service with the service instance linked to all other resource instances: logical-links representing the EVC and the VxLAN tunnel, two VNFs representing router and VM, one network resource representing virtual LAN network. If any errors occur during workflow execution, the BPMN runs a sub-process to delete all the resources instantiated up to that time and restores the system to a consistent state.

At this point any client, connected to the customer CPE, can access the VM hosted in Openstack through a full layer 2 communication and use the service that the VM exposes.

Chapter 7

ONAP current & future directions

7.1 Key concepts

In this section we provide details on what are the objectives and basic principles on which the ONAP project is based and what ONAP is doing to harmonize open source and standards. We focus on three areas of ONAP-related industry standards and best practices: architecture, model-driven approaches, and APIs.

Architecture

ONAP is a platform above the network infrastructure layer that automates the operation and management of the entire network that is, both virtual and physical network functions. It allows operators to connect their products and services through the infrastructure and scale the network in a fully automated manner. In other words, ONAP aims to provide a utility network abstraction to the business layer, making services that demand just-in-time networking capabilities more attainable. The separation of concerns (SoC) design pattern is key in modeling the scope of ONAP: it is focused on modeling the information and related management functions in the service and resource layers, while its entire ecosystem spans many vertical industries and business scenarios, from end-user products to infrastructure layer.

There are a few ONAP architecture design principles that guide the realization of the platform:

- ONAP creates an open, model- and metadata-driven reference platform for service providers to support full lifecycle management of cloud-centric, software-controlled networks (SDN/NFV). The target goals include: (1) a modular, model-driven, and microservices-based architecture, (2) a layered management architecture including orchestrator, controllers, and multi-cloud (multi-VIM) infrastructure abstractions,

and (3) well-defined APIs for all modules to foster interoperability both within ONAP and across complementary projects and applications

- ONAP must support a common approach to manage various network functions and related lifecycle management from different vendors. This approach includes: (1) all ONAP platform modules must be product/service/resource-agnostic, with a common information model for all vendors to follow, and (2) support standards for consistency across vendor products, such as standard templates for instantiations, standard language for configuration, standard telemetry for monitoring and management, and so on.

- Enable service providers to define and onboard resources to support any type of infrastructure and services, and to define analytics and policies that will be used at runtime. The design goals include: (1) unified models between design-time and runtime modules to facilitate end-to-end, zero-touch operations, (2) well-defined northbound APIs for all modules, and (3) a central design studio where all required artifacts are designed, tested/certified and distributed.

Model-driven approach

Model-driven is a widely adopted principle of IT system design, and often a business requirement in large enterprises or complex ecosystem operations. In this approach, the business logic of the software application is specified through the model at a higher level of abstraction, which is decoupled from the implementation code in a specific programming language. Running code can be generated or behaviours can be changed through model transformation techniques, such as code generation or interpreting/executing the models. Therefore, a model-driven approach enables enterprises to sustain technology changes and gain the agility to support multiple business and service scenarios. For example, within the current ONAP release ('Amsterdam'), only a few modules are using the model generated code, such as A&AI. The majority of the ONAP core modules are "template-driven", i.e., using the common execution engine as a service-independent platform to parse and execute templates for services and resource lifecycle management. Those models/templates are described in domain-specific languages (DSLs), such as TOSCA, YANG, etc. To support service and resource management that is model/template-driven, ONAP features the separation of Design Time and Run Time environments: the Service Design & Creation module (SDC) in Design Time is responsible for the design, encapsulation, certification, and distribution of the related models/templates; the Run Time modules are responsible for parsing and executing the distributed templates.

Moreover, there are four modeling domains in ONAP: deployment, closed-loop, SDN,

and configuration. Further, each domain model can be subdivided into an information model and a data model: the information model describes the concept and the relationship among those concepts at an abstract level, while the data model adheres to the semantics described in the information model with strict syntax specifications within its domain. The data model facilitates system coding without ambiguity. For example, with the template-driven approach, there is no need for any code modification to ONAP when deploying a new service if its deployment requirements can be described with the ONAP information model using ONAP data modeling templates.

APIs

To enable service providers and users of ONAP to quickly integrate ONAP with their existing systems, such as the OSS/BSS, ONAP embraces an architecture with well-defined APIs that fosters interoperability both within ONAP and across complementary projects and applications. The ONAP API design principles include: (1) support for self-service and user-focused business objectives, (2) ease of integration via standardized APIs, and (3) model-driven approach (API code generation instead of static coding per scenario) and agnostic to VNF, resource, product, and service type. There are two categories of APIs in the ONAP platform, which adhere to the above design principles:

- **ONAP External APIs:** These allow ONAP to be viewed as a "black box" by providing an abstracted view of the ONAP platform's capabilities. They can also be used for connecting to systems where ONAP uses the capabilities of other systems.
- **ONAP Internal APIs:** These are APIs exposed by individual ONAP modules with the primary goals of exchanging information with other modules and jointly fulfill the functions provided by ONAP.

The ONAP External API Framework project provides the entry point for external API interfaces for the northbound OSS/BSS interface. It shields the ONAP details from the consumer interfaces as well as providing the consistency required for internal modules, such as authentication and authorization.

7.2 ONAP releases

Amsterdam is the first version of the ONAP project and is the one used for the development of this thesis; it came out in 2017 with the aim to anticipate as much as possible the learning of the main concepts tied to a new way of designing and managing networks: enable VNFs provisioning, facilitate a centralized and simplified Life Cycle

Management of all services, decrease vendor lock-in. The Amsterdam release provides a good coverage of the basic functionalities and the possibility of a strong customization with in-house developments but it is very complex architecturally and not mature from an operational point of view. Indeed, (1) the automation of a service instantiation is still missing, (2) only a cloud provider (Openstack) is supported for the deployment of VNFs and virtual network devices, (3) installation and configuration via Helm charts are complex and not user-friendly and the platform is still unreliable and not high-available, for example power outage highly impacts infrastructure (Kubernetes and Openstack) and ONAP stability; also re-installing Amsterdam ONAP is not always an easy task because of software changes that do not grant backward compatibility or issues to access remote images repository.

The activities of the ONAP community are articulated around Projects and Releases. The Release Lifecycle should be considered as a sub process of the overarching Project Lifecycle. The latter should be seen as a long term endeavor whereas the Release Lifecycle has a short term goal. From 2017, ONAP has adopted a 6 months release cadence for Release Lifecycle. Fig. 7.1 shows the releases following the first version, Bejing and Casablanca, compared according to the improvements and features offered, on a scale from 0 to 10.

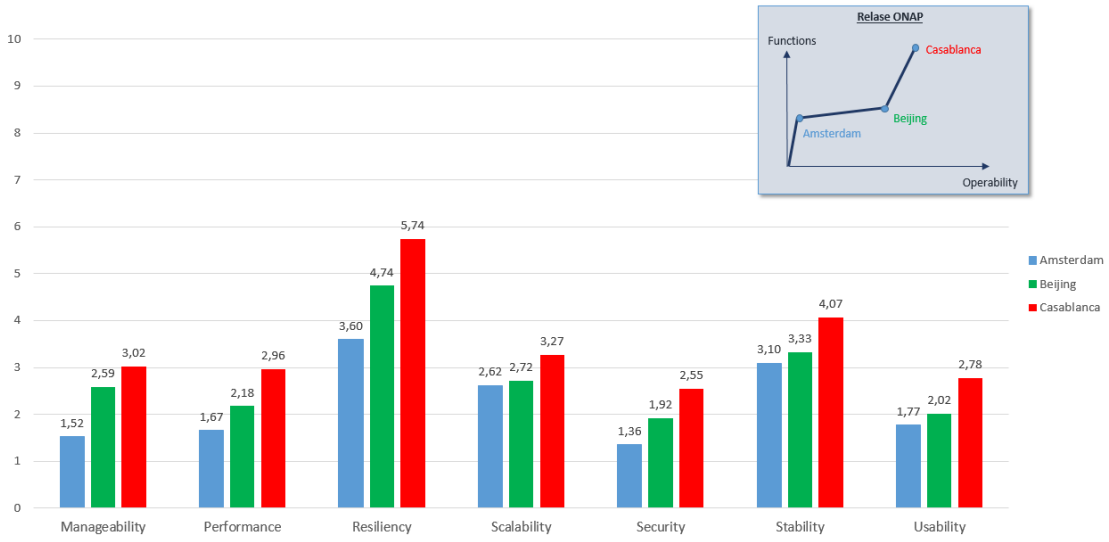


Figure 7.1: ONAP Releases

ONAP Casablanca is the project's third release and has been enhanced with respect to maturity, policy-driven orchestration, ETSI based NFV onboarding, stability and performance in support of real world deployments. Moreover, it brings additional support for

cross-stack deployments across new and existing use cases such as virtual CPE (vCPE), virtual Firewall (vFW), 5G and Cross-Carrier VPN (CCVPN), as well as enhancements to cloud-native VPN. The 5G blueprint is a multi-release effort, with Casablanca introducing the first set of capabilities around PNF integration, edge automation, real-time analytics, network slicing, data modeling, homing, scaling, and network optimization.

Casablanca also includes new features, architectural changes, deployability enhancements and bug fixes. Some of these highlights include:

- the design time environment includes two new dashboards to simplify design activities
- the VNF testing is enhanced to help ease deployment pains and improve VNF quality and interoperability across real-world deployments
- the runtime environment includes new lifecycle management functions in both the Service Orchestrator (MSO) and its three controllers, expanded hardware platform awareness (HPA) to improve performance, geo-redundancy, support for ETSI NFV for VNFM compatibility, MultiCloud enhancements, and edge cloud onboarding

7.3 ONAP evolution in TIM labs

In 2018, TIM company in its Innovation labs began to get in touch with the ONAP project to understand what it is, what capabilities it enables and how it could improve network management and the deployment of high-available and scalable services. Therefore, we decided to use the ONAP Amsterdam release to (1) start learning about the platform with respect to architecture, cloud infrastructure required to host ONAP, installation procedures and features offered and (2) design and deploy a new service (the EVPL service discussed in chapter 6) over the ONAP framework.

After these first steps, TIM's work on ONAP has evolved focusing on the following main objectives:

- **installation improvements**: creation of scripts that automate the installation and manage all the software infrastructure on which ONAP is deployed. In this way it is easier to deal with the software upgrade versioning and the change from one ONAP release to another. Furthermore, these scripts use local repositories in order to avoid problems accessing remote image repositories and accelerating the configuration of ONAP

- **CI/CD automation**: is a method for the frequent deployment of apps to customers, which involves the introduction of automation in the application development phases. Mainly, it is based on the concepts of continuous integration, distribution and deployment. The CI/CD approach overcomes the difficulties associated with the integration of new code by introducing constant automation and continuous monitoring throughout the application lifecycle, from integration and test phases to distribution and deployment phases

- **Flex Communication Service**: is a new service intended to provide a flexible service for enterprises: it offers either L2 connectivity or L3 connectivity or both ones. No constrain is set for customer that is free to start the service as a pure Internet access of the Headquarter or as a layer 2 service among the headquarter and one or more subsidiaries. At any time customer may request Internet access for some of its sites, a vFW is instantiated and configurated and some relevant parameters are monitored in order to make vFW scale-out/scale-in happen according to configuration

Chapter 8

Conclusion and Future work

8.1 Conclusion

The first conclusion we can reach is that using ONAP as a global orchestration platform we are able to design and offer on the fly high-avalaible and scalable cloud services and also manage whatever network infrastructure, from physical devices to datacenters, cloud environments and domain controllers. The modular and layered nature of ONAP allows us to describe/integrate every single component of a service within the platform itself and to define custom business process models to orchestrate the service deployment. For our EVPL service we have exploited the features of only 4 components of the many that constitute the ONAP platform:

- **SDC** has been used to design and compose the service with all its components and distribute it to the Runtime framework
- **MSO** has been used to define, through BPMN business models, the entire orchestration and interact with AAI/SDNC to make up the service
- **SDNC**: has been used to deal with all the networking system, from the preloading of network parameters to the interactions with ONOS controllers, Openstack environment and Huawei switch for configuring the network infrastructure
- **AAI** has been used as a common database between the other components (SDC, MSO, SDNC) to store information about: the service (such as customers, Openstack environments, and so on), the entire network topology and all the resources instantiated (and the related relationships) during the global BPMN execution

As described in chapter 6, our EVPL service allows to manage the whole connection from the customer's CPE to the service provider's datacenter, configuring the cloud

resources within Openstack, the connection between Openstack and the provider's CPE, and the EVC between the provider's CPE and that of customer.

8.2 Future work

From the development of our service prototype, we have begun to understand how ONAP operates, some of the capabilities it offers and the main interactions between the platform components. In the next months, we are going to update the EVPL service in order to improve its features and performance by using other ONAP components and adding, if required, other software modules.
Furthermore, we want extend this service with the DCAE component to collect, ingest, transform and store data for analysis and provide the ability to detect anomalous conditions in the network, for example, fault conditions that need healing or capacity conditions that require resource scaling. The gathered data is distributed to various analytic micro-services, and if anomalies or significant events are detected, the results trigger appropriate actions.
Finally, given the frequent updating of the platform by the ONAP community, we want to keep up with ONAP releases and continuously integrate our service into them.

Bibliography

[1] SDN, *Software-Defined Networking: A Comprehensive Survey* , Diego Kreutz, Member, IEEE, Fernando M. V. Ramos, Member, IEEE, Paulo Verissimo, Fellow, IEEE, Christian Esteve Rothenberg, Member, IEEE, Siamak Azodolmolky, Senior Member, IEEE, and Steve Uhlig, Member, IEEE.

[2] NFV, *Network Function Virtualization: State-of-the-art and Research Challenges* , Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, Raouf Boutaba, 2015.

[3] ONAP CaseSolution Architecture, URL: https://www.onap.org/wp-content/uploads/sites/20/2018/06/ONAP_CaseSolution_Architecture_0618FNL.pdf.

[4] OpenDaylight: Towards a Model-Driven SDN Controller Architecture, URL: http://dx.doi.org/10.1109/WoWMoM.2014.6918985.

[5] Apache Kafka, URL: http://kafka.apache.org/.

[6] Portable Cloud Services Using TOSCA, URL: http://dx.doi.org/10.1109/MIC.2012.43.

[7] YANG Data Model, *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*, Björklund, Martin, 2010.

[8] Maven, URL: https://maven.apache.org/.

[9] BPMN, *BPMN Modeling and Reference Guide: Understanding and Using BPMN* , Stephen A. White, PhD Derek Miers, 2008.

[10] The Camunda BPM Manual, URL: https://docs.camunda.org/manual/develop/.

[11] Openstack, URL: https://docs.openstack.org/.

[12] Kubernetes, URL: https://kubernetes.io, Google,2014.

[13] Rancher, URL: https://rancher.com/.

[14] Helm, URL: https://helm.sh/.

[15] CORD: Central Office Re-Architected as a Datacenter, URL: http://dx.doi.org/10.1109/MCOM.2016.7588276.

[16] XoS, *Xos: An extensible cloud operating system.* In Proceedings of the 2Nd International Workshop on Software-Defined Ecosystems, Larry Peterson, Scott Baker, Marc De Leenheer, Andy Bavier, Sapan Bhatia, Mike Wawrzoniak, Jude Nelson, and John Hartman. BigSystem '15, pages 23–30, New York, NY, USA, 2015.

[17] ONOS, *Onos: Towards an open, distributed sdn os.* In Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, HotSDN '14, pages 1–6, New York, NY, USA, 2014.

[18] Apache Karaf, URL: http://karaf.apache.org/.

[19] MEF, URL: https://www.mef.net/resources/technical-specifications.

[20] E-CORD, *Network Infrastructures for Highly Distributed Cloud-Computing*, PhD Francesco Lucrezia, pages 46-58, Turin, 2018.