UNIVERSIDAD AUTÓNOMA DE MADRID

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación (EUR-ACE®)

ESCUELA POLITÉCNICA SUPERIOR

TRABAJO FINAL DE GRADO

Transport API development and validation for the Netphony environment

Rodrigo José Jiménez Ríos

Junio 2018

Transport API development and validation for the Netphony environment

Autor: Rodrigo José Jiménez Ríos

Tutor: Víctor López Álvarez

Ponente: Jorge López de Vergara Méndez

RESUMEN

Este proyecto final de grado, contribuye al desarrollo e implementación del servicio de conectividad y el servicio de topología de la Transport API (T-API) en el conjunto de módulos Netphony, desarrollado por Telefónica I + D, y su posterior validación. Netphony sigue una arquitectura Application-Based Network Operations (ABNO) que se puede definir como una arquitectura basada en la colaboración entre diferentes elementos, entre los cuales los principales son el controlador ABNO y el PCE, para procesos de automatización en la gestión de una red, como configuración de rutas LSP, lo que hace que la red sea más escalable y dinámica. Este es el concepto que se persigue con las redes definidas por software (SDN), para lograr una red totalmente programable y la capacidad de modificar cualquier aspecto de la red automáticamente.

Las pruebas de validación y los test de rendimiento con el controlador de Netphony, se han realizado en un entorno emulado de nodos GMPLS, con OSPF y RSVP para llevar a cabo la configuración de los LSP entre los nodos, conformando así el entorno emulado que actúa como una simulación de red de transporte óptico de Telefónica España.

El estándar T-API cumple los requisitos para convertirse en el NBI (North Bound Interface) del Netphony como orquestador SDN. Las características principales de este estándar es su simplicidad y usabilidad para extender su adopción en diferentes tipos de redes de transporte óptico.

Una primera parte de este trabajo consiste en realizar una documentación de las tecnologías usadas y del estado actual de las mismas, para posteriormente presentar como se ha integrado la T-API en el controlador Netphony, casos de uso y definición de los test de validación. Finalmente se ha procedido a desarrollar y compilar el código implementado, configurar la creación de LSPs y realizar una evaluación de rendimiento.

PALBRAS CLAVE

Multi-layer networks, ABNO, PCE, SDN, control plane, programmability, Transport-API, Netphony.

ABSTRACT

This final degree project contributes to the development and implementation of the connectivity service and the topology service of the Transport API (T-API) in the set of Netphony modules, developed by Telefónica I + D, and then the validation of that. Netphony follows an Application-Based Network Operations (ABNO) architecture which can be defined as an architecture based on collaboration between different elements, for automation processes in the management of a network , as configuration of LSP routes, which makes the network more scalable and dynamic, where the main elements are the ABNO controller and the PCE. This is the purpose with software defined networks (SDN), to achieve a fully programmable network and the ability to modify any demand of the network automatically.

Validation and performance tests with the Netphony controller have been performed in a GMPLS nodes emulated environment, with OSPF and RSVP to carry out the configuration of the LSPs between the nodes, thus conforming the emulated environment of Netphony GMPLS.

The T-API standard meets the requirements to become the NBI (North Bound interface) of the Netphony. The main characteristics of this standard is its simplicity and usability to extend it in different types of transport networks. This work focuses on the implementation for optical networks.

Documentation of the technologies used and the current state of the same is the first part of this work, to then present how the T-API has been integrated into the Netphony controller, use cases and definition of the validation tests. Finally, the implemented code has been compiled, the creation of LSPs has been configured and a performance evaluation has been carried out.

KEYWORDS

Multi-layer networks, ABNO, PCE, SDN, control plane, programmability, Transport-API, Netphony.

ACKNOWLEDGEMENTS

First of all, I want to thank my family. To my parents to give me the opportunity to study the university career I wanted, thank for the effort to give me the best education and for being patient with me, and to my sister for always support me in everything I have done.

Second, I want to thank my friends at the university, Jaime Aragón, Daniel Martin, Manuel Iglesias y Juan de Santiago, for making my university life easier and more fun. For helping me and being great practice partners.

I wanted to thank my supervisor, Víctor López, for trusting me and giving me the opportunity to carry out this project and be a reference for me professionally. I will not forget it.

Thanks to my internship colleagues in Telefonica I+D, Nico and Arturo. It has been a pleasure to be able to work with all of you.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ACRONYMS

API	Application Programming Interface
ATM	Asynchronous Transfer Mode
DXC	Digital Cross Connect
E2E	End-to-End
EP	End Point
ERO	Explicit Route Object
GMPLS	Generalized Multiprotocol Label Switching
IGP	Internal Gateway Protocol
IP	Internet Protocol
LER	Label Edge Router
LSP	Label Switch Path
LSR	Label Switch Router
ML	Multi-Layer
MPLS	Multiprotocol Label Switching
NBI	North Bound Interface
NE	Network Element
NFV	Networks Function Virtualization
NMS	Network Management System
NNI	Network to Network Interface
ONF	Open Network Foundation
OTN	Optical Transport Network
OSPF	Open Shortest Path First
OXC	Optical Cross Connect
PCE	Path Computation Element
PCEP	Path Computation Element Communication Protocol
PXC	Photonic Cross Connect
QoS	Quality of Service
RSVP	ReSerVation Protocol
RSVP-TE	ReSerVation Protocol Traffic Engineering
SDH	Synchronous Digital Hierchary
T-API/TAPI	Transport API
TE	Traffic Engineering
TED	Traffic Engineering Database
TID	Telefónica I+D
UNI	User Network Interface
WDM	Wavelength-Division Multiplexing
DWDM	Dense Wavelength-Division Multiplexing

1. Introduction

1.1. Motivation

At first, I would like to start showing the relevance to work in a technological environment as is the Telefonica I+D premises, involving in European projects and working with highly skilled people. This has helped me to improve as professional and in my personal growth. Thanks to this experience I have learned to develop further in some programming language as Python, Java, REST API and command console of Linux, hand-in-hand with truly experts. In other hand, I also have learned to draft technical related documents with several themes (e.g. SDN, IP/Optical networks, techno-economics studies...etc.) inside in a European framework and that is the main reason why I feel comfortable drafting this in English. For all the reasons mentioned above, the opportunity to make a scholarship in TID has helped me to focus in a more precise and concrete way my professional orientation, approaching technologies related to transport networks, SDN, NFV and virtualization, in a company which was awarded, among other prizes, for the best NFV/SDN solution at LTE & 5G World Awards in 2016 thanks to the work carried out on SDN technologies.

Nowadays, conventional networks have rules already integrated in the proprietary switch firmware, which tells it where to transfer the packets, treating all packets in the same way. In the business world there are intelligent switches designed to treat different types of packets differently, however, these switches can become very costly. These networks are a key element when planning IT architecture in companies. Its optimization is fundamental to ensure that the different IT resources work in a synchronized way. However, the difficult in the operation and management is increasing in the new scenario of exponential growth of data, business mobility and consumption of online apps. Network virtualization is the next step in network evolution.

Software Defined Networks (SDN), is a new paradigm with the solution to address the issue in the creation of networks which the control and management are given to a software application, called orchestrator or controller, thus detaching itself from the hardware. This not only configures, but also centrally programs the entire structure without needing to go node by node. Some of the main advantages of a virtualized network are:

- ability to dynamically manage bandwidth;
- greater agility in the deployment and control of networks;
- monitoring of networks and users;
- analytics.

The SDN is just one step more towards the network softwarization, where all the components software-defined are virtualized and separated from physical hardware (e.g. computing, storage, networking and security).

Netphony is a suite of modules developed by Telefonica I + D and forming an SDN tool following the ABNO (Application-based network operations) model. Netphony suit is composed by several modules, all of them, open source and distributed in repositories of Github. The modules are Topology Module, Path Computation Element, Provisioning Manager, ABNO Controller, BGP-LS Peer, Netphony GMPLS emulator. Here, therefore, lies the new hope for multi-vendor interoperability in multi-tenant network architectures where Telefonica pretends to be a referent with the Netphony. The Netphony has been involved in European projects as ACINO (Application-Centric IP/Optical Network Orchestration) which main goal is to develop and demonstrate a modular open-source orchestrator for IP/Optical networks, filling the gap between large applications and the control solutions of current transport networks in terms of bandwidth and also according to the others parameters that would benefit the treatment at the IP and at the optical layer.

As already mentioned, the best solution is to use Software Defined Networks with the main purpose of facilitating the implementation of network services in a more dynamic way. In this type of networks, everything revolves around the separation of the data forwarding plane and the control plane. To perform this process, it is not new in transport networks, based on application programmer's interfaces (APIs) for the network control. The Transport API (T-API) is an interfaces that pursues the following network control services: Topology Service, Connectivity Service, Path Computation Service, Virtual Network Service and Notification Service. The T-API meets the requirements to become the North Bound Interface (NBI) of the ABNO controller of the Netphony. The main features of this standard are its simplicity and usability to extend its adoption.

1.2. Objectives

The purpose of the project is divided in the next objectives:

1. Study and analyse the Netphony framework, its modules and tools to development and extend its functionality.
2. Study and analyse the state of the art of the PCE and SDN control for optical transport networks.
3. Extend the functionality of the Netphony framework to support the Transport API.
4. Demonstrate and experiment the Transport API implementation with real equipment and an emulated environment.

1.3. Structure

The following milestones are defined for the elaboration of the previously mentioned objectives. The duration of each milestone can be affected by the different problems that may arise during the development of the project, so the working times are indicative:

1. Study and analyse the state of the art of the SDN, the ABNO architecture and the Transport application programmer's interface. (1 week)
2. Understand and familiarize with the previous tools for the project and the development environment: Eclipse, Maven, Git, Travis and Swagger (1_week).
3. Understand and familiarize with the environment and the repositories of the Netphony suite, as well as the operation based on the ABNO model and the different plugin developed so far. (2 weeks).
4. Download, compile and run the self-generated code for the Transport API and integrate it into Netphony Project (4 weeks).
5. Test the integration and carry out the implementation of the necessary methods and classes as well as the realization of the query (4 weeks).
6. Performance evaluation with functional tests with different network requirements and different network environment. (2 weeks).
7. Efficiency tests (2 weeks).
8. Writing the project for delivery to the school (4 weeks). Using the work done during the project will write the document for the defence.

2. Software-Defined Networking for Optical Networks

In this section we will first make a brief introduction to the SDN and the motivation to deploy them, the architecture of this type of networks that explains the main key of this model, which consists of separation of the data forwarding plane and the control plane. Next, GMPLS [1] protocol as an extension of MPLS is going to be discussed, its advantages and the great benefits it brings to the network, for then discuss about the Netphony emulated environment acting as a transport network emulator. After is going to be presented the PCE-GMPLS based control as a communication between the PCE and the GMPLS emulated network to calculate the route between nodes or to create remotely the LSP (Label Switch Path) [2], depending the PCE mode is being used. Finally, the ABNO [3] (Application-Based Network Operations) is going to be presented as a functional architecture, is not a software design, offering interoperability between implementations of key components, and easy interaction with the applications and with the network devices.

2.1. SDN architecture based control

Software-Defined Networks (SDN) has been created with the main objective to have an absolute programmability of the network and the ability to modify any aspect of its operation automatically. The SDN concept is based on the idea of decoupling the control and data plane. This is not a new concept cause this concept is inherent to the optical networks using out-of-band channels to carry the control protocols since the beginning. Network Management Systems (NMS) can act in a similar way as controller of the network too, configuring and maintaining the optical equipment, but not in a real-time way and this communication between the network and the NMS was proprietary. Thus, the expansion of SDN must go hand in hand with the use of open and standard interfaces to allow interoperability.

The objective with SDN is to simplify the operation and costs of network management. The current network, which is a very static network, will evolve into an intelligent network, being scalable and reprogrammable. The need for SDN implementation, increasingly to the great operators who have adopted this network architecture design, is to be prepared for the explosion of data traffic led by video, cloud services, etc. In fact, in the last two Mobile World Congress the main topics were SDN and NFV (Networks Function Virtualization) [4], and how to make them a reality. Before continuing with the advantages of SDNs, a brief explanation is presented to contextualize what NFV is and the differences between NFV and SDN. As Figure 1 shows, SDN's general objective is to enable IT managers to respond quickly to business needs, through centralized software-based control, while NFV (Network Functions Virtualization) was developed by operators and aims to accelerate the deployment of new services (NAT, firewall, IPS, DNS, increased cache, etc.)

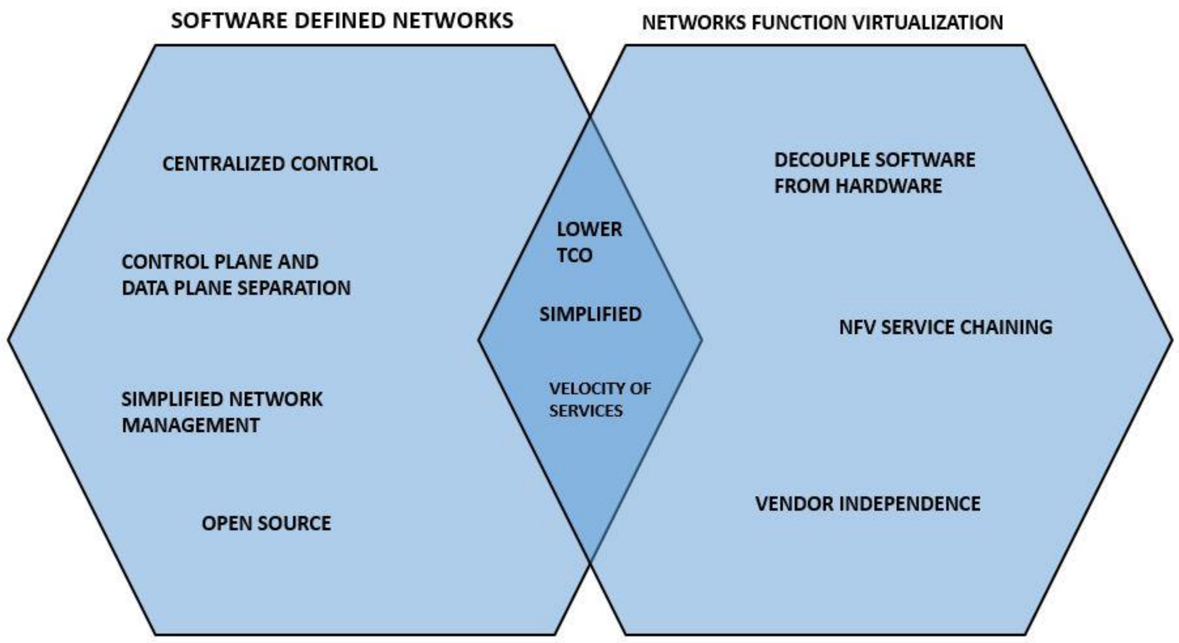

SOFTWARE DEFINED NETWORKS
NETWORKS FUNCTION VIRTUALIZATION
CENTRALIZED CONTROL
CONTROL PLANE AND DATA PLANE SEPARATION
SIMPLIFIED NETWORK MANAGEMENT
OPEN SOURCE
LOWER TCO
SIMPLIFIED
VELOCITY OF SERVICES
DECOUPLE SOFTWARE FROM HARDWARE
NFV SERVICE CHAINING
VENDOR INDEPENDENCE

Figure 1 SDN and NFV comparison

The great advantages of SDN is that it maximizes network flexibility, performance, scalability and quality and all this can be done without human intervention, since the network allows applications to request services through API and it is the network that in turn informs new applications.

The SDN architecture, specified by the ONF [5] (Open Networking Foundation), which is a non-profit industrial consortium which currently leads the evolution of SDN services in addition to performing a standardization work, considers the following layers at a high level. Figure 2 shows the SDN architecture and the division on the different layers based on the restriction of the ONF.

- **Infrastructure layer or data plane.** It is formed by the nodes of the network and in this layer only the hardware that composes the nodes is considered, not the protocols they contain. These physical devices must have installed the necessary support to establish communication with the controller, such as the OpenFlow protocol.
- **Control layer or control plane.** In the control layer is the controller or controllers, responsible for maintaining and managing network intelligence. They communicate with the infrastructure layer through some protocol such as OpenFlow and with the application layer through API.
- **Application layer.** In the application layer, the behaviour of the network is controlled and it is the highest-level layer. To carry out this management, the controller's API, used by the user or by a Software Orchestrator, is used.

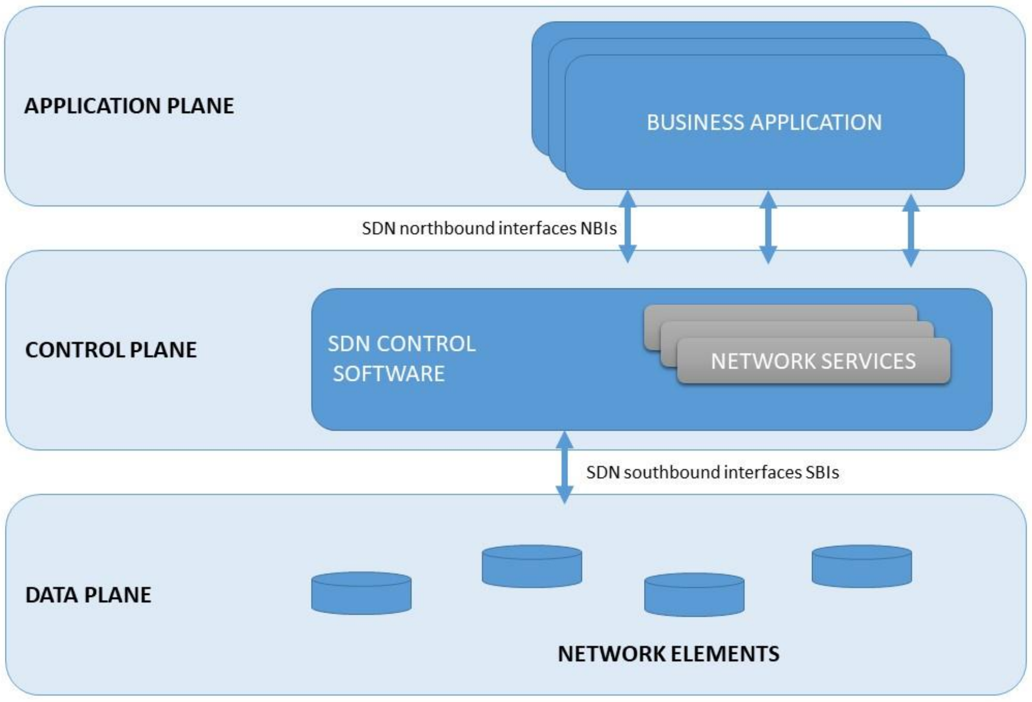


Figure 2 SDN Architecture

However, it should be noted that most of the SDN solutions that are available today in the market are based on a single domain and are aimed at a single provider. This can not be a case of real use, since the networks are composed in combinations of multiple technologies, with multiple domains and the solutions should be oriented to different providers being open-source solutions. On the other hand, in this type of SDN architectures, it is possible to deal with multiple South Bound Interfaces (SBI) such as, for example, the OpenFlow [6] or GMPLS protocols, of which we have already mentioned.

As a solution to this, the ONF proposes hierarchical architectures that fit a multi-vendor / multi-domain scenario, with several SDN controllers coexisting with an SDN orchestrator over them.

In an architecture of this type an SDN Orchestrator would be connected to SDN controllers of different domains, communicating with them through North Bound Interfaces (NBI) and in turn these, with their corresponding network through SBI. The implementations of this type of orchestrators depend on the provider, but there are already open source approaches such as ODL, ONOS [7] or the one that this work deals with, Netphony whose implementation is being carried out in Telefonica I + D.

Following these premises and knowing that an SDN controller should be focused on interoperability with several domains and several providers, the Netphony suite of modules, which will be discussed later, is open source and has implemented several NBI, such as COP, IETF, UNIFY or T-API. In this work, the NBI T-API has been implemented and then the following Figure 3 tries to show the correlation of the different elements of the SDN architecture model proposed by the ONF, with the elements that will be discussed in this paper: scripts generated in python acting as an application plane generating the requests through HTTP and sending them to the plane control through the T-API, where the Netphony will be, which will process them thanks to, among other services, the connectivity service and the topology service and to the PCE for the calculation of the routes. Finally, for the establishment of the connections between the nodes of the GMPLS emulated plane, PCEP messages will be used. The emulated GMPLS environment can not be considered within the pure control plane, since there is communication between the nodes emulated by RSVP messages, which is why it is considered to have some intelligence.

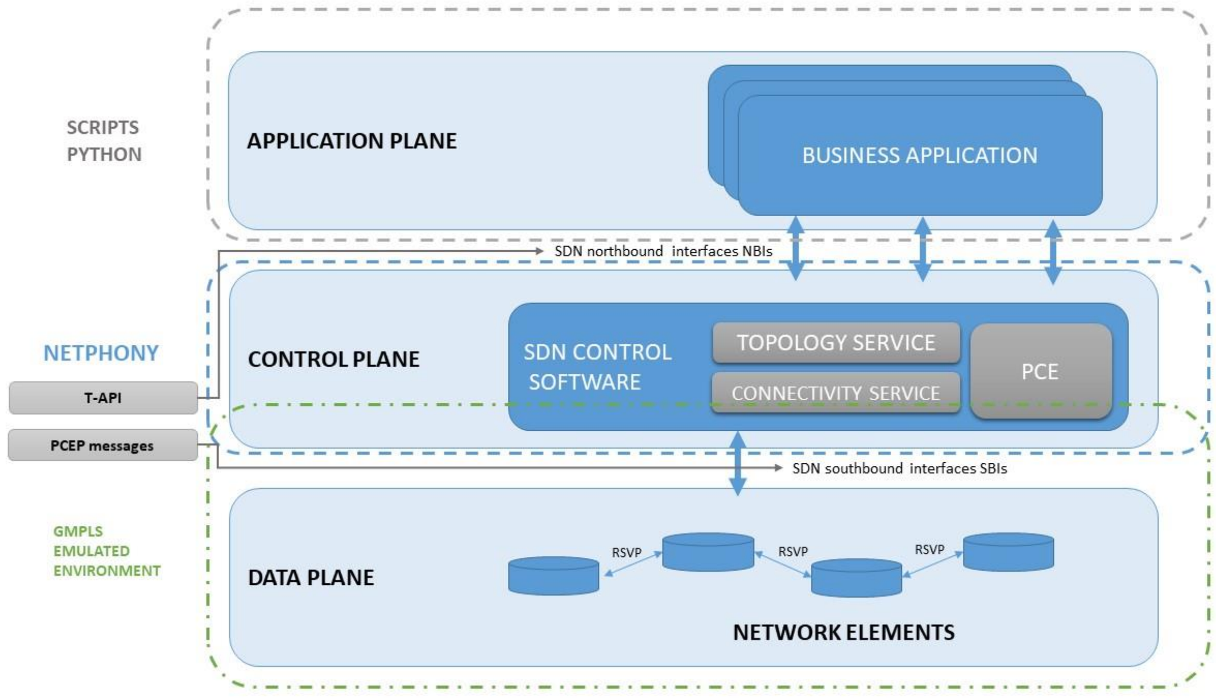


Figure 3 Netphony in SDN architecture

2.2. Distributed GMPLS Control

Generalized Multi-Protocol Label Switching (GMPLS) is an extension of the MPLS protocol (MultiProtocol Label Switching), before presenting GMPLS the operation and objectives pursued by MPLS will be briefly described [9]. The initial objective of MPLS was to provide some of the characteristics of networks oriented to connection to non-connection-oriented networks, thus allowing a single IP network to offer all kinds of services. MPLS combines the advantages of intelligent level 3 routing with fast level 2 switching (IPv4 and IPv6), using packet switching by a small fixed length label, achieving greater performance in the transport of IP packets. This label is assigned to the package following criteria such as the destination address, service type parameters or other criteria. GMPLS is an evolution of MPLS that supports not only packet switching, but also switching over time, wavelength and optical fibers. GMPLS covers, in addition to IP routers and ATM switches, switching devices such as: DXC (Digital Cross Connect), OXC (Optical Cross Connect) and PXC (Photonic Cross Connect). For this, GMPLS extends basic functions of the traditional MPLS and, in some cases, adds new functionality. These adaptations have led to the creation of G-LSP (Generalized LSP), also affecting the routing and signalling protocols for activities such as: label distribution, traffic engineering, and protection and restoration of links.

Thanks to GMPLS, which adds additional information to the IGP protocols, such as OSPF [8] (Open Shortest Path First), are able to provide information about the topology and availability of optical resources of the network. Routing algorithms based on restrictions are considered by GMPLS to use the topological information and available resources to calculate paths through the network and make optical connections. Once a path is selected, GMPLS extends the protocols for signalling and traffic engineering, such as RSVP (ReSerVation Protocol), for the creation of the LSPs along the optical network. These protocols are used to establish, modify, eliminate or recover the information of the LSP.

Finally, it should be noted that the great utility of GMPLS is based on the ability to offer a rapid provision of services, at any time, with any QoS and with any destination. In addition, modern networks are gradually evolving to a transport network model with only two layers (IP and WDM), thus eliminating the complex transport network model of end of the nineties (ATM, SDH).

To perform the tests within a GMPLS environment and to carry out the creation of LSP with the Netphony Controller, an emulated environment of GMPLS nodes has been generated. This emulated environment supports multilayer scenarios and consists of the ABNO controller and a set of Java-based libraries that allows the implementation of a GMPLS control plane, so we refer to this architecture as the Netphony GMPLS emulated environment acting as a transport network emulator. It has an OSPF-TE daemon to send and receive the packets and an RSVP daemon to be able to carry out the configuration of the LSPs between the emulated nodes. This emulated network, in order to communicate with the PCE (Path Computation Element), maintains a connection through PCEP (Path Computation Element Protocol) and that if used in stateless mode, it will only be used for route queries, while if it is used in stateful mode it can be used for remote configuration of LSPs. Each node of the optical emulation is based on a Java implementation to emulate a transport node with GMPLS capabilities, with RSVP-TE processes and based on OSPF-TE, in addition to the PCEP interface.

2.3. ABNO

Nowadays, the networks integrate multiple technologies that allow offering a variety of services adapted to the characteristics and demands of different applications. There is an increase in traffic due to the new services offered by the Internet, which demand connectivity, reliability and resources in a range of network technologies from the IP / MPLS layer to the optical layer. That is why during the last years a great effort has been made in the definition of the control plane for transport networks. However, when there are several manufacturers and technologies, the interoperability between the different domains or layers becomes complicated. An important point to focus on is how to manage the control plane. As there is so much heterogeneity in technologies, defining an automation of the control plane is essential to avoid possible problems and reduce the operating costs of the network (CAPEX and OPEX). From the IETF, the Application-Based Network Operations (ABNO) architecture is proposed, it is not a software design, which allows operations to be carried out automatically, which in today's non-automated networks require manual intervention. ABNO can provide the following types of service to applications: Optimization of traffic flows between applications, remote control of the components of the network, CDN (Content Delivery Network) interconnection, coordination of network resources and VPN (Private Virtual Networks).

Then, the ABNO is a network architecture composed of several modules and technologies that gather information about the topology and the resources available on the network, with the aim of providing routes for traffic and reserving the necessary resources for them. ABNO can be seen as a collaboration between a series of existing elements and new architectural elements that are coordinated to manage connections within a network in an optimal and automatic way. The key component in the ABNO architecture is the Path Computation Element (PCE), which is responsible for computing the routes and is also extended to provide policies and restrictions in the calculation of them. This section describes the ABNO architecture, its components and the connection between them with particular emphasis to the PCE based on the GMPLS control.

Figure 4 shows the ABNO architecture modules. The main component of the architecture is the ABNO controller and is responsible for managing, controlling and organizing the workflows and calling the necessary components in the correct order, depending on the action to be taken at each moment.

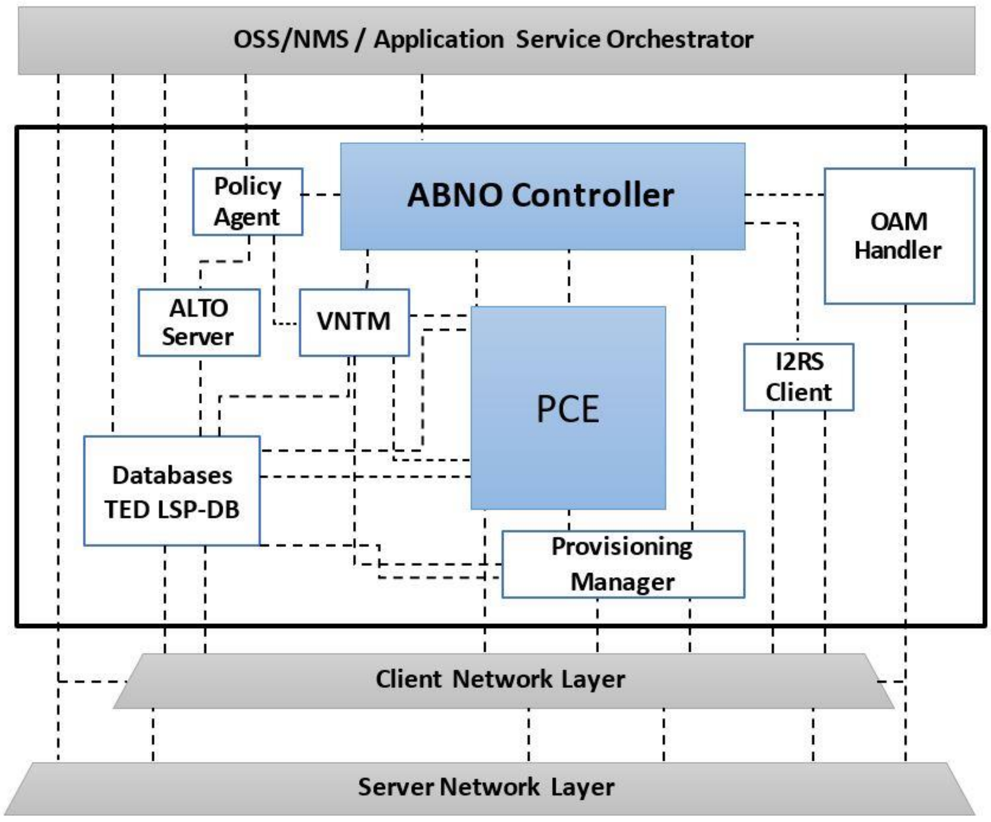


Figure 4 ABNO architecture modules

2.3.1. NMS/OSS/Application Services

NMS (Network Management System) or OSS (Operations Support System) can be used to control, operate and manage a network within the ABNO architecture, to issue high-level service requests to the ABNO controller. In the case of this work, as will be explained later, this component has been emulated by Python scripts that generate such requests. Finally, the NMS and the OSS can use an interface to interact with network elements within the network, to communicate with the SDN controller. In other hand, services in the ABNO architecture may be requested by applications (e.g. a program that runs on a host or server providing services to a user) and are grouped in the Application Service Orchestrator concept.

2.3.2. ABNO Controller

The main gateway to the network for the NMS, OSS and the application services coordinator is the ABNO controller. This communication is for the provision of functions and network coordination. Based on the requirements and restrictions of the application, the ABNO controller governs the behavior of the network by calling the different components in the correct order.

2.3.3. PCE-GMPLS based control

The PCE (Path Computation Element) is a fundamental component defined by the IETF inside of the ABNO architecture. The PCE is a component responsible for the calculation of routes based on the state of the network and can apply certain restrictions in its calculations. It was developed for the calculation of paths in MPLS and GMPLS networks. The ABNO architecture includes a series of databases that contain the necessary information for the system. The two main ones are the Traffic Engineering Database (TED) and the Label Switched Path Database (LSP-DB). The TED stores the information of the resources of the network indicating which are available and which are not and can contain information about metrics, bandwidth capacity of the links, etc.

The TED can be built or updated by means of direct requests through the Network Management System (NMS) or through routing messages read directly from the network. The LSP-DB stores information about the LSPs that are active in the network or that may be established. This database is used only when there is an active/stateful PCE, that is, PCE capable of managing the creation of LSP autonomously.

The PCE follows a modular architecture and allows to add customized algorithms. The PCE has also stateful and remote initiation capabilities. The protocol that allows communication between PCE and Path Computation Client (PCC) is the Path Computation Element Protocol (PCEP). The PCE computes the routes based on its TED that is updated with the information sent by the routing protocols. This update is not instantaneous, which causes that the TED may be out of date with respect to the information of the control plane.

2.3.4. Policy Agent

Policy Agent is a component that is configured by the NMS / OSS with the policies and restrictions. The Policy Agent is responsible for propagating those policies into the other components of the system.

2.3.5. Interface to the Routing System (I2RS) Client

The routing system interface (I2RS) provides a way to access both for read and for write, routing status, and policy information on routers in the network. The objective of the client is to manage requests for information through several routers (each of them running an I2RS Agent) and to coordinate the configuration of the status to / from those routers.

2.3.6. OAM Handler

The OAM Handler (Operations, Administration and Maintenance Handler) basically performs a monitoring function, that is, Monitoring Service, reporting problems in the network and also, detecting faults or incidents and reacting to them. The OAM Handler receives the alerts and acts activating the necessary components to recover the services that had been established by the ABNO Controller.

2.3.7. Databases

Several databases are included in the ABNO architecture that contains information used by the system. The TED (Traffic Engineering Database) and LSP-DB (LSP Database) are the two main databases. The TED contains information about the network, relative the possibility to improve the network with capability data such as bandwidth or status information. The main objective is to provide information to the PCE to compute the route based on it. In the other hand, the LSP-DB refers to the information about the LSPs that have been set up in the network, that is, the routes created, the resources used by the LSPs. The main objective of the LSP-DB within the ABNO architecture is to improve the planning and optimization of LSPs. As is mentioned above, the stateful PCE is the primary consumer of this database.

2.3.8. ALTO Server

The main purpose of the ALTO server is to provide network information to the application layer based on abstract maps of a network giving a view of the locations of the nodes and the costs of the routes between them. An abstract view of a single node of a network consists of two maps: a network map and a cost map. The selection criteria of the Application Services Coordinator to choose between two locations may depend for instance on information such as the width of maximum band, the minimum traffic between domains, the lowest cost, etc. Without going into greater detail about this element of the ABNO architecture, the purpose of the ALTO Server is explained.

2.3.9. VNTM

The VNTM could be defined simply as a set of LSPs in a lower layer network than the ABNO Controller is installed, in such a way as to simplify the management of routes in the upper layer, configuring or deleting LSPs in this lower layer and announcing the changes in the upper layer.

2.3.10. Provisioning Manager

The Provisioning Manager deals with the requests for establishment of the LSPs giving instructions to the different elements of the control plane for the correct configuration of the requests

2.4. Conclusion

As a conclusion, in this section an introduction to the concept of SDN has been shown, detailing the reasons why it is necessary to create this type of networks and the objectives and advantages pursued by this model, as well as a clear structure in which the separation of the control plane and the data plane is clear. The ways of interacting between the different layers have been also presented. Next, due to the need of SDN structures oriented to several domains and several providers, the Netphony modules suite has been presented and also an overview has been given correlating the elements of a general SDN structure with the elements used in this work: The requests generated by Python scripts, the Netphony services, the T-API as NBI and the GMPLS emulated environment as a mixed model of data plane and control plane. Due to the importance of understanding this emulated environment, a section has been dedicated to explain the MPLS and GMPLS protocol and then an explanation of how the communication between the emulated GMPLS nodes is established.

Finally, the ABNO architecture has been introduced, describing a functional architecture through which a controller can be implemented based on the network operations and the benefits that this brings, explaining each of its parts and taking advantage to relate them to the different modules of the Netphony.

3. Transport API

In this section, the API concept is introduced, its main functionality and the reason for its increasing use today. In addition to explaining this concept, is expanded into the types of APIs that exist and then present the REST protocol, in which T-API is based on this work implementing it to Netphony.

3.1. API Concept

The main advantage offered by the design of the SDN architecture, as previously mentioned, goes around the separation of the control plane with the data plane. To understand well this concept, it can be said that the user traffic is the data plane while the traffic destined to managing, maintain and modify the network's state is the control plane. The use of these two traffics is different since the first is important for the stability of the network, and the second to provide the service of the network. In order to manage these network traffic, a communication interface between the physical devices of the network and the controller must be implemented, acting as a southbound interface, and a communication between the controller and the applications or what we could define as a possible user, acting as northbound interface. It is in this aspect where it would be necessary to deepen the concept of Application Programming Interfaces (API) and because its use is so important within the framework of transport networks. The SDN control plane exercises direct control over the state in the network data-plane elements (i.e., routers, switches, and other middle-boxes such as firewalls) via a well-defined API.

An API could be defined as the most useful mechanism to connect two software to each other for the exchange of messages or data in standard format such as XML or JSON. The APIs are designed to make life easier for the programmer, since they do not have to start programming code from scratch, but rather that there are predefined functions that must be implemented according to the context in the one that is working. Each API is designed in a specific programming language and has several specifications that define it. In this project it is an API developed in Java but can be found in many others (e.g. Python, JavaScript, etc.). APIs can be used to communicate with the operating system, with databases or with communication protocols. An end user never sees the API in its work process, however, we profit from the work of the APIs on many occasions in our day, from the moment you use social media icons on a web or now to buy movie tickets for which an API must send credit card information to another program. The APIs are can be subdivided in four main classes as is going to be presented below:

3.1.1. Web services APIs

These are APIs that are used to exchange information between a web service, software that provides access to a concrete service through a URL, and an application. This usually occurs via HTTP or HTTPS requests. The request by the application and the response, as well as the HTTP of the web service, contain information of both the header and the message, normally in the two most common formats: XML and JSON.

Currently there are four types of web services APIs: SOAP (Simple object access protocol) is a standard protocol used to exchange information and data in XML between two objects, currently increasingly disused; XML-RPC, a protocol that calls a remote procedure and uses XML as a data format and HTTP calls as a communication system; JSON-RPC, which is the same protocol but in JSON format (JavaScript Object Notation) which is a light text format for the exchange of data as an alternative to XML; and REST (Representational State Transfer), a software architecture for hypermedia systems on the World Wide Web; a REST API uses the HTTP protocol and this technology is discussed more in depth below.

3.1.2. Library-based APIs

Thanks to the library-based APIs, it is possible that an application can import a library from other software, these libraries are usually designed in JavaScript and a good example of this type of API could be the Google Maps API.

3.1.3. Class-based APIs

This type of APIs enables the connection with data related to classes in object-oriented programming with Java. The Java API uses abstract classes for the creation of applications, providing the necessary to perform functions within those applications, not having to develop code again and being able to use functions already implemented in other Java classes. It is organized in packages and each of these packages contains a set of related classes.

3.1.4. Function APIs in operating systems

The operating systems are continuously interacting with software programs and in many cases in reality, the way in which this interaction takes place is through APIs. The OS (operating systems), such as Linux or Windows, have APIs that enable communication between the programs and the OS. The different Windows APIs can be the user interface, for messaging, for access and data management, for error diagnosis ... etc.

3.2. RESTCONF introduction

Following the description provided by the Internet Engineering Task Force (IETF) [10], RESTCONF is an HTTP (Hypertext Transfer Protocol)-based transfer and configuration management protocol used to access data defined in a YANG file, which is a data modelling language [11]. This means that the aim of the RESTCONF protocol is to map a YANG specification to a RESTful interface. This protocol is not intended to replace NETCONF [12], in fact, it is based on the concepts of processing NETCONF data stores, also using the operations defined by the verbs, which are be presented below.

The web service concept has been mentioned above and they can be defined as the set of applications or technologies with the ability to interoperate on the web. These technologies exchange data between them in order to offer services. REST (Representational State Transfer) is a RESTCONF based style of software architecture linked to the HTTP and URI protocol (Uniform Resource Identifier) identifying a resource by name in a particular namespace, consisting of a set of guidelines to create scalable web services. REST it is not a protocol in itself is an architecture style.

It is necessary to review the HTTP protocol as it is the basis of the REST interfaces. HTTP allows the sending and receiving of documents through a set of rules that determine which messages can be exchanged and how they should be. It must be clear that HTTP understands two different roles, the client and the server, with the client initiating the conversation and the server responding to a request with a response. HTTP messages are composed of headers and a body. HTTP is based on text, messages are bits of text, although the body of the message may contain other formats or just empty, thus being easy to monitoring, meanwhile headers contain metadata with information on how to encode messages or with methods HTTP.

As it is mentioned above, REST uses URI to accessing resources; URLs (Uniform Resource Locator) is a URI to specify access mechanisms or the location of that resource. It is necessary to understand that each URL identifies a resource identifying its location and nothing else, that is, it does not describe any action within the URL since it would not be a RESTful system (based on REST). Optionally, the request may contain additional indications such as saving data in the specified resource. This data can be saved using the "-d" option with CURL , which is an HTTP client functions library that can be used as a command line tool and be used by different languages to connect servers and work with them.

HTTP methods are used to specify the desired action with respect to a resource. Each request specifies a verb or HTTP method in the header. This is the first capitalized word that appears in any header and tells the server what to do with the resource identified in the URL. To build a RESTful API the verbs usually used are GET, POST, PUT and DELETE. GET tells the server to send the data identified by the URL to the client. This data will not be modified by this method by the server since it is only a reading method. With PUT you can edit or create the resource identified by the URL. POST is used many times instead of PUT and vice versa, when the process that is sought to happen can be repeated. Finally, DELETE is used to remove the resource identified by the URL of the request.

All these methods can be classified by secure or insecure methods, depending on whether or not they modify the resources, that is, the only safe method of those listed above would be GET since it does not modify the resource described in the URL. They are also classified as idempotent if the same result is obtained regardless of the repetition of the request. GET, PUT and DELETE are idempotent that is the recurring execution of a request with the same parameters on the same resource will have the same effect on the state of the resource in the system if it is executed 1 or N times, while the only non-idempotent method is POST.

Finally, we have to talk about the representation of the resource both in the request and in the response, understanding by representation as the information about the state of the resource in a specific format. In the header will be included metadata only in text format and the body can contain data of any format (texts, images, XML, HTML, etc.) in the case of this work JSON will be used so the HTTP response should specify the content type of the body in the header with the Content-Type field as will be seen later in the tests. In addition, in the response header the client must be informed about the result of his request. For this the response codes is a standardized way of doing it. Among the different response codes most commonly used are:

- 200 OK. Satisfactory
- 201 Created. A resource has been created.
- 404 Not Found. The URL does not correspond to a resource.
- 405 Method Not Allowed.
- 500 Internal Server Error.

Here ends the introduction to the RESTCONF protocol whose purpose is communication between two systems that must share the understanding of the HTTP protocol. Following this protocol REST presents several advantages such as the separation between the client and the server, independence of the platform type or programming languages where the REST API is located, security and great scalability.

3.3. YANG MODELS

YANG (Yet Another Next Generation) is a data modelling language used in the definition of configurations and interfaces. A configuration can be considered as a set of ordered data, dependent or not on other values. The YANG model interprets the configuration data as a hierarchical tree following a typical XML structure, in which each node has a value or is a point of connection with other nodes, thus being defined together with the interaction with other nodes. In a generic YANG model, configuration data, status data, Remote Procedure Call (RPC) and notifications are included.

The YANG models are directly mapped in XML or JSON format and transmitted by the NETCONF or RESTCONF protocol, allowing configuration management. The YANG language was developed by the IETF NETCONF Data Modelling Language Working Group (NETMOD). The modelling of configurations is a concept that manufacturers of network equipment, both physical and virtual, must take into consideration, both in the long term and in the present. It is one of the elements that network service providers need to have operational today.

From a YANG model it is possible to auto-generate code in several different languages using a set of tools called Swagger. Swagger, reading the API's structure can automatically build API documentation, also automatically generate client libraries for the API in different languages. Swagger does this by asking to the API to return a JSON that contains a detailed description of the entire API. This file is a resource listing of the API which adheres to OpenAPI Specification. The OpenAPI is a specification for machine-readable interface files for describing, producing and visualizing RESTful Web services. Then it is possible to design-first users by using Swagger Codegen [13], which can be download in the official page, to generate a server stub and after implement the server logic. Another useful tool is with Swagger Codegen is to generate client libraries for the API in many languages. In this work will be shown the use of this tool. One f the annotation due by the OpenAPI specification is the JAX-RS [14] (Java API for RESTful Web Service) annotation. JAX-RS is a Java API which support the creation of web services according the REST architecture style. JAX-RS allows some annotations to help the mapping of a specific web resource, some of them are @Path, to indicate the relative path for a resource or method class. @GET, @PUT, @POST and @DELETE to specific the HTTP request type.

3.4. Transport API

Once SDN and the API concept are reviewed, the T-API (Transport API) is introduced in an overall vision as an application programming interface for SDN architectures that are based on data models, providing a vendor agnostic network topology, connectivity services, path computation and virtualization models. That means is independent of the protocols and technologies used and can be mapped to one of the protocols pursued by automated code generation. The T-API was defined by the ONF (Open Network Foundation) [15], allowing for a T-API client retrieve information from a domain of transport network equipment controlled by a T-API server as a transport SDN Controller. It can be said T-API is a NBI (Northbound Interface) for that T-API server and which main objective is to control of the transport network to support growing application demands as bandwidth and latency.

Some of the great benefits of the T-API are a cost reduction of the CAPEX and OPEX, due to an operational simplification of introduction to the new equipment to the network and also a complexity reduction of the management and control of that equipment. The capability to offer new products such as network slicing and virtualization for 5G applications. On the other hand, T-API allows the control of multiple domains with a different technology using a common universal technology framework based on abstracted information models. Thanks to this, the deployment of an SDN across multiple vendors is possible.

The T-API presents several opportunities to bring more control and monitoring of the optical transport network. Some of this applications could be the end-to-end dynamic bandwidth service in a multi-domain scenario supporting the re-optimization of the network architecture. The possibility to offer virtual resources to connect remote sites and supporting the VNFs (Virtual Networks Functions). By last, could be said the T-API allows the support of network slicing, isolating some virtual layers of the networks and giving the capacity to implement 5G services with high bandwidth or ultra-low latency.

To exposes all these information, T-API terms have to be considered and assimilated as a several abstracts representations. T-API is based on a shared *Context* relationship between a server (SDN Controller) and client (SDN Application), where the server exposes its desired the level of abstraction of the network information and the scope of control. The T-API *Context* is defined by a set of *ServiceEndPoints* which enable request of T-API Services by accessing the forwarding capabilities provided by the *Node* (e.g. *ConnectivityService*) which is an abstract representation of the forwarding-capabilities of a particular set of Network Resources, can be connected by *Links* or *TE Links* or by *Connection* paths. Referring to *ConnectivityServices*, a *ServiceEndPoint* conceptually represents a pool of potential *ConnectionEndPoints* to access the forwarding function provided by the *Connection* which represents the enabled capacity to forwarding traffic between two or more *NodeEdgePoints*. By last it is necessary to presents the *Route* and the *Path* as abstract representations of the *Connection* through the *Nodes* in the first case and the ordered list of *Links* or *TE Links* for the second one. The Figure 5, tries to account this structure explained before.

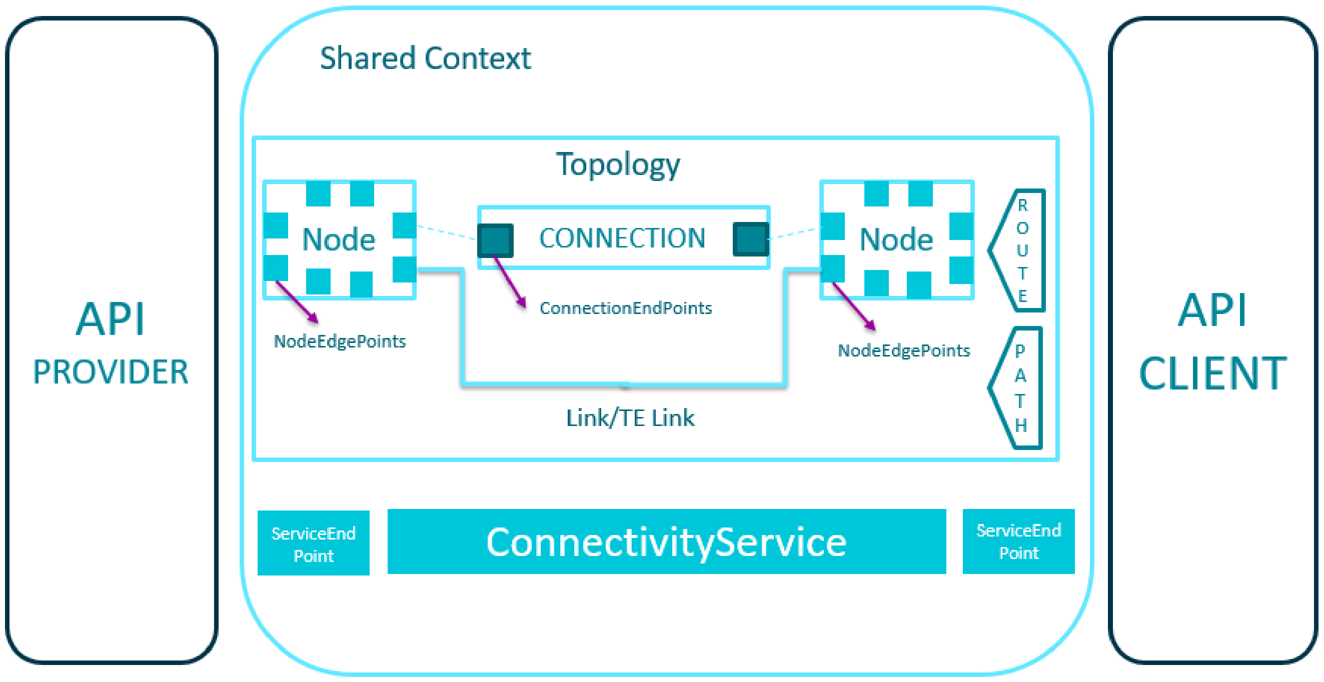


Figure 5 T-API overview elements

Because there is a shared Context which determines the level of the network resource abstraction instances over which the API operates an API client could, for example, request retrieval of the Service End Points in the shared Context or request creation of a Connectivity Service between the Service End Points, also could request creation or the modification of Virtual Network Topology or to subscribe to notification of events. As it shown in the Figure 6, the T-API abstracts five main functional requirements for all these actions are the Topology Service, Connectivity Service, Path Computation Service, Virtual Network Service and Notification Service, this is presented in. Each of this service are composed for a several APIs to provide several functions referring to different actions and will be named hereinafter as sub APIs..

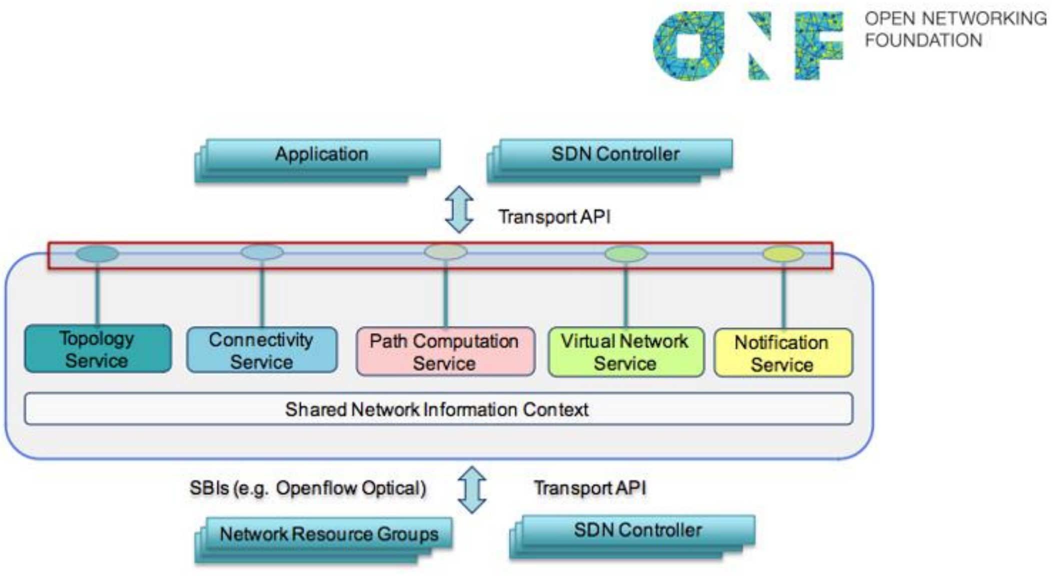


Figure 6 T-API services

3.4.1. Network Topology Service

This service exports the network topology information for an API client that is within its shared Context. Some of the topology retrieval sub APIs are: Get Topology List, which returns a list of the top-level Topology instances including references to the lower-level instances as Nodes and Links. Get Topology Details, which returns attributes of the Topology depending of the inputs providing for example an IDs list or a list of encompassed Nodes indexed by the Name of the Layer-Protocol including Node details. Get Node Details which returns attributes of the *Node* identified by the inputs including the state and the capabilities of such Node. Get *Link* Details in the same way, returns attributes of the Link scope providing by the inputs parameters.

3.4.2. Connectivity Service

Enable the set-up, tear down and modification of connections. The Connectivity Service allow an API client to retrieve connectivity information and request connectivity service within its shared Context. The connectivity sub APIs are divided by connectivity retrieval sub APIs and connectivity request sub APIs.

The connectivity retrieval sub APIs are the next ones: Get Service End Point List which returns a list of *ServiceEndPoints* including the attributes details for each *ServiceEndPoint* also with Get Service End Point Details. Get Connectivity Service List, returns a list of *ConnectivityService* entities including attribute details for each as for example, the reference of the terminate *ServiceEndPoints* with Get Connectivity Service Details. Get Connection Details which returns the attributes of the Connection entity identified by the inputs parameters including also the references to the server and client *NodeEdgePoints* for the *ConnectionEndPoint* with Get Connection End Point Details.

The connectivity request sub APIs are: Create Connectivity Service to create a forwarding entity representing the Connectivity Service request to connect the *ServiceEndPoints* between APIClient and Provider returning a Service ID and a point to point bidirectional service. This entity can be modified by the sub API Update Connectivity Service which returns confirmation or rejection of the modification or Delete Connectivity Service to delete of an existing *ConnectivityService* entity.

3.4.3. Path Computation

End-to-end view for the orchestrator. The sub APIs here have been made taking into account the division of responsibilities between different interfaces of the T-API service. Compute P2P Path which compute a list of paths depending of the inputs parameters, with a several information as path identifier or routing constraints. It is necessary the sub API Optimize P2P Path, because a connection can be reconfigured to include new constraints and achieve this in the path computation via this sub API.

3.4.4. Network Virtualization to a set of Service interfaces

Enable to expose a subset of the network resources to different tenants by different sub APIs as Get Virtual Network Service List to returns a list of Virtual Network Service entities and get the details with the sub API Get Virtual Network Service Details including references to *ServiceEndPoints* of the Virtual Network Service.
In the other hand, the Virtual Network Request sub APIs are: Create Virtual Network Service for the API client to request creation of a virtual network and reserve a set of resources to build up the virtual network. To delete this virtual network the sub API Delete Virtual Network Service can be used.

3.4.5. Notification

Messages that provide information about events. The Notification specifications are used go around the model of an agent and a manager. The agent is the entity which is able to recognizes events and inform about this to the different subscribers. On the other hand, the manager is the entity that governs notification subscriptions and receives notification messages. According to the ONF, Notifications follow publish and subscribe model. A precondition for the subscription is the possibility to know the available notification and their sources. To do this there is a sub API called Discover Supported Notification Types, to allows an API Client to discover the notifications capabilities supported by an API Provider.

3.5. Multi-domain abstraction model

In order to demonstrate the capability of the T-API to orchestrate operations across multiple vendor domains, a multi-vendor multi-domain optical transport network testbed was established at TID (Telefonica I+D) premises and a view of the physical topology, and the abstracted virtual topology in each orchestration domain is presented in Figure below. As shown in Figure 7, two MDO (Multi-Domain Orchestrator) expose the topology of their domains and an orchestration domain was established that manages three different vendor domains, and exposes a simplified topology to the multi-domain network orchestrator. Each domain employs edge-based abstraction, where each domain edge is exposed as an abstracted node and virtual links are offered between the edges of the network. As seen in the case of the child MDO, the multi-domain topology is abstracted to three abstract nodes, indicating the nodes that are connected to other nodes outside the control domain boundaries

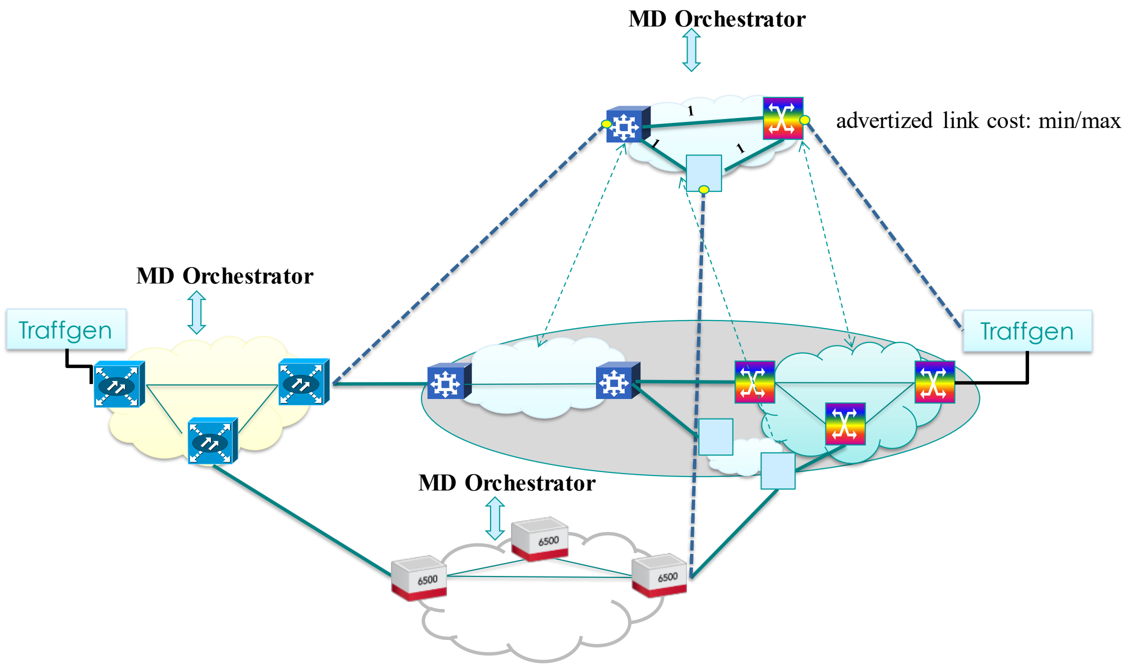


Figure 7 Abstracted Hierarchical Network Topology

In order to discover the topology, the MDOs must not only gather abstract topology information from the domain controllers and the child MDOs, but must also incorporate information about the interconnections between different domains. In order to discover the multi-domain network topology

1. MDOs discover the topology of individual domains including service endpoints and node perimeter points over the T-API
2. MDO topology is populated with connectivity information between domains.
3. MDOs generate an abstract topology using the available information which is then exposed over the T-API

3.6. Conclusion

As conclusion, the API concept has been defined and then an introduction of RESTCONF protocol based on HTTP. Then the definition of YANG models as modelling language and how are directly mapped in XML format and transmitted by the RESTCONF protocol to build a network topology and also how from a YANG model it is possible to auto-generate code in several different programming languages using a set of tools called Swagger. After, the presentation of the Transport API as the main concept of this work and the different T-API functionalities and services. By last, a recovery topology demonstration in a test scenario has shown and the viability for the T-API use in a multi-domain or multi-vendor scenarios.

4. Netphony Environment

In this section, the Netphony suite of modules are presented in detail as a research and development work carried out in Telefonica. It is defined as a set of libraries implemented in Java that allow to create a control plane based on a PCE, with a transport network emulated with GMPLS and an ABNO controller. It is an open source project and is publicly available in Github repositories as previously mentioned. It will present the purpose and the need of the Netphony in the current context about the increase in network traffic that operators are suffering today. Its architecture will be presented and it will be related to the ABNO architecture, component by component. Finally, it will deepen in the own implementation of the Netphony components at a more practical level.

4.1. Netphony: an ABNO implementation

Nowadays, network operators must deal with multilayer architectures in their production networks. Current implementations are based on an IP / MPLS layer, either by themselves using dark fiber marketing the raw connectivity and being the same client who applies appropriate transmission technology to their needs or compatible implementations with an optical switching network. IP / MPLS are the layers that take advantage of statistical multiplexing and maximize the use of optical links created in the underlying layer by switching and routing, and this multilayer architecture requires a layer of network programmability that allows control and administration of IP and optical resources.

The implementation of the Internet through distributed protocols, such as GMPLS, allows the creation of LSP tunnels to create circuits between different network elements. In addition, this provides a resilience capacity in a given network, where thanks to the control layer can make decisions for each node individually and a central intelligence to obtain a network vision and to make a better use of resources.

The proposal of multilayer network programmability by Telefonica I+D, is based on the use of REST-based APIs, since this approach allows rapid development of the network and greater interoperability. A GMPLS control plane to emulate the network elements control, a Path Computation Element with active and stateful capabilities, a Topology Module capable of importing and exporting TE information in different protocols as well as an Application - based Network Operations (ABNO) controller, composes the Open Source Netphony suite. This framework enables multi - layer programmability for IP and optical networks.

There are currently several open source projects that offer a solution for the control of multi-layers, SDN tools which are consistent with different APIs types and implemented in many different programming languages such as C, Python, Java, etc. Particularly some of the most important are OpenDaylight [16], which is an open platform supported by the Linux Foundation is an SDN solution implemented in Java that also supports REST APIs and provides a fast and scalable programming of a multi-layer network and it was worth remarking the great open source platform that this tool has. The other one is Open Networking Operating System (ONOS), also an open source and implemented in Java, with multi-layer SDN control, packet network and optical networks management, with multiple interfaces, types of equipment and protocols.

4.1.1. Netphony architecture

Netphony is a suite of modules developed by Telefónica I+D and forming an SDN tool following the ABNO architecture. Netphony is an open source project, divided into libraries to make it easier and faster the integration and possible expansion to other frameworks, since the purpose of this suite is to be as effective and functional as possible. The reason for following an ABNO architecture is due to being able to allow interoperability between the key components so that separate actions and processes can be performed in the same or different modules. The Figure 8 shows such libraries implemented in the Netphony suite of modules to facilitate the integration and the expansion of it.

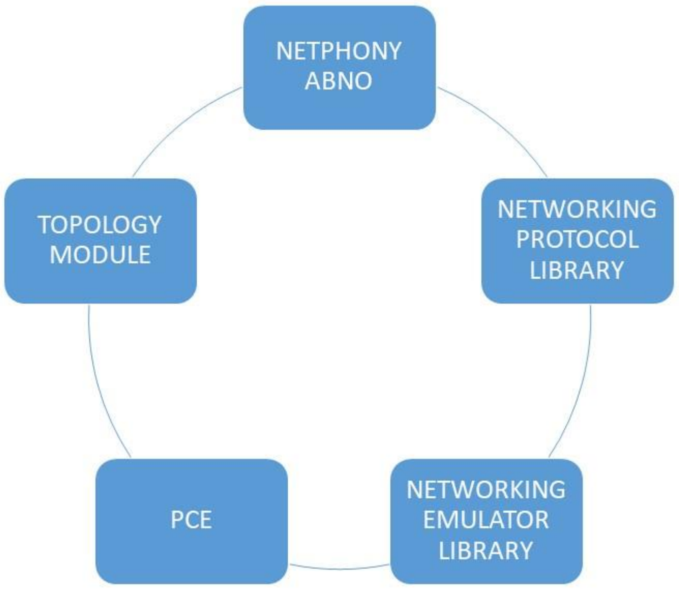


Figure 8 Netphony libraries

1-

In order to carry out all the functions of the network control plane, certain network protocols are necessary. These protocols are contained in Networking Protocol Library, they are PCEP, RSVP, OSPF and BGP-LS. This is a general library that can be integrated by any software that requires the use of these protocols. It is necessary to briefly explain each of these protocols and mention their characteristics and their use.

Open Shortest Path First (OSPF) is a network protocol for hierarchical routing using the Dijkstra algorithm, which has been seen in the degree in programming, second year, to calculate the ideal route between two nodes in a network, taking into account distances of the links and certain restrictions of network. BGP-LS is an extension to Border Gateway Protocol (BGP) for distributing the network's link-state (LS) topology model to external entities, such as the SDN controller. As the OSPF protocol discovers the topology of the network, the changes are also reflected in the BGP-LS model and are also distributed by BGP-LS messages to any interested party, such as SDN controller. The devices are not interested in learning the network topology through BGP-LS, since they already participate in OSPF. PCEP is a request / response protocol that allows communication between multilayer and multi-domain networks. Finally, RSVP (resource reservation protocol), is a protocol of the transport layer designed to reserve resources of a network. RSVP reserves routes in networks for transmission with scalability and robustness resulting in a reserve of resources at each node along a path and was designed to interoperate with current and future routing protocols. Once these protocols are seen, the rest of the libraries are presented.

The library in which the emulated environment of nodes with GMPLS functionality is found is the Networking Emulator Library. This repository is implemented in Java and emulates a transport network that includes emulated nodes with GMPLS control capabilities, RSVP processes and an OSPF daemon that is, running in the background. For the creation of LSPs in each of these nodes, a PCEP interface is also installed. The Traffic Engineering Database (TEDB) which the graphics are stored with all the relative information regarding the nodes, the links and the attributes of network traffic, that is, the bandwidth capacity of the links ... etc. is in the Topology Module. This Netphony topology module, also include a BGP-LS speaker acting as a BGP4 peer to initiate the BGP connections between the peers and as has been explained above, this topology can be implemented using protocols like OSPF.

The next two modules to be presented are the PCE and the Netphony ABNO. Path Computation Element (PCE) carry out for the management of routes in the network graph, calculating routes and optimizing them in terms of optical spectrum consumption. There are many architectures, combining PCEs operating in different domains and / or layers and calculating TE LSPs taking into account restrictions such as available bandwidth, latency, etc. In the ABNO Netphony ,an ABNO controller and a Provisioning Manager are implemented, the first being the main component of the entire ABNO architecture and is the one in charge of orchestrating all the necessary components for a certain operation in the right order. The requests would come from the NMS / OSS (Network Management System / Operation Support System) but in this work this function is given by Python scripts that have been implemented to simulate these requests, as will be presented later. On the other hand the Provisioning Manager configures the network elements to enable the creation of LSP. The protocol used for this, as already explained, is PCEP, but there are other protocols currently such as OpenFlow or Netconf [17].

4.1.2. Relationship with ABNO architecture

Once the different components of the ABNO architecture have been presented in section 2.3, a correlation work can be carried out between the components implemented in the set of Netphony modules and those of the ABNO architecture. In the ABNO architecture defined by the IETF, is referenced to the NMS / OSS / Application Service Orchestrator to control, operate and manage a network by requests to the ABNO Controller. This is done in this work by Python implemented scripts which generate the request to create LSPs between random networks nodes to make the performance evaluation of the code implemented to the connectivity service. A deep explanation about these scripts will be presented later.

As shown in **¡Error! No se encuentra el origen de la referencia.**, the ABNO controller corresponds to the Netphony ABNO controller, in charge of the same functions of an ABNO controller specified by the IETF, exactly as explained above. The Traffic Engineering database (TED) together with the LSP-DB is located in the Netphony Topology Module. The Provisioning Manager is within the ABNO Netphony module but it is not necessary in the presence of a PCE with full status because this has the ability to create the LSP between the nodes and the reservation of resources. Although there is a Policy Agent in the ABNO architecture, there is not any implementation of this in the Netphony. This is due the Policy Agent is a component that is configured by the NMS/OSS with the policies that it applies, and the NMS/OSS in the Netphony as is already mentioned to this work is the Python scripts with no policy to apply.

ABNO ARCHITECTURE	NETPHONY ELEMENTS
OSS/NMS/APPLICATION SERVICE ORCHESTRATOR	PYTHON SCRIPTS
ABNO CONTROLLER	NETPHONY-ABNO
POLICY AGENT	-
DATABASES TED AND LSP-DB	TOPOLOGY MODULE
PROVISIONING MANAGER	NETPHONY-PROVISIONING MANAGER (INSIDE ABNO) (when PCE is stateful the Provisioning Manager is not neccesary)
PCE	NETPHONY-PCE
I2RS CLIENT	-
ALTO SERVER	-
OAM HANDLER	-
VNTM	VNTM BASIC IMPLEMENTED
CLIENT NETWORK LAYER	NETPHONY-GMPLS-EMULATOR
SERVER NETWORK LAYER	

Figure 9 ABNO and Netphony components comparison

Although there is a policy agent in the ABNO architecture, there is no implementation of this in Netphony. This is because the Policy Agent is a component that is configured by the NMS / OSS with the policies that apply, and the NMS / OSS in Netphony, as already mentioned in this final project, are the Python scripts without any policy what to apply The same goes for three other components of the ABNO architecture. The Interface for the routing system (I2RS) Client whose objective is to manage requests for information through a series of routers and Netphony consists of an environment emulated without any router. The HIGH server could be classified as a commercial component with the objective of allowing service providers to share information about network locations and the costs of the routes between them. Netphony is an innovation project without commercial application at this time. Finally, the OAM driver, operations, administration and maintenance plays an important role in an ABNO architecture and the main component to implement the monitoring service that is not implemented in Netphony, which is the reason why OAM Handler is not yet implemented in Netphony.

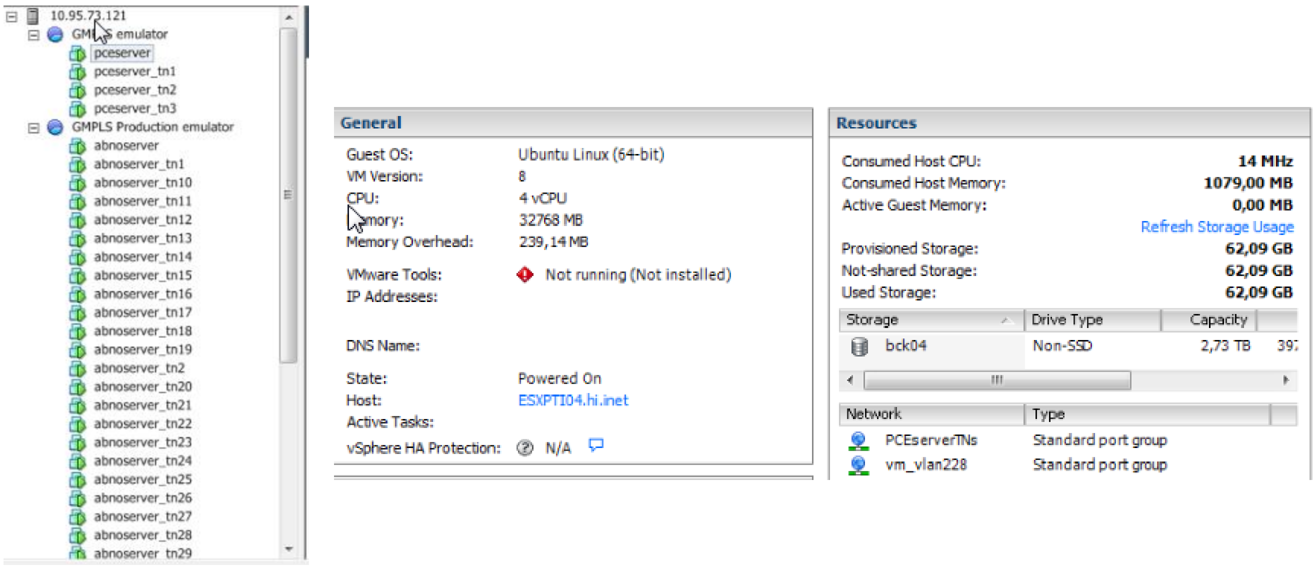
The client network layer and the server network layer could come with the Netphony GMPLS emulated environment nodes and all the interactions between them.

4.2. Netphony components and virtualization environment

Once the architecture of the Netphony has been described and its relation with the ABNO architecture defined in the IETF standard is done, it is possible to detail the Netphony architecture components and to explain certain intrinsic characteristics of each of these.

4.2.1. Virtualization environment

Firstly, it should be explained how the implementation of a virtualized environment is carried out in order to carry out the tests and the operation of the Netphony. At least two virtual machines, VM from now on, are necessary to run a scenario, a pceserver and a TN node, but still to test in different scenarios, up to 30 nodes have been emulated with VMs. These are VM running with the Ubuntu operating system and the pceserver not only acts as a PCE, it also initializes or stops the GMPLS scenarios. These VMs must be installed in the virtualization environment and in addition to being installed on the same host, the pceserver and the "tnX" nodes, they must share the same LAN to have connectivity between them. It is also necessary to implement a VM for the installation of the Netphony ABNO, this VM is abnocontroller, also on the same host. As is shown in Figure 10, the virtualization environment is deployed by VMWare



- 10.95.73.121
  - GMPLS emulator
    - pceserver
    - pceserver_tn1
    - pceserver_tn2
    - pceserver_tn3
  - GMPLS Production emulator
    - abnoserver
    - abnoserver_tn1
    - abnoserver_tn10
    - abnoserver_tn11
    - abnoserver_tn12
    - abnoserver_tn13
    - abnoserver_tn14
    - abnoserver_tn15
    - abnoserver_tn16
    - abnoserver_tn17
    - abnoserver_tn18
    - abnoserver_tn19
    - abnoserver_tn2
    - abnoserver_tn20
    - abnoserver_tn21
    - abnoserver_tn22
    - abnoserver_tn23
    - abnoserver_tn24
    - abnoserver_tn25
    - abnoserver_tn26
    - abnoserver_tn27
    - abnoserver_tn28
    - abnoserver_tn29

General

Guest OS:	Ubuntu Linux (64-bit)
VM Version:	8
CPU:	4 vCPU
Memory:	32768 MB
Memory Overhead:	239,14 MB
VMware Tools:	Not running (Not installed)
IP Addresses:	
DNS Name:	
State:	Powered On
Host:	ESXPTI04.hi.inet
Active Tasks:	
vSphere HA Protection:	N/A

Resources

Consumed Host CPU:	14 MHz
Consumed Host Memory:	1079,00 MB
Active Guest Memory:	0,00 MB
	Refresh Storage Usage
Provisioned Storage:	62,09 GB
Not-shared Storage:	62,09 GB
Used Storage:	62,09 GB

Storage	Drive Type	Capacity
bck04	Non-SSD	2,73 TB 39

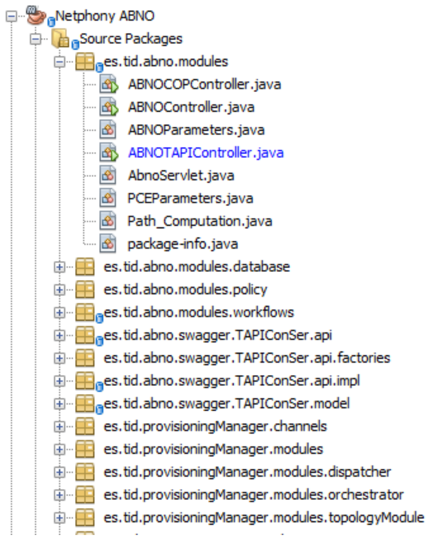
Network	Type
PCEserverTNs	Standard port group
vm_vlan228	Standard port group

Figure 10 VMWare emulated environment
2-

The IP address of these virtual machines will be in the range from 192.168.1.1 for tn1 to 192.168.1.X for "tnX". This configuration must be done in /etc/networking/interfaces in each virtual machine. To modify or add files in any of these virtual machines some programs can be used seen in the degree in different subjects, such as FileZilla or WinSCP to connect through the SFTP protocol.

4.2.2. Netphony ABNO

Application-Based Network Operations (ABNO) architecture proposed by IETF to allow the automation and programmability of networks is followed in the Netphony project and is in progress to implement T-API as part of the purpose of this work, defined using YANG models and following REST architecture software as is shown in Figure 11. It is a set of control plane functions that allow interoperability between control plane paradigms such as OpenFlow, GMPLS and PCEP, etc.



```
Netphony ABNO
  Source Packages
    es.tid.abno.modules
      ABNOCOPController.java
      ABNOController.java
      ABNOParameters.java
      ABNOTAPIController.java
      AbnoServlet.java
      PCEParameters.java
      Path_Computation.java
      package-info.java
    es.tid.abno.modules.database
    es.tid.abno.modules.policy
    es.tid.abno.modules.workflows
    es.tid.abno.swagger.TAPIConSer.api
    es.tid.abno.swagger.TAPIConSer.api.factories
    es.tid.abno.swagger.TAPIConSer.api.impl
    es.tid.abno.swagger.TAPIConSer.model
    es.tid.provisioningManager.channels
    es.tid.provisioningManager.modules
    es.tid.provisioningManager.modules.dispatcher
    es.tid.provisioningManager.modules.orchestrator
    es.tid.provisioningManager.modules.topologyModule
```

Figure 11 Netphony ABNO packages
3-

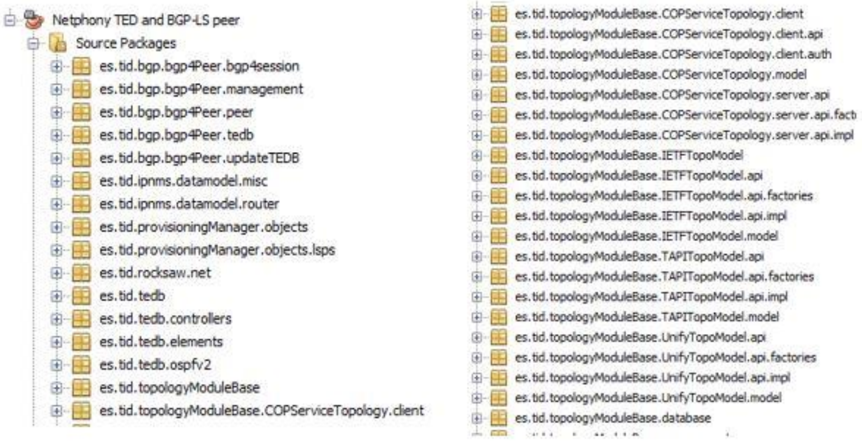
The ABNO Controller implementation needs a PCE session and an .xml file with ABNO parameters is required. Some of these parameters are some PCE parameters, the workflow name, the PCE IP address, the layer PCE capabilities true if send Initiate message to PCE or false if send Initiate message to Provisioning Manager. An example of .xml is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
        <TMAddress>localhost</TMAddress>
        <TMPort>1239</TMPort>
        <ABNOPort>4445</ABNOPort>
        <PCEPPortPM>4446</PCEPPortPM>
        <PMAddress>localhost</PMAddress>
        <ABNOMode>4</ABNOMode>
        <VNTMPort>4190</VNTMPort>
        <VNTMParameters>localhost</VNTMParameters>
        <VNTMAddress>localhost</VNTMAddress>
        <PCEParameterslist>
                <PCEParameters>
                    <PCEMode>2</PCEMode>
                    <PCEPort>4189</PCEPort>
                    <PCEAddress>192.168.1.200</PCEAddress>
                    <Policy>
```

```
                        <WFName>L0ProvisioningWF</WFName>
                        <L0PCECapabilities>

<Instantiation>false</Instantiation>
                        </L0PCECapabilities>
                    </Policy>
                </PCEParameters>
        </PCEParameterslist>
</config>
```

4.2.3. Netphony Topology

As previously mentioned, the Netphony Topology is composed of a BGP-LS speaker, a Traffic Engineering Database (TED), which is a collection of nodes and links, each with traffic engineering attributes and a topology module with plugins enabled to export and import into the database. These plugins are BGP-LS or RESCONF based APIs to import or export the topologies to external elements. The Topology Module exports via RESTCONF with COP model, IETF model, and with UNIFY model. In other hand, the Topology Module import via XML which can learn a topology described in an XML file or via BGP-LS which can run in three modes to only export, only import or in an import/export mode. In the following figure is shown the packages implemented in the project Netphony TED and BGP-LS peer that would constitute the topology module and where it can be seen in Figure 12 the named plugins, in addition to the most recently implemented which is the topology model for the T -API.



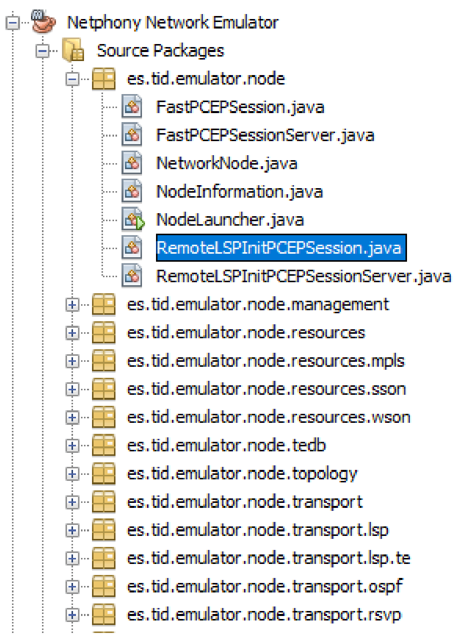
```
Netphony TED and BGP-LS peer
  Source Packages
    es.tid.bgp.bgp4Peer.bgp4session
    es.tid.bgp.bgp4Peer.management
    es.tid.bgp.bgp4Peer.peer
    es.tid.bgp.bgp4Peer.tedb
    es.tid.bgp.bgp4Peer.updateTEDB
    es.tid.ipnms.datamodel.misc
    es.tid.ipnms.datamodel.router
    es.tid.provisioningManager.objects
    es.tid.provisioningManager.objects.lsps
    es.tid.rocksaw.net
    es.tid.tedb
    es.tid.tedb.controllers
    es.tid.tedb.elements
    es.tid.tedb.ospfv2
    es.tid.topologyModuleBase
    es.tid.topologyModuleBase.COPServiceTopology.client

    es.tid.topologyModuleBase.COPServiceTopology.client
    es.tid.topologyModuleBase.COPServiceTopology.client.api
    es.tid.topologyModuleBase.COPServiceTopology.client.auth
    es.tid.topologyModuleBase.COPServiceTopology.model
    es.tid.topologyModuleBase.COPServiceTopology.server.api
    es.tid.topologyModuleBase.COPServiceTopology.server.api.fact
    es.tid.topologyModuleBase.COPServiceTopology.server.api.impl
    es.tid.topologyModuleBase.IETFTopoModel
    es.tid.topologyModuleBase.IETFTopoModel.api
    es.tid.topologyModuleBase.IETFTopoModel.api.factories
    es.tid.topologyModuleBase.IETFTopoModel.api.impl
    es.tid.topologyModuleBase.IETFTopoModel.model
    es.tid.topologyModuleBase.TAPITopoModel.api
    es.tid.topologyModuleBase.TAPITopoModel.api.factories
    es.tid.topologyModuleBase.TAPITopoModel.api.impl
    es.tid.topologyModuleBase.TAPITopoModel.model
    es.tid.topologyModuleBase.UnifyTopoModel.api
    es.tid.topologyModuleBase.UnifyTopoModel.api.factories
    es.tid.topologyModuleBase.UnifyTopoModel.api.impl
    es.tid.topologyModuleBase.UnifyTopoModel.model
    es.tid.topologyModuleBase.database
```

Figure 12 Netphony Topology packages

4.2.4. Netphony GMPLS emulator

The netphony GMPLS Emulator is a Java based emulator of a Transport Network, that runs a GMPLS based control plane with a OSPF-TE daemon that sends/receives OSPF-TE packets and a RSVP-TE daemon to set-up LSPs. There is PCEP connection with a PCE which can be used in two modes, stateless mode or stateful mode, to path queries or to remote initiation of LSPs.

To run a GMPLS node emulator, NodeLauncher is the main class inside es.tid.emulator.node package as Figure 13 presents. This class creates a NetworkNode object and establishes it with the correct args and then starts the node using the startNode method, which is found in the NetworkNode class. It represents a transport node with GMPLS capabilities, an optical transport network (OTN) node or a reconfigurable optical add / extract multiplexer (ROADM) [18]. Launches the necessary connections OSPF-TE, RSVP-TE and PCEP. The parameters of the node can be configured in an .xml file..



```
Netphony Network Emulator
  Source Packages
    es.tid.emulator.node
      FastPCEPSession.java
      FastPCEPSessionServer.java
      NetworkNode.java
      NodeInformation.java
      NodeLauncher.java
      RemoteLSPInitPCEPSession.java
      RemoteLSPInitPCEPSessionServer.java
    es.tid.emulator.node.management
    es.tid.emulator.node.resources
    es.tid.emulator.node.resources.mpls
    es.tid.emulator.node.resources.sson
    es.tid.emulator.node.resources.wson
    es.tid.emulator.node.tedb
    es.tid.emulator.node.topology
    es.tid.emulator.node.transport
    es.tid.emulator.node.transport.lsp
    es.tid.emulator.node.transport.lsp.te
    es.tid.emulator.node.transport.ospf
    es.tid.emulator.node.transport.rsvp
```

Figure 13 Netphony GMPLS emulator packages

4-

4.3. Conclusion

To sum up, an introduction to the suite of Netphony modules has been made in section 4.1 and the purpose of its creation. Secondly, a general vision of the architecture of the Netphony presenting each one of the libraries that compose it and the main function of each of them, briefly explaining also the network protocols used and their characteristics in section 4.1.1. Next, a correlation work has been carried out between the elements of an ABNO architecture and the elements implemented in the Netphony following this architecture in section 4.1.2. Finally, in section 4.2 the Netphony's own components and in the virtualization environment, explained in section 4.2.1, the Netphony ABNO in section 4.2.2, Netphony Topology in section 4.2.3, have been explored. and finally Netphony GMPLS Emulator in section 4.2.4.

5. Tests definition

Once the concept of SDN has been presented and its role in the current context of the evolution of transport networks and the increase in traffic they will suffer, the ABNO architecture has been presented in order to implement SDN. The T-API has been presented as an API for a transport network to manage traffic flow through services and sub API. Finally, Netphony was presented as a set of modules developed by TID (Telefónica I+D) based on an ABNO architecture and in which it is intended to implement and integrate T-API as an add-on to Netphony, which makes it more scalable and dynamic as traffic manager in transport networks converting it into SDN. In this section, the definition of the tests will be presented, that is, everything necessary to understand in which environment the tests that will be subsequently presented in the subsequent sections have been carried out. The experimental scenario will be exposed with 30 emulated nodes, presenting Telefónica Spain optical network in the emulated control plane. In other hand, the workflow of the ABNO architecture for the connectivity service and for the topology service will be shown. The implementation of the connectivity service and topology service of the T-API will also be presented as well the auto-generation of the code with swagger codegen and the integration on Netphony. Finally, the functional validation using a sniffer as Wireshark, to capture the traffic and interpret the results of these traces of the protocols messages, that is, it will show how this traffic capture is done so that in the later section it can be interpreted.

5.1. Experimental Scenario

In this project it has been implemented the ABNO architecture to support the connectivity and topology services workflow presented in the Netphony module suite. The intention is to demonstrate the capabilities of the architecture to support the Transport API in a GMPLS network. To do this, requires to run the Netphony Modules on a server named "abnoserver" and the GMPLS in the Telefonica I+D control plane with virtual machines that are accessible from the abnoserver. The control plane testbed is composed of GMPLS nodes with software developed internally. The experimental setup is built with emulated nodes, which run in an Ubuntu server Linux distribution. Each emulated node implements a GMPLS stack (including RSVP, OSPFv2, and PCEP) and a flexible mode emulator. Each GMPLS controller is a virtual machine, all are running as a server with two Intel Xeon E5-2630 2.30 GHz processors, six cores each, and 192 GB RAM. This use VMWare 5.0 to deploy the VMs. The installation of the VMs is public at the Netphony project wiki. The GMPLS control plane is implemented in Java 1.6. Figure 14 presents the optical mesh of Telefonica of Spain in the emulated control plane.

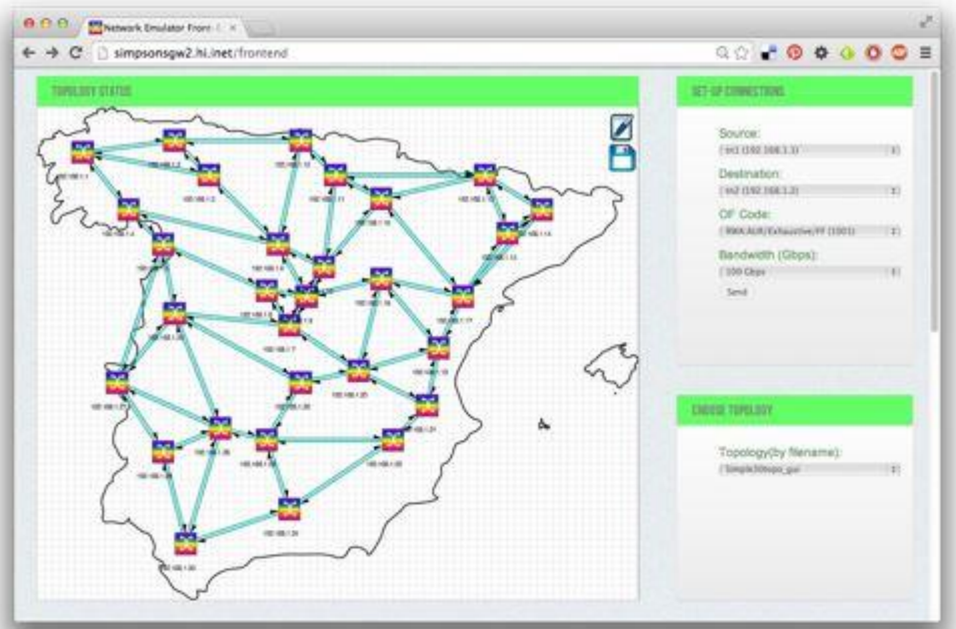


Figure 14 Telefonica de España emulated environment

5.2. Auto-generated Swagger code

Previously in section 3.3, an introduction was made to the YANG models used in the configuration and definition of the APIs. It was also explained that from a YANG model file it is possible to generate code in different programming languages and it is precisely what we want the T-API to be implemented in Java. Therefore, to integrate both services, connectivity and topology, in Netphony, the first step is to generate the code with Swagger based on the YANG models that are in the public Snowmass repository.

This Java code will consist of methods and classes without implementing its operation, which must be implemented according to the actions required by each method. In this way, we have a structure according to the Yang model in Java. In the generation process of the T-API a plugin called Pyang comes into play. Pyang is an extensible YANG validator written in python. It can be used to transform the YANG modules into other formats and generate code from the modules and thus obtain the RESTCONF API of a yang model. The RESTCONF API of the YANG model is interpreted with Swagger, for the description of the API. Finally, the Swagger Codegen generator, which is an open source project, simplifies the compilation process, and then focuses on the implementation and adoption of the T-API in the Netphony.

5.2.1. Connectivity Service

To auto generate code with swagger is necessary to get the tapi-connectivity folder where will be the YANG model. This is in the Snowmass repository [19] and must be cloned in our virtual machine where must be also swagger installed and launch the next command:

```
java -jar swagger-codegen-cli-2.2.1.jar generate -i /home/rodrigo/swagger/tapi-connectivity.swagger -l jaxrs -o tapi-connectivity/ --api-package es.tid.abno.swagger.TAPIConSer.api --model-package es.tid.abno.swagger.TAPIConSer.model
```

To get the swagger generator we can run in the Linux Shell the next command before the last one:

```
wget https://oss.sonatype.org/content/repositories/releases/io/swagger/swagger-codegen-cli/2.2.1/swagger-codegen-cli-2.2.1.jar
```

5.2.2. Topology Service

To auto generate code with swagger is necessary to get the tapi-topology folder where will be the YANG model. This is in the Snowmass repository and must be cloned in our virtual machine where must be also swagger installed and launch the next command:

```
java      -jar      swagger-codegen-cli-2.2.1.jar      generate      -i
/home/javier/swagger/TapiTopology.swagger -l jaxrs -o tapi-network/ --api-
package     es.tid.topologyModuleBase.TAPITopoModel.api     --model-package
es.tid.topologyModuleBase.TAPITopoModel.model
```

5.3. Integration on Netphony

5.3.1. Connectivity Service

Once the generated code is already done, to integrate this code into the Netphony project, the appropriate packages have been created in the project: tid.abno.swagger.TAPIConSer.api (also .impl and .factories and .model would appear). On the other hand, the dependencies and properties of pom.xml must be verified. POM (Project Object Model) is an XML file, which is the main file of a Maven project. It contains information about the project, such as sources, dependencies, complements, the version ... etc. In the pom file, the main changes have been variable, such as <java.version> <swagger-core-version> <wharf - version> <jersey2-version> ... etc. Also, what is really important is to add the profile that will be compiled by maven when executing the mvn -package -P command, this profile must be for the T-API since it was previously added to the COP: <profile> <id> generate- jar-complete-ABNO-TAPI </ id> [..] </profile> and put in the tag <mainClass> ABNOTAPIController </mainClass> in tid.abno.modules. The ABNOTAPIController.java class has been created within the.tid.abno.modules, changing it in the following way with respect to the class that was already created for the COP::

```
     jerseyServlet.setInitParameter(
                "com.sun.jersey.config.property.packages",

"io.swagger.jaxrs.json;io.swagger.jaxrs.listing;es.tid.abno.swagger.TAPIConSer.api");
```

Finally, by console, whenever the project had been modified, must be compiled with the following line, where generate-full-jar-ABNO-TAPI is the profile previously implemented. Then a .jar file in the project folder called abno-1.3.3-shaded has been generated which is the one that will have to be uploaded to PCEServer, for example by winscp and rename it with the name ABNO.jar and put in the directory tid@abnoserver:/usr/local/abno/lib$ and so it is ready to launch the abno with the script ./restart_abno.sh

```
mvn package -P generate-full-jar-ABNO-TAPI
```

5.3.2. Topology Service

The exporter plugin is needed in order to export the topology with the TAPI. Inside of "es.tid.topologyModuleBase.plugins.writer" package, a new class called "TopologyServerTAPI.java" is created with one minor change respect the template file "TopologyServerCOP.java": Instead of getting the COP export port in the parameters, is changed to the TAPI export port.

Finally is necessary to compile the code with Maven with the mvn package -P instruction. If the build is successful, a message is shown indicating: "Build success".

5.4. Use definition and workflow

Through a workflow diagram, the visualization of the communication chain between the different components of an architecture in the correct order is shown. In this case, the aim is to explain the order of communication between the different parts of the ABNO architecture within the Netphony for each of the implemented services of the T-API. These are the connectivity service and the topology service.

5.4.1. Connectivity Service

The connectivity services do the operations for the set-up, tear down and modification of connections. These connections could be as basic as a point-to-point connection between two locations. Nonetheless, there are scenarios where specific constraints are required on the connection. Some examples include the node/link inclusion/exclusion from a connection path doing that by the ERO constraint, constraints on Traffic-Engineering (TE) parameters, like delay or bandwidth. The Figure 15 shows the following steps which are required to provision bandwidth in the ABNO architecture for Connectivity Service Provisioning use case with the Transport API:

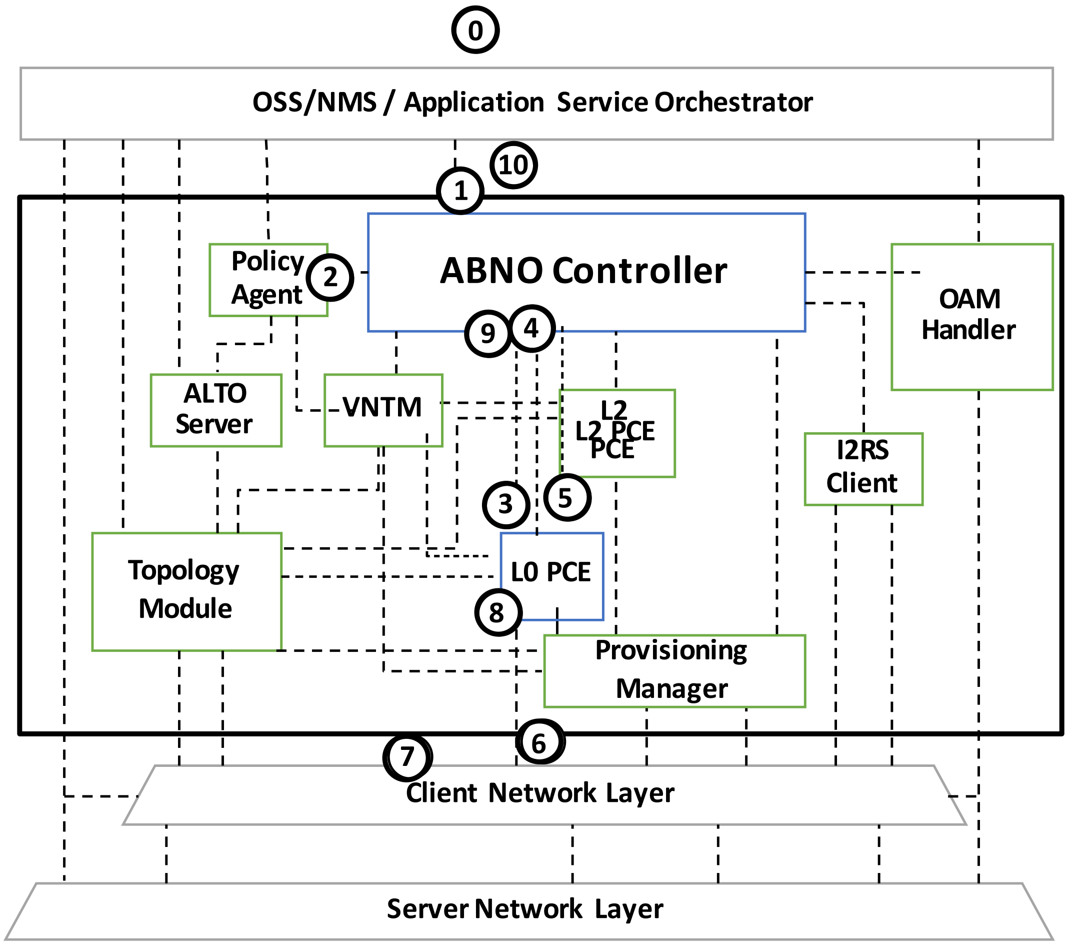


Figure 15 Netphony ABNO workflow

0. The initial trigger for this use case is a request from an end user via an NMS or an application.
1. The NMS sends the information to the ABNO controller with the traffic parameters (e.g. bandwidth). In this experiment, we use the Transport API to request the connection with the Connectivity Service.
2. The ABNO controller asks the policy agent to check whether it has sufficient rights to make the request.
3. The ABNO Controller sends a PCEP Request message to the L0 PCE for a path.
4. The L0 PCE response to the ABNO with PCEP Reply with the computed path. The PCE uses the OSPF information to build the TEDB.
5. The ABNO sends again a PCEP Initiate to start the configuration.

6. The PCE sends the PCEP Initiate to the node (with the information of the ends points).
7. The LSP is configured between the nodes with RSVP message.
8. The node notifies the L0 PCE with a PECP Path Computation LSP State Report.
9. Once the path has been established, the L0 PCE notifies to the ABNO controller with a PCEP Report.
10. Similarly, the ABNO controller advertises the OSS/NMS/Application, with the Transport API Connectivity Service.

Figure 16 shows how a bandwidth reservation is made for an LSP but not from the communication between the components of the architecture based on ABNO, but through the implemented classes and following the workflow model of the T-API protocol, which calls the T-API classes, such as ABNOTAPIController. As the diagram in the figure shows, the route request is managed by the ConfigApi class to redirect the action depending on the verb (POST / GET) and depending on the route and from there it is sent to ConfigApiServiceImpl with the "uuid" and "context" "of the topology. The getPath_Computationlist method must be invoked in the ABNOTAPIController class and, as we have seen previously in the workflow of the ABNO-based architecture, it is necessary to ask the PCE to calculate the route, so the getPCEOpticalLayer method is invoked in ABNOParameters and from there it is sent to PCEParameters with the address where the PCE is located and the port in the variables "pceAddress" and "pcePort". Once the LSP is calculated, the information is returned..

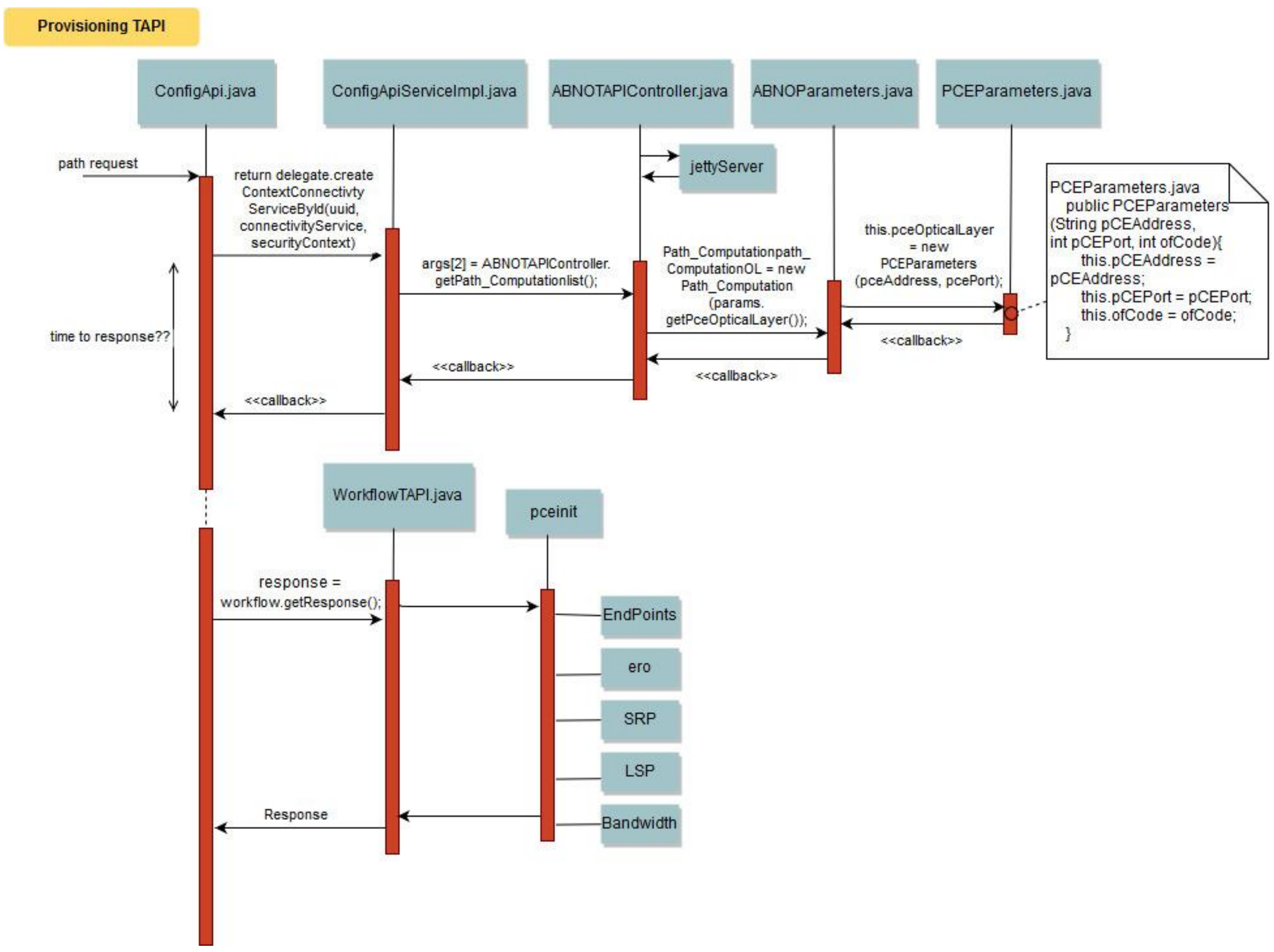

Provisioning TAPI
ConfigApi.java
ConfigApiServiceImpl.java
ABNOTAPIController.java
ABNOParameters.java
PCEParameters.java
path request
return delegate.create ContextConnectivty ServiceById(uuid, connectivityService, securityContext)
jettyServer
args[2] = ABNOTAPIController. getPath_Computationlist();
Path_Computationpath_ ComputationOL = new Path_Computation (params. getPceOpticalLayer());
this.pceOpticalLayer = new PCEParameters (pceAddress, pcePort);
PCEParameters.java public PCEParameters (String pCEAddress, int pCEPort, int ofCode){ this.pCEAddress = pCEAddress; this.pCEPort = pCEPort; this.ofCode = ofCode; }
time to response??
<<callback>>
<<callback>>
<<callback>>
<<callback>>
WorkflowTAPI.java
pceinit
response = workflow.getResponse();
EndPoints
ero
SRP
LSP
Bandwidth
Response

Figure 16 LSP reservation workflow

Afterwards, the pceinit message must be created, which contains the information of the two nodes between the reservation is created, that is, the “EndPoints”, “ERO” (Explicit Route Object), “SRP” (Stateful PCE

Request Parameters), “LSP” the “Bandwidth” and the “id”, so the next step in the ConfigApiServiceImpl.java is to call up the WorkflowTAPI.java and create the object workflow passing by parameters for the constructor the “cArg” and “args”.

```
WorkflowTAPI workflow =
(WorkflowTAPI)act.getDeclaredConstructor(cArg).newInstance(args);
```

Then the response as workflow.getResponse can be saved, but to understand how works the WorkflowTAPI and how the pceInit is created, the diagram bellow explain the several methods in the WorkflowTAPIclass and the function of all of them. As is presented in Figure 17, this follow a scalar implementation because some methods do something that was done just in another one before but adding some new functionality, depending the objective when WorkflowTAPIis called up. The main actions presented here is the call to the provisioning Manager, the creation of the pceInit with their parameters: ERO, SRP, LSP, Bandwidth, Endpoints and Id, send the message by the socket, open a new PCEP session on the port.

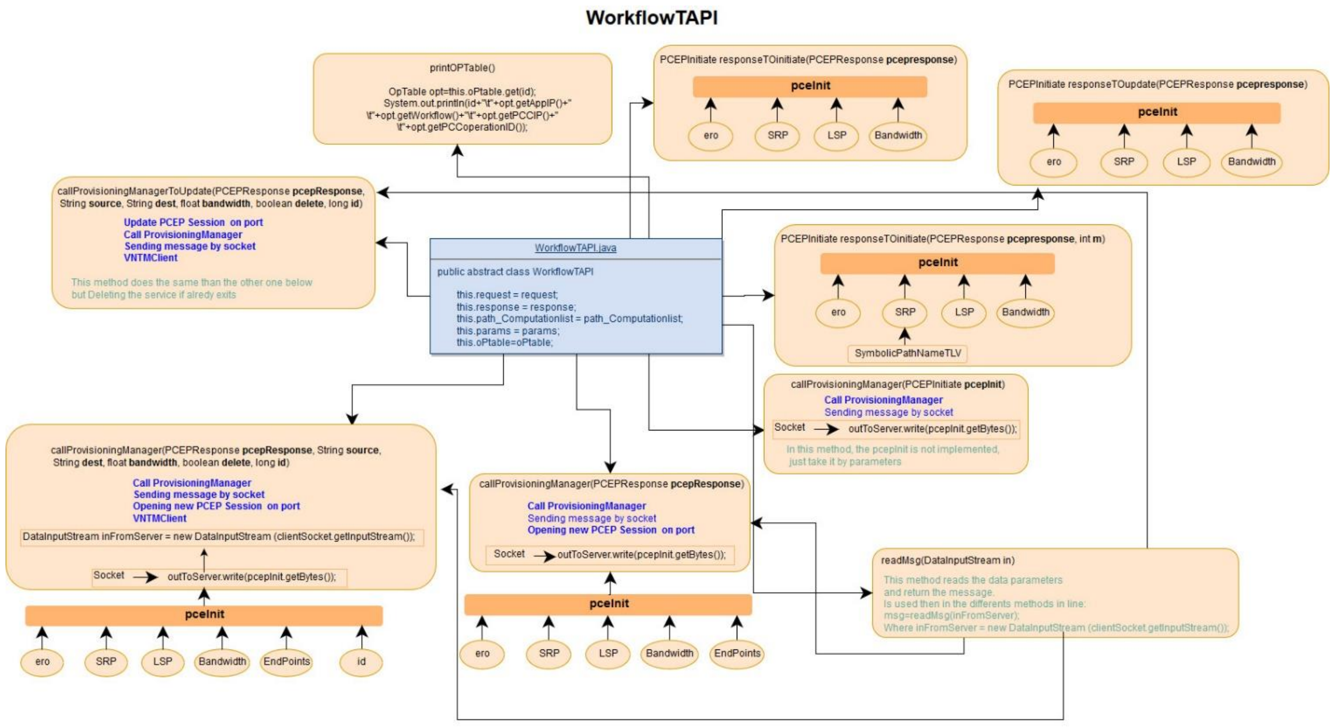

WorkflowTAPI
printOPTable()
OpTable opt=this.oPtable.get(id);
System.out.println(id+"\t"+opt.getAppIP()+"
\t"+opt.getWorkflow()+"\t"+opt.getPCCIP()+"
\t"+opt.getPCCoperationID());
PCEPInitiate responseTOinitiate(PCEPResponse pcepresponse)
pceInit
ero
SRP
LSP
Bandwidth
PCEPInitiate responseTOupdate(PCEPResponse pcepresponse)
pceInit
ero
SRP
LSP
Bandwidth
callProvisioningManagerToUpdate(PCEPResponse pcepResponse, String source, String dest, float bandwidth, boolean delete, long id)
Update PCEP Session on port
Call ProvisioningManager
Sending message by socket
VNTMClient
This method does the same than the other one below but Deleting the service if alredy exits
WorkflowTAPI.java
public abstract class WorkflowTAPI
this.request = request;
this.response = response;
this.path_Computationlist = path_Computationlist;
this.params = params;
this.oPtable=oPtable;
PCEPInitiate responseTOinitiate(PCEPResponse pcepresponse, int m)
pceInit
ero
SRP
LSP
Bandwidth
SymbolicPathNameTLV
callProvisioningManager(PCEPInitiate pcepInit)
Call ProvisioningManager
Sending message by socket
Socket
outToServer.write(pcepInit.getBytes());
In this method, the pcepInit is not implemented, just take it by parameters
callProvisioningManager(PCEPResponse pcepResponse, String source, String dest, float bandwidth, boolean delete, long id)
Call ProvisioningManager
Sending message by socket
Opening new PCEP Session on port
VNTMClient
DataInputStream inFromServer = new DataInputStream (clientSocket.getInputStream());
Socket
outToServer.write(pcepInit.getBytes());
pceInit
ero
SRP
LSP
Bandwidth
EndPoints
id
callProvisioningManager(PCEPResponse pcepResponse)
Call ProvisioningManager
Sending message by socket
Opening new PCEP Session on port
Socket
outToServer.write(pcepInit.getBytes());
pceInit
ero
SRP
LSP
Bandwidth
EndPoints
readMsg(DataInputStream in)
This method reads the data parameters and return the message.
Is used then in the differents methods in line:
msg=readMsg(inFromServer);
Where inFromServer = new DataInputStream (clientSocket.getInputStream());

Figure 17 WorkflowTAPI classes

5.4.2. Topology Service

The topology service exports the topological information with unique identifiers. Such network identifiers which are IPv4 addresses or data-path-IDs, are required for identifying components in while using other TAPI functionalities. Network topology service exports nodes and the links in a given domain in a share context, which can be physical or virtual. In order to get the information from the TEDB, an implementation of the code must be done in cne-BGPLSpeer es.tid.topologyModuleBase.TAPITopoModel.api.impl ConfigApiServiceImpl.java and the retrieveContextTopologyTopology is called up from the ConfigApi.java class following the next path.

```
    public Response
retrieveContextTopologyTopology(SecurityContext
securityContext) throws NotFoundException {
        TopologiesDataBase ted =
TopologyServerTAPI.getActualTed();
     TopologyContext tc= new TopologyContext();
```

```
        for(Map.Entry<String, TEDB>entry :
ted.getTeds().entrySet() ){
                System.out.println("Topology with id:
"+entry.getKey());
                 //check if an object is an instance of a
specific class
                 if (entry.getValue() instanceof DomainTEDB) {

     tc.addTopologyItem(TranslateModel.translateTopology(entry.g
etKey(), (DomainTEDB)entry.getValue()));
                 }
        }
         return Response.ok().entity(tc.getTopology()).build();
    }
```

```
@POST
@Path("/operations/tapi-topology%3Aget-topology-list/")
```

In the same way that has been done with the provisioning of bandwidth with the connectivity service, the Figure 18 shows the iteration between classes for the Netphony topology service. The request is managed by the ConfigApiServiceImpl class within the TAPITopoModel package, which creates an TopologiesDataBase object type called "ted" by calling the getActualTed method in the TopologyServerTAPI class. Next, a TopologyContext object called "tc" is created by calling the TopolgyContext class. Finally, the items in the topology are added to "tc" using the TranslateModel class. This class is in charge of initializing the elements of the topology, links and nodes, taking them from a .xml configuration file.

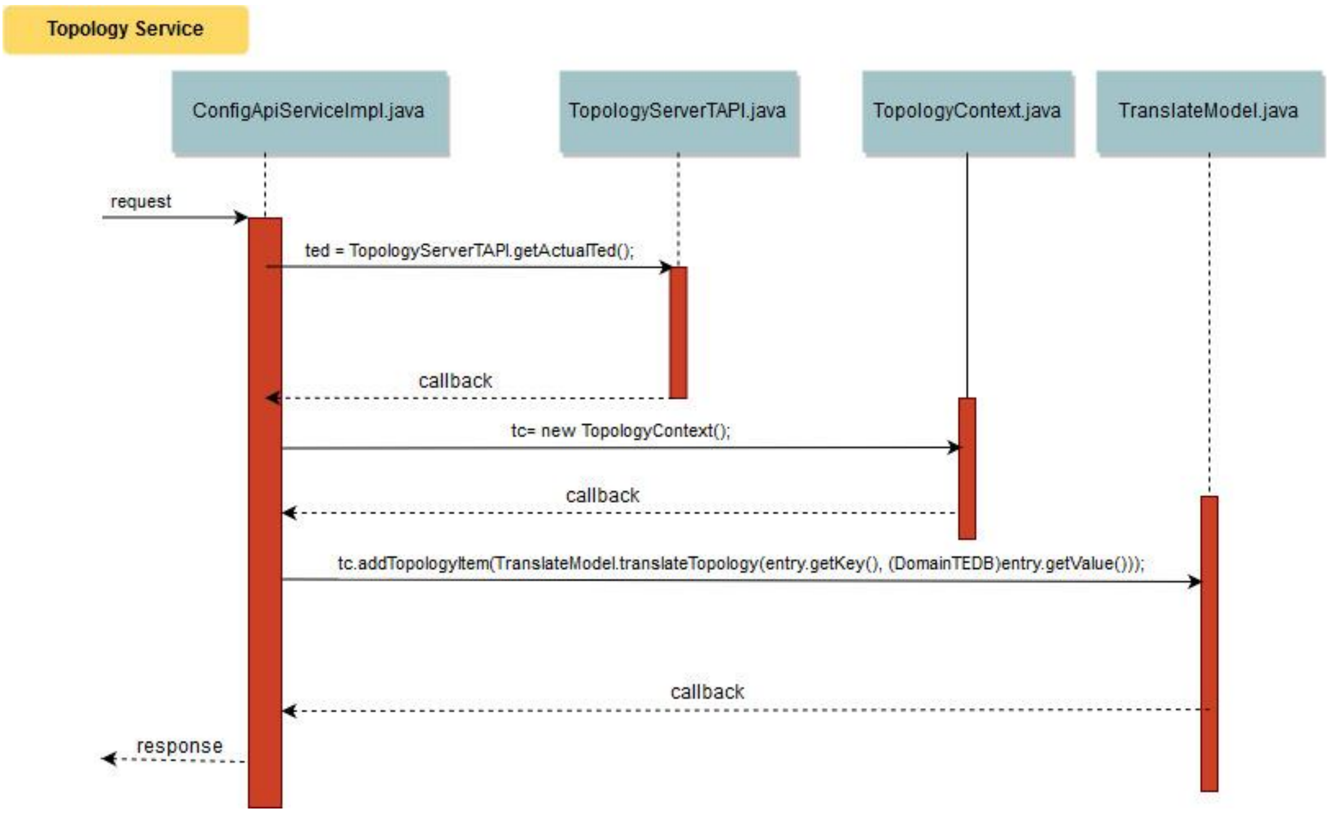

Topology Service
ConfigApiServiceImpl.java
TopologyServerTAPI.java
TopologyContext.java
TranslateModel.java
request
ted = TopologyServerTAPI.getActualTed();
callback
tc= new TopologyContext();
callback
tc.addTopologyItem(TranslateModel.translateTopology(entry.getKey(), (DomainTEDB)entry.getValue()));
callback
response

Figure 18 Topology discovery workflow

5.5. Test validation definition

In order to verify the functionality of the T-API within the Netphony environment, several tests validation will be presented and will be validated in the following sections. These tests are defined with the aim of validating the connectivity and topology services, this means that the other services of the T-API, such as monitoring or path computation, will not be validated. The next list presents the tests for the provisioning functions:

- Set up an unconstrained optical service between two endpoints.
- Set up an optical service between two endpoints with constraints on the path.
    - Constraints on the path can include:
        - Explicit optical path defined by the orchestrator
        - Constraints on bandwidth
- In case a service setup is not successful, the controller must provide appropriate reason codes.
- Configure a service to employ a restoration/protection path for the current working path.

Next list presents the tests for the topology discovery functions:

- Topology must be able to present Information about the optical nodes, identifiers, location and interconnections between them
- Topology should provide information about the links including optical characteristics, hop by nodes, bandwidth, latency... etc.

5.6. Conclusion

Summarize, the experimentation scenario has been presented in which the tests that will be explained in the later section will be carried out. It has been explained how the tools mentioned in section 3.3 have been used, such as swagger, YANG models and thus generate the documentation for the REST API for both the connectivity service and the topology service. It has also been presented, for both services, how they have been integrated into Netphony. Subsequently, the use cases are presented, which are the creation of an LSP and the discovery of the topology and how is the workflow in both cases, in a schematic way to represent the communication between the components of the ABNO architecture and also through diagrams of sequence to represent the communication between Java code classes. Finally, define the validation tests for the next section.

6. Experimental validation

Once the experimental scenario composed of 30 emulated nodes in virtual machines has been presented in section 5, simulating the optical network of Telefonica Spain, it has been explained how the documentation of the REST T-API has been auto-generated as both for the connectivity service and for the topology service and has been integrated into the Netphony project. Therefore, in this section we will explain how to perform a request for provisioning or topology recognition as well as the process to launch the Netphony ABNO. The scripts implemented in Python simulating an NMS / OSS will be presented. A demonstration of the creation of LSPs by executing the telnet command in the nodes in question will be carried out. Finally, a traffic capture with Wireshark will be exposed to detect the LSPs creation messages and the explanation of these messages.

6.1. Project compilation

Maven is an open source tool and the main objective is to facilitate the compilation and creation of shaded executables. This tool is extremely useful to facilitate the work of the developers, which as in my case, was new for a company and a research project has already started and had to catch up with all the dependencies of the project in Java, having to learn the peculiarities of the new project, if you want to generate a .shaded executable as in my case and then load it into a virtual machine and run it there, or simply compile all the code while passing the validation tests.

Now, with the Maven tool, the process of building any project, regardless of its modules, its dependencies, libraries, etc., is as simple as executing “mvn install” command as mentioned above. Not only the construction, in addition, Maven can perform a complete management of the software project, perform the test run, generate performance reports based on the level of registration we have established and also generate documentation. Finally, thanks to the pom.xml file as mentioned above, maven is able to manage the dependencies between the project modules and their different versions, in the case of libraries, it is not even necessary to download them since it has a remote central repository where the Most libraries are generally used and if there is not always the option to contribute to this repository with new libraries. In Netphony, each module contains a pom.xml file that defines the compilation and manages the dependencies.

Once the classes and methods of the auto generated code have been modified, the objective is to be able to execute it in the development environment that will be the virtual machine mentioned in section 4.2.1, abnoserver, whose address eth0 is 10.95.86.33 and eth1 192.168 .1.200. For this purpose, the "mvn package -P generate-full-jar-ABNO-TAPI" command will be executed locally in the project directory, with "profile-jar-ABNO-TAPI" being the profile that has been configured in the pom.xml and which must be compiled. In the case of compiling the topology module it will be necessary to be placed in the "Netphony TED and BGP-LS peer" project directory, as seen in section 4.2.3 above. In the case of compiling the connectivity service it will be necessary to execute the command having been located in the "Netphony ABNO" directory as seen in section 4.2.2. The result of this compilation, as already mentioned, will generate a .jar executable file, which will later be transferred to the virtual machine through winscp and once there will be executed. This executable should be placed in a particular abnoserver directory. This directory is / usr / local / abno / lib /. The command to execute this executable is inside a script called start_abno.sh which will be deepened in the following section but here is an example to execute the shaded generated for the topology service.

```
sudo java -Dlog4j.configurationFile=target/log4j2.xml  -jar
target/topology-1.3.3-shaded.jar target/example3tapi/TM1.xml
```

6.2. Creation of requests and launching the ABNO

First of all, it is necessary to start ABNO Netphony to initialize a series of processes, to speed up the procedure, the script "start_abno.sh", implemented in bash, is executed, which will carry out the necessary processes. Two arguments are passed in the execution of this script in abnoserver with the command below to run scenario. It is necessary to be located in the next directory, /home/tid/ABNO and launch the restart_abno.sh script.

```
./start_abno.sh "Orchestrator_scenario" "Domain_scenario"
```

- "Orchestrator_scenario" should be stay locate in /home/tid/ABNO/scenarios/ "Orchestrator_scenario". This folder contains the configuration .xml file of ABNO. PCEActivoTopo is the orchestrator scenario for this tests. The files inside of this directory are:
    - ABNOConfiguration.xml
    - BGP4Parameters.xml
    - BGP4Parameters_PCE.xml
    - defaultConfiguration.properties
    - log4j2.xml
    - TMConfiguration.xml
    - Topology.xml
- "Domain_scenario" should be stay locate in /home/tid/scenarios/"Domain_scenario". This scenario should have three subfolders, "gui", "nodeConfig" and "pce". In "gui" folder contains configurations to web graphical user interface. In "nodeConfig" folder contains configuration about the topology. The .xml files should be contains nodes and links information. The list of nodes obtained from this file is used to run the virtual machines using the file Network_TdE_30nodes_SSON.xml to configure the 30 nodes network of Telefonica Spain. By last, "pce" folder contains the log4j2.xml file which is the needed file to use the log4j2 library which is the updated version of the popular and influential log4j library, used extensively throughout the Java ecosystem to obtain logging for any application, both for audit and debugging purposes. Also here are located the configuration file "PCEServerConfiguration_SSON.xml to configure the pce server.

It is necessary to clarify that the script restart_abno.sh is actually executed, passing as arguments the Orchestrator_scenario and the Domain_scenario since this script also kills processes that may be previously running to avoid failures by executing the script ./kill_abno.sh, then executes the abno with the command ./start_abno.sh $ 1 $ 2 passing the arguments. Then kill logs of previous executions with ./kill_logs.sh and restart the logs with ./start_logs.sh. When all the nodes have been initialized, a confirmation message should appear. To verify the Java processes are running the command "ps -ef | grep java" is can be used.

```
./restart_abno.sh PCEActivoTopo TdESSON
```

With the nodes already initialized, requests for reservation of bandwidth can be made between nodes creating LSPs, to make these requests the "curl" tool must be used as already mentioned in section 3.2 with the -X option to use the indicated proxy that is indicated in the same way as the environment variables including the protocol prefix "http: //". The action must also be indicated, which in this case will be the POST verb as previously mentioned in section 3.2. The path must be indicated with the pce address and the abno port "192.168.1.200:445" and the end points will be the nodes between which the LSP is created.

```
curl -X POST -i -H "Content-Type: application/json" -H "Accept: application/json" http://192.168.1.200:4445/config/context/connectivity-service/1/ -d '{"service-port":[{"service-end-point": "192.168.1.1/1"},{"service-end-point": "192.168.1.2/1"}]}'
```

If the LSP has been created correctly, the confirmation message that will be displayed will be as follows.

```
HTTP/1.1 200 OK
Date: Wed, 08 Dec 2017 13:27:28 GMT
Content-Type: application/json
Transfer-Encoding: chunked
Server: Jetty(9.2.z-SNAPSHOT)
```

```
{"code":4,"type":"ok","message":"Created OK"}
```

To discover the topology of a network, the following request must be made, in which, similarly to the connectivity service case, the "curl" tool with the -X option and the POST verb is used, the path is indicated in this case to obtain the topology and the data are dumped in a file "TopologyAnswer.json"

```
tid@abnoserver:~/ABNO$ curl -X POST
http://localhost:8089/config/operations/tapi-topology%3Aget-
topology-list -H "Content-Type: application/json" -H "Accept:
application/json" | python -m json.tool > TopologyAnswer.json
  % Total    % Received % Xferd  Average Speed   Time    Time
Time  Current
                                 Dload  Upload   Total   Spent
Left  Speed
100  212k    0  212k    0     0   294k      0 --:--:-- --:--:--
--:--:--  294k
```

6.3. Tests validation

The objective of this work is the development and validation of the T-API in Netphony. Up to this point, the development has already been explained and it has been possible to verify the validation of the T-API of both the connectivity service and the topology service, since it has been possible to create an LSP between two nodes randomly in a network and has You could discover the network topology correctly and dump it into a file. However, the objective of the next section is to obtain an analysis of the performance of the T-API in the creation and deletion of a large number of LSPs in the 30-node TdESSON topology that simulates the Telefónica Spain optical network.

To make this evaluation, it would be extremely slow to make the LSP creation and deletion requests one by one manually, and it would not provide a reliable system analysis. For this reason, several Python scripts have been implemented to automatically perform the number of requests that are desired for the creation of LSP.

Before beginning to explain the scripts that have been implemented, it is necessary to have an overview of the process of creating LSP between two nodes and what time is going to be measured and how to generate a graph later to draw firm conclusions. As shown in Figure 19, the request comes from the OSS, which will be simulated by the implemented scripts, through the connectivity service of the T-API, arriving at the ABNO controller, which will exchange PCEP messages as seen previously in section 2.3 to ask the PCE to calculate the route, a first PCEP request is necessary, being answered by the PCE with a PCEP response and finally the controller responds with a PCEP initiated. The PCE sends another PCEP initiated to the first node defined in the request as one of the EndPoints, which in turn sends RSVP route messages, as mentioned above in section 4.1.1, to the destination node, having to pass if it is necessary for intermediate nodes. When the messages arrive at the destination node, are sent back, through the nodes with RSVP Resv, when arriving at the beginning node with PCEP Report towards the PCE and from there to the ABNO Controller with another PCEP Report, the ABNO Controller sends message to confirmation via connectivity service from the T-API to the OSS. The time that is measured and determined as the creation of an LSP is since the OSS sends the request until the confirmation is returned.

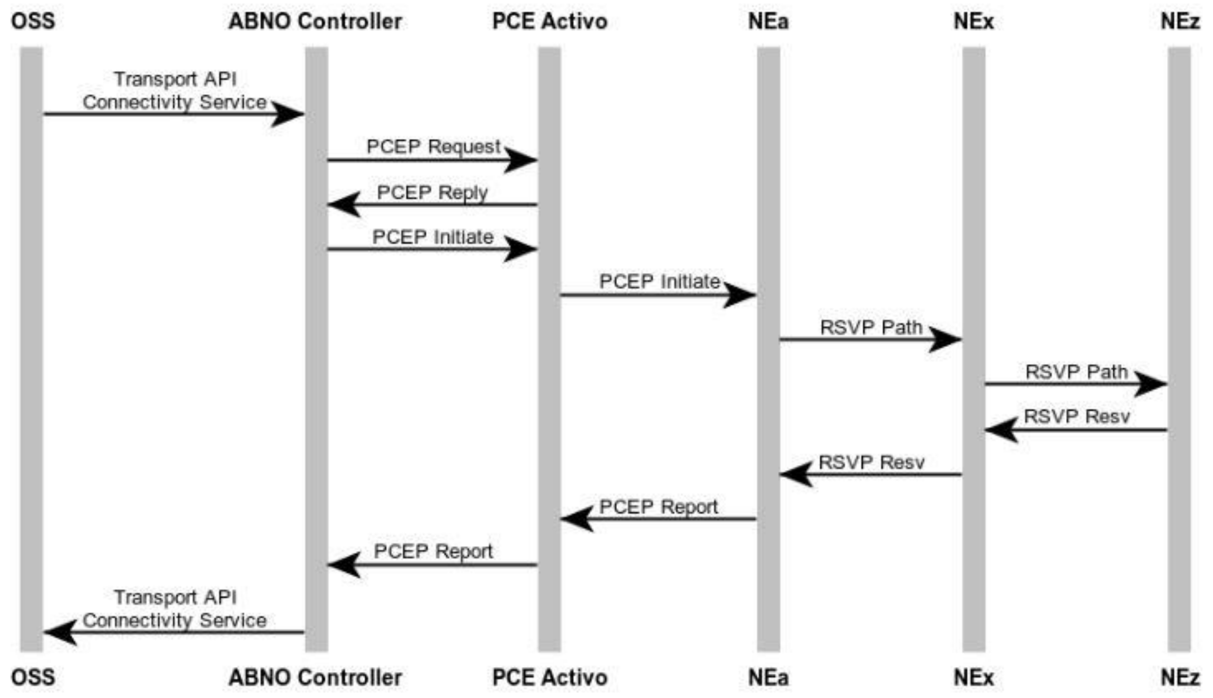


Figure 19 Request management connectivity service

The scripts implemented in python for the creation of LSP and its deletion will be presented below. Three scripts have been implemented, which are: "create_request_v1.py", "RestInterface.py" and "main.py". Each of these scripts has a function and must be executed in abnoserver in a specific order. To be able to carry out its execution it is necessary to install Python in abnoserver and to have granted permissions with the "chmod" command to each script.

The function of the "create_request_v1.py" script is the generation of requests, both for the creation and deletion of LSP. To do this, this script must be executed by passing two data as arguments. The first argument will be the number of nodes of the network that is being executed, being in this case TdESSON with 30 nodes, so the first argument would be 30. The second argument will be the number of iterations that you want to carry out, the number of iterations chosen has been 3000. The call for the execution of this script would then be as follows.

```
tid@abnoserver:~/ABNO$ python create_request_v1.py 30 3000
```

A "request_TdESSON.json" file is generated in response to "create_request_v1.py", with 3000 requests, interspersed between POST and DELETE, and the EndPoints of the source node and the destination node will be random. The reason why an LSP is deleted after creating it is because it avoids congesting the network by creating the same LSP several times. The operation of this script is based on taking the arguments and save them in variables, use them to create a "for" cycle that generates 3000 iterations and within this cycle, it is assigned to the variable "node1" which will be the source node the address " 192.168.1. " being the last digit of the address a random number between 1 and 30, both included, the same procedure for the destination node in the variable "node2", making a check so that the same address is not the origin and destination. Next, six "queryX" variables are generated where the fields necessary for the construction of the request will be saved separately and then concatenated into a variable that will save the final request for the creation of LSP "querypost" and another one for the deletion of the LSP "querydelete". Finally, these interleaved requests will be written to the file "request_TdESSON.json" and the file will be closed once they have been written.

The "RestInterface.py" scritp defines the http protocol header to create REST requests as explained in section 3.2. This script imports the libraries "httplib2", "httplib" and "json" and defines the "httpreq" function that will be imported by the other two scripts with the instruction "From RestInterface import httpReq".

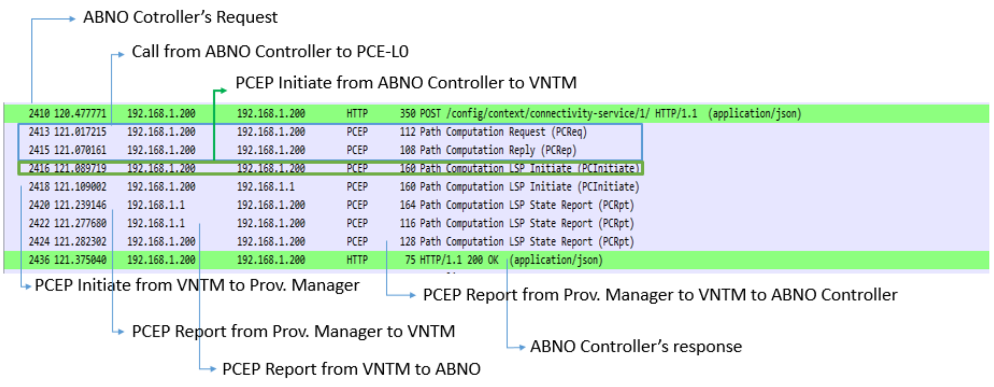
The script "main.py" executes the requests and generates a file "results.json" in which all the information of each request is saved, that is, the origin node and destination node, the POST or DELETE method and the time of start before launching the petition and the time when the request has been made. This is achieved by reading line by line the file "request_TdESSON.json" containing the requests and saving in the variable "request" the number of lines in this file, to then build a loop of as many iterations as it is variable and for each iteration execute the request taking before and after the current time of the system with the instruction "datetime.datetime.now ()" saving the result in one variable before "init" and another after the execution "end". Both the response and the start and end times are saved in the variable "results" whose content is subsequently downloaded into the file "results.json". This file will be treated later as will be explained in the following section to generate the performance evaluation graphs. The instruction to execute this script and the messages that should appear by console are the following.

```
tid@abnoserver:~/ABNO$ python main.py
RESPONSE: {"code":4,"type":"ok","message":"Created OK"}
RESPONSE: {"code":4,"type":"ok","message":"DELETED OK"}
.
.
.
```

Finally, the process to obtain traffic captures with Wireshark and a description of what can be observed in them is explained below. A visually check of the exchange of messages in different protocols made between the ABNO controller, the PCE and the nodes van be seen by this way. To capture the traces and see the established connection between two nodes, the following procedure have to be followed. Inside the abnoserver or in pceserver indistinctly the session must be duplicated and to be located in /traces directory to execute the next "tcpdump" command which is a tool whose main utility is to analyse the traffic that circulates through the network. The option –I is used to print the interface on each dump line and the option "any" indicate to capture by all the interfaces. The option –s means snaplen and 65535 refers the size of the packages. By last, "tracescapture.pcap" is the file name.

```
sudo tcpdump -i any -s 65535 -w tracescapture.pcap
```

Once this command is executed, the new session where it was executed will be listening and in the original session the process to initialize the ABNO with the "restart_abno.sh" can be carried out later to start the process of making the requests. All the messages can be verified, which can be HTTP to establish the connection or all the PCEP messages that are shared between the different entities, thanks to Wireshark. The Figure 20 below is a screenshot in which this exchange can be visualized and verified.


ABNO Cotroller's Request
Call from ABNO Controller to PCE-L0
PCEP Initiate from ABNO Controller to VNTM
PCEP Initiate from VNTM to Prov. Manager
PCEP Report from Prov. Manager to VNTM to ABNO Controller
PCEP Report from Prov. Manager to VNTM
ABNO Controller's response
PCEP Report from VNTM to ABNO

```
2410 120.477771  192.168.1.200  192.168.1.200  HTTP  350 POST /config/context/connectivity-service/1/ HTTP/1.1  (application/json)
2413 121.017215  192.168.1.200  192.168.1.200  PCEP  112 Path Computation Request (PCReq)
2415 121.070161  192.168.1.200  192.168.1.200  PCEP  108 Path Computation Reply (PCRep)
2416 121.089719  192.168.1.200  192.168.1.200  PCEP  160 Path Computation LSP Initiate (PCInitiate)
2418 121.109002  192.168.1.200  192.168.1.1    PCEP  160 Path Computation LSP Initiate (PCInitiate)
2420 121.239146  192.168.1.1    192.168.1.200  PCEP  164 Path Computation LSP State Report (PCRpt)
2422 121.277680  192.168.1.1    192.168.1.200  PCEP  116 Path Computation LSP State Report (PCRpt)
2424 121.282302  192.168.1.200  192.168.1.200  PCEP  128 Path Computation LSP State Report (PCRpt)
2436 121.375040  192.168.1.200  192.168.1.200  HTTP   75 HTTP/1.1 200 OK  (application/json)
```

Figure 20 Wireshark capture protocol messages

First, the ABNO driver receives a request to establish a new connection using the API Transport connectivity service. The ABNO controller makes a request with a PCEP request to the PCE and responds with a PCEPReply. This is because the ABNO driver has the information of the Policy Agent, so it has the decision to create or not the LSP although, for our workflow, the active PCE has complete status, so once it arrives to the PCEP respond from the PCE to the ABNO controller. The ABNO driver sends the PCE started with the endpoints to the L0 PCE. Later, PCE Active sends a PCEP Initiate to the determined node as the source address, in turn sends messages by RSVP protocol (PATH and RESV) to the nodes indicated in the ERO (explicit route object) until it reaches the node determined as destination. RSVP messages can only be viewed if traffic is captured in any of the nodes. Once the LSP has been established, it will notify on the way back, sending an RSVP RESV to the originating node, which in turn will respond with a PCEP report to the active PCE that will notify the PCEP to the ABNO controller. Finally, the ABNO driver will allow the NMS / OSS / Application service to know that the LSP has been right created.

6.4. Summary of the results

The objective of this section is to perform a performance evaluation when creating or eliminating LSP between the different nodes of the network used, TdESSON, using the connectivity service of the T-API. As mentioned in the previous section, the procedure to be able to perform an analysis of the times is to capture the time before making a request and capture it again after making the request, so that the difference between both times can be obtained. This work is carried out in the script explained above "main.py" and the results have been dumped in the file "results.json". This format has been useful to be able to process the data with Excel since it has been easily divisible the information by columns and thus be able to see the data that are object of interest, the times. The format in which the information related to the time to generate the graphics must have been discussed. To be able to visualize the estimated time of creation or deletion of LSP and to be able to deduce in what time value was invested when creating or eliminating the majority of the created LSPs, cumulative distribution function were generated. To perform them, the following instruction has been programmed in Excel.

```
= 1 / COUNT ($ H $ 2: $ H $ 6000)
```

H is the column in which the times were located. H6000 is the last cell, since it is necessary to take into account that 3000 LSP creation requests have been made with the POST method and this implies 3000 removal requests with the DELETE method. While advancing to calculate the values of the cumulative function, the value of the previous cell must be taken, this is column I, therefore, the instruction will be the following for the second row.

```
= I2 + 1 / COUNT ($ H $ 2: $ H $ 6000)
```

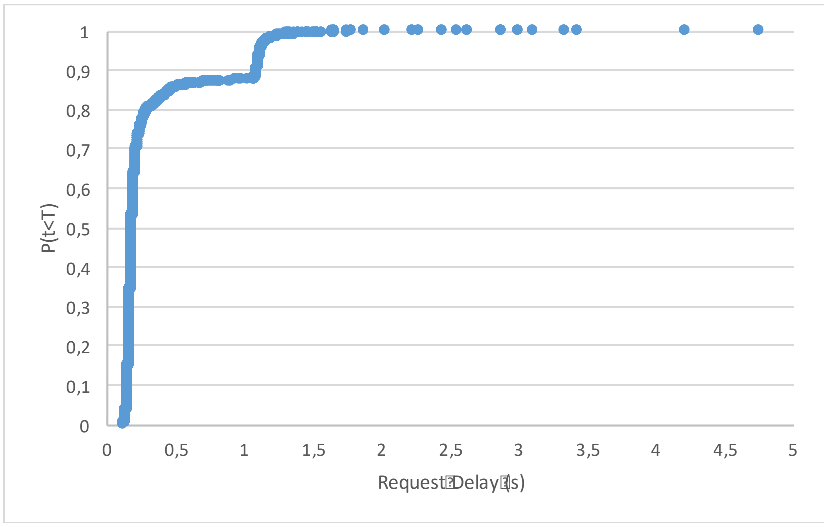


Figure 21 Cumulative histogram with 3000 LSPs created in Telefónica of Spain topology

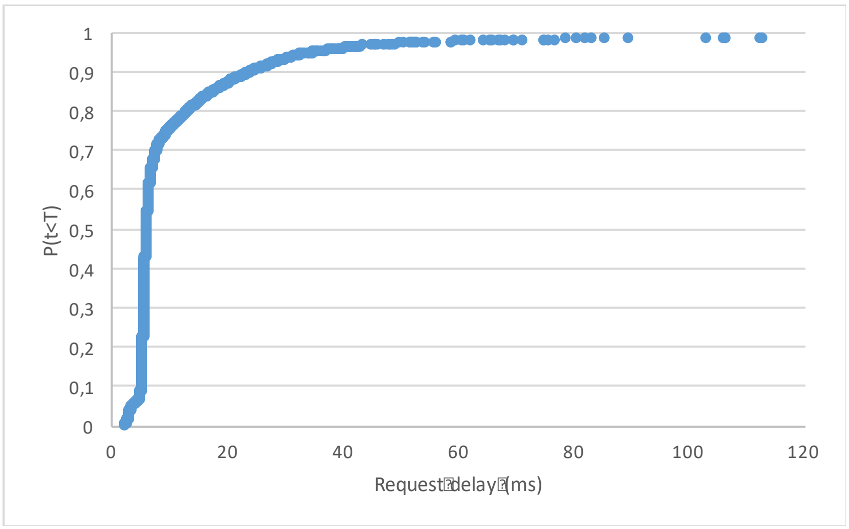


Figure 22 Cumulative histogram with 3000 LSPs deleted in Telefónica of Spain topology

5-

To show the results, we have obtained the cumulative distribution function with 3000 iterations of creating LSP, and then another 3000 iterations for the case of deletion. The previous Figure 21 and Figure 22 show the creation requests and the delay in the elimination requests of the LSP, respectively. The average time to eliminate connections is 18.33 ms, while the 80th percentile is 30.45 ms. Higher delay values can be seen, but these correspond to cases, which are caused due to an additional delay in the communication change between the nodes. This is because in the simulated environment one of the virtual machines of the 30 that are running is likely to overlap with another, when this happens, the virtual machine collapses and when a request for the creation of LSP that requires a reserve of resources tries to enter this node, stays waiting until it can be managed, or until sometime passes and the next request is passed. This for the case of deletion of LSP does not usually happen so often.

In addition, it can also be said that for the elimination of LSP, the average time to eliminate connections is 6.2 ms, while the 80th percentile is 13 ms, as shown in Figure 22. As a conclusion of this section of the performance evaluation, it can be said that for the system of creation of LSP, as for the elimination, with the architecture ABNO and the connectivity service of the T-API, it is a system with delays below the average of the limitations of the data plane, which are of the order of a minute to configure the commercial data components.

6.5. Conclusion

In this section, the compilation using maven of the project has been presented with the integrated services and the creation of the requests to be able to validate the tests presented in the previous section. The validation is the bulk of this section and it has been possible to demonstrate the functionality of the T-API thanks to the logs and traffic capture using Wireshark. A sequence diagram to understand the communication between the ABNO Controller, the PCE Server and the different nodes, has also been presented. Having finalised these tests, it is necessary to obtain data and process them in order to later analyse them and obtain conclusions in the last section. To carry out this undertaking, the data in the JSON file has been exported to an Excel spreadsheet and has been processed with the objective to obtain graphs of the cumulative distribution functions of the created and eliminated LSPs to obtain important parameters for the study as is the average and the 80th percentile.

7. Conclusions and future work

Network operators have implemented optical transmission systems in their production networks for several years. The migration to optical networks technologies is crucial to increase optical fiber utilization for network operators. The control architecture is a key element to take advantage of the capabilities of flexgrid networks. The use of SDN paradigms such as the Transport API combined with PCE / GMPLS technologies, which are implemented in production environments, is an evolutionary solution that facilitates the introduction of SDN in real networks.

The Open Source Netphony suite is composed by a GMPLS control plane to emulate the network elements control, a Path Computation Element with active and stateful capabilities, a Topology Module capable of importing and exporting TE information in different protocols as well as an Application-based Network Operations (ABNO) controller. This framework allows the provision of connectivity services in the network using the SDN and PCE / GMPLS paradigms as demonstrated in this work.

As future work, the capabilities of the monitoring services of the T-API in the Netphony should be expanded and include new plugins that make it fully adaptable and scalable to any topology.

This work has contributed to the following invited journal and conference papers:

- V. López, J. M. Gran, R. Jimenez, J. P. Fernandez–Palacios, D. Siracusa, F. Pederzolli, O. Gerstel, Y. Shikhmanter, J. Mårtensson, P. Sköldström, T. Szyrkowiec, M. Chamania, A. Autenrieth, I. Tomkos and D. Klonidis, "The Role of SDN in Application Centric IP and Optical Networks", Journal of Green Engineering, Vol 6 (3), July 2016.
- V. López, R. Jimenez, O. Gonzalez, L.M. Contreras, J.P. Fernandez-Palacios: Open Source Netphony Suite: Enabling Multi-layer Network Programmability, in Optical Networking Design and Modeling (ONDM), May 2017.
- V. López, R. Jimenez, O. Gonzalez De Dios, and J.P. Fernandez Palacios: Control plane architectures for Elastic Optical Networks, in Journal of Optical Communications and Networking, February 2018, Vol. 10, Issue 2, pp. A241-A249.

8. References

[1] E. Mannie, Ed. Network Working Group "Generalized Multi-Protocol Label Switching (GMPLS) Architecture", RFC 3945, October 2004. https://tools.ietf.org/html/rfc3945

[2] K. Kompella, Y. Rekhter, Network Working Group, "Label Switched Paths (LSP) Hierarchy with Generalized Multi-Protocol Label Switching (GMPLS) Traffic Engineering (TE)", RFC 4206, October 2005. https://tools.ietf.org/html/rfc4206

[3] D. King, A. Farrel, IETF, "A PCE-Based Architecture for Application-Based Network Operations", RFC 7491, March 2015. https://tools.ietf.org/html/rfc7491#section-2.3.1

[4] Diego Lopez. Ramki Krishnan. Sarah Banks. Internet Engineering Task Force (IETF). "Network Function Virtualization". https://datatracker.ietf.org/rg/nfvrg/about/

[5] Software-Defined Networking (SDN) Definition https://www.opennetworking.org/sdn-definition/

[6] "OpenFlow Switch Specification Version 1.0.0 (Wire Protocol 0x01)", December 31, 2009 https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf

[7] ONOS https://onosproject.org/

[8] J. Moy, "OSPF Version 2.", RFC 2328, April 1998. https://www.ietf.org/rfc/rfc2328.txt

[9] E. Rosen, A. Viswanathan, R. Callon, Network Working Group, "Multiprotocol Label Switching Architecture", RFC 3031, January 2001. https://tools.ietf.org/html/rfc3031

[10] A. Bierman, M. Bjorklund, K. Watsen, IETF, "RESTCONF Protocol", RFC 8040, January 2017 https://tools.ietf.org/html/rfc8040

[11] M. Bjorklund, Ed., IETF, "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, October 2010. https://tools.ietf.org/html/rfc6020

[12] R. Enns, Ed., M. Bjorklund, Ed., J. Schoenwaelder, Ed., A. Bierman, Ed. IETF. "Network Configuration Protocol (NETCONF)", RFC 6241, June 2011. https://tools.ietf.org/html/rfc6241

[13] Swagger CODEGEN https://swagger.io/swagger-codegen/

[14] Oracle Corporation "JAX-RS: Java™ API for RESTful. Web Services" Version 2.0 Final Release. May 22, 2013. Editors: Santiago Pericas-Geertsen. Marek Potociar. http://download.oracle.com/otn-pub/jcp/jaxrs-2_0-fr-spec/jsr339-jaxrs-2.0-final-spec.pdf?AuthParam=1525270877_051c13171e1e8ad0e030929ff0aa584f

[15] Transport API (TAPI) 2.0 Overview Version 0.0. August 13, 2017. Open Networking Foundation https://www.opennetworking.org/wp-content/uploads/2017/08/TAPI-2-WP_DRAFT.pdf

[16] Technical Charter (the "Charter") OpenDaylight Project a Series of LF Projects, LLC. https://www.opendaylight.org/wp-content/uploads/sites/14/2018/01/ODL-Technical-Charter-LF-Projects-LLC-FINAL.pdf

[17] E. Haleplidis, Ed, K. Pentikousis, Ed., S. Denazis, J. Hadi Salim, D. Meyer, O. Koufopavlou. IRTF. "Software-Defined Networking (SDN): Layers and Architecture Terminology". RFC 7426. https://tools.ietf.org/html/rfc7426

[18] G. Bernstein, Ed., S. Xu, Y. Lee, Ed., G. Martinelli, H. Harai, IETF, "Signaling Extensions for Wavelength Switched Optical Networks", RFC 7689, November 2015. https://tools.ietf.org/html/rfc7689

[19] "ONF Open Transport (SNOWMASS)" OpenNetworkingFoundation/TAPI https://github.com/OpenNetworkingFoundation/TAPI

9. Annexes

9.1. Implementation of methods and explanation

9.1.1. Connectivity Service

Here the implementation of the methods in the ConfigApiServiceImpl.java classes in the package es.tid.abno.swagger.TAPIConSer.api.impl will be explained.

For this, because through the request with curl POST, we access the path: "/ context / connectivity-service / {uuid} /" in the ConfigApi.java class, we must implement the method "createContextConnectivityServiceConnectivityServiceById" inside the ConfigApiServiceImpl.java class.

```
public Response
createContextConnectivityServiceConnectivityServiceById(String uuid,
ConnectivityService connectivityService, SecurityContext securityContext)
throws NotFoundException {

           log.info("recieved retrieveCallsCallCallById request.");
        log.info(" BW: "+connectivityService.getConnConstraint());

   log .info("conncetivity:" +connectivityService.getServicePort());
   for (ConnectivityServicePort endpoint:
connectivityService.getServicePort()){
   log.info("EP:"+endpoint.getServiceEndPoint());

   }
           Iterator<ConnectivityServicePort> epIter =
connectivityService.getServicePort().iterator();

                String workflowParam = "L0ProvisioningCOPWF";

                Hashtable<String, String> request = new Hashtable<String,
String>();
                request.put("ID_Operation", "1234");
                request.put("Operation", "add");
                request.put("remoteAddr", "TAPI Protocol");
                request.put("Bandwidth", "107.1");
                      String[] source =
epIter.next().getServiceEndPoint().split("/");
                request.put("Source_Node", source[0]);
                      String[] destination =
epIter.next().getServiceEndPoint().split("/");
                request.put("Destination_Node", destination[0]);
                request.put("source_interface", source[1]);
                      request.put("destination_interface", destination[1]);
                /*if (call.getAEnd().getInterfaceId()!= null){
                        request.put("source_interface",
call.getAEnd().getInterfaceId());
                }

                if (call.getZEnd().getInterfaceId()!= null){
                        request.put("destination_interface",
call.getZEnd().getInterfaceId());
                }*/
                String response = null;
                Class<?> act;
                try {
        act = Class.forName("es.tid.abno.modules.workflows."+workflowParam);
                        @SuppressWarnings("rawtypes")
```

```
                  Class[] cArg = new Class[5];

                  cArg[0] = Hashtable.class;
                  cArg[1] = String.class;
                  cArg[2] = LinkedList.class;
                  cArg[3] = ABNOParameters.class;
                  cArg[4] = HashMap.class;

                  Object[] args = new Object[5];
                  args[0] = request;
                  args[1] = response;
                  args[2] = ABNOTAPIController.getPath_Computationlist();
                  args[3] = ABNOTAPIController.getParams();
                  args[4] = ABNOTAPIController.getOPtable();

                  WorkflowTAPIworkflow =
(WorkflowCOP)act.getDeclaredConstructor(cArg).newInstance(args);

                  workflow.handleRequest();
                  response = workflow.getResponse();
            }
            catch (Exception e1)
                  {
                        log.info(UtilsFunctions.exceptionToString(e1));
                              }
        log.info("response: "+response);

        return Response.ok().entity(new
ApiResponseMessage(ApiResponseMessage.OK, "Created OK")).build();
    }
```

In addition, from this method, the class L0ProvisioningCOPWF.java is called for both the creation of paths, and for deletion. In the delete case, has been implemented from scratch, since there was not the delete option implemented for the COP.

Other steps that have been necessary to implement is in the class ConnectivityServicePort.java in the package es.tid.abno.swagger.TAPIConSer.model, @JsonProperty ("service-end-point") to be able to call from the query then to service-end-point without and the same was done in the ConnectivityService.java class, @ JsonProperty ("service-port").

```
    @JsonProperty("service-end-point")
  public ConnectivityServicePort serviceEndPoint(String serviceEndPoint) {
    this.serviceEndPoint = serviceEndPoint;
    return this;
  }
```

9.2. PCE and nodes logs

By using the telnet tool you can see the information about routes created in pceserver.

```
telnet 192.168.1.200 6666
PCE:>help
Available commands:
show topology
queue size
show reachability
set traces on
set traces off
show interDomain links
show sessions
send update
show lsps
quit

PCE:>show topology
Nodes:
     /192.168.1.1
     /192.168.1.2
     /192.168.1.3

Intradomain Link list:
        /192.168.1.1:2-->/192.168.1.3:1 NumFibers = 1 Bitmap:
{00000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000} Reserved:
{00000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000}
        /192.168.1.3:1-->/192.168.1.1:2 NumFibers = 1 Bitmap:
{00000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000} Reserved:
{00000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000}
        /192.168.1.2:2-->/192.168.1.3:2 NumFibers = 1 Bitmap:
{00000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000} Reserved:
{00000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000}
        /192.168.1.3:2-->/192.168.1.2:2 NumFibers = 1 Bitmap:
{00000000
000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000} Reserved:
{00000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000}
```

```
        /192.168.1.1:1-->/192.168.1.2:1 NumFibers = 1 Bitmap:
{00000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000} Reserved:
{00000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000}
        /192.168.1.2:1-->/192.168.1.1:1 NumFibers = 1 Bitmap:
{00000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000} Reserved:
{00000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000}
PCE:>quit
bye!
Connection closed by foreign host.
```

Similarly, we can check the view in the nodes.

```
tid@pceserver:~$ telnet tn1 6666
Trying 192.168.1.1...
Connected to tn1.
Escape character is '^]'.
*********************************************************** ROADM
CONSOLE
USER INTERFACE
************************************************************

ROADM Main Menu:

        1) Configure ROADM (WSON)
        2) Configure ROADM (Flexigrid)
        3) Turn off the ROADM
        4) LSPs Management NODE
        5) Show Topology NODE

Please, choose an option
ROADM:>5
Nodes:
        /192.168.1.1
        /192.168.1.2
        /192.168.1.3
Intradomain Link list:
        /192.168.1.1:2-->/192.168.1.3:1 NumFibers = 1 Bitmap:
{000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000} Reserved:
{000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000}
        /192.168.1.1:1-->/192.168.1.2:1 NumFibers = 1 Bitmap:
{000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000} Reserved:
{000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000}
Interdomain Link list:
ROADM Main Menu:

        1) Configure ROADM (WSON)
        2) Configure ROADM (Flexigrid)
        3) Turn off the ROADM
        4) LSPs Management NODE
        5) Show Topology NODE

Please, choose an option
ROADM:>
```

The procedure to launch the ABNO in PCEServer as is have been explained:

```
cd ABNO
./restart PCEActivo TdESSON
```

For this, a request was implemented that accesses the most basic path of the connectivity service to obtain simply the confirmation 200 OK

```
curl   -X   POST   -i   -H   "Content-Type:   application/json"   -H   "Accept:
application/json"      http://192.168.1.200:4445/config/context/connectivity-
service/1/            -d             '{"service-port":[{"service-end-point":
"192.168.1.3/1"},{"service-end-point": "192.168.1.2/1"}]}'
```

And now the following is the query implementation, with the source node and the port as input parameters, as well as the destination node and the destination port:

To see the LSP creation, we can connect in the tn1.

```
tid@pceserver:~$ telnet tn1 6666
Trying 192.168.1.1...
Connected to tn1.

ROADM Main Menu:

        1) Configure ROADM (WSON)
        2) Configure ROADM (Flexigrid)
        3) Turn off the ROADM
        4) LSPs Management NODE
        5) Show Topology NODE

Please, choose an option
ROADM:>4

Node Management Main Menu:

Available commands:

show LSPs
set LSP
teardown LSP
help
set traces on
set traces off
back

print eros

quit

NODE:>show LSPs

LSP id: 1  ---->  Source: /192.168.1.3 - Destination: /192.168.1.2
Node Management Main Menu:
quit
NODE:>quit
bye!
Connection closed by foreign host.
```

It had been implemented also the method “deleteContextConnectivityServiceConnectivityServiceById” inside of ConfigApiService.java class.

```
curl -X DELETE -i -H "Content-Type: application/json" -H "Accept:
application/json"
http://192.168.1.200:4445/config/context/connectivity-service/1/ -d
'{"service-port":[{"service-end-point": "192.168.1.3/1"},{"service-
end-point": "192.168.1.2/1"}]}'
```

9.3. Python Scripts

create_request_v1.py:

```
#!/usr/bin/python
__author__ = 'Rodrigo Jimenez Rios TID'
import json
import sys
import datetime
import random
from RestInterface import httpReq

#listresp=[]
f=open("ejemplo_de_request_auto.json",'w')
x=int(sys.argv[1])
y=int(sys.argv[2])

print(x)
print(y)
method='POST'
method2='DELETE'
bw='107.1'

f.write('[ \n')
for r in range(y):
    #Random access to each of these positions
    i=random.randint(1,x)
    node1="192.168.1."+str(i)+"/1"
    j=random.randint(1,x)
    node2="192.168.1."+str(j)+"/1"
    while(node2==node1):
        j=random.randint(1,x)
        node2="192.168.1."+str(j)+"/1"
#example:
#  {"src":"192.168.1.1/1","dst":"192.168.1.2/1","bw":107.1,"method":"POST"},
    #put randomly the end points
    query1='{"src":"'
    query2='","dst":"'
    query3='","bw":'
    query4=',"method":"'
    query5='"}'
    query6='"} '
    querypost= query1 + node1 + query2 + node2 + query3 + bw + query4 + method
+ query5
    querydelete= query1  + node1  + query2  + node2  + query3  + bw  + query4  +
method2 + query5
    #query='{"src":                                        "'+node1+'","dst":
"'+node2+'","bw":'+str(bw)+',"method":"'+method'"} '
    #listresp.append(resp)
    f.write(querypost)
    f.write(",\n")
    f.write(querydelete)
    f.write(",\n")
fquery= query1  + node1  + query2  + node2  + query3  + bw  + query4  + method +
query6
f.write(fquery)
f.write("\n ]")
#print "query: " +query
print "argv: " +str(x)
f.close()
```

RestInterface.py:

```
#! /usr/bin/python
EXIT__author__ = 'Rodrigo Jimenez Rios TID'

import sys
import threading
import httplib2, httplib
import json


def httpReq(baseUrl, method, jstr=""):
    h = httplib2.Http(".cache")
    #h.add_credentials('admin','admin')
    content=""
    try:
        respaux,  content  =  h.request(baseUrl,  method,  headers={'Content-
Type': 'application/json; charset=UTF-8'},body=jstr)
        #print "URL: "+baseUrl
        #print "METHOD: "+method
        #print "DATA: "+jstr
    except IOError, e:
        print "NETWORK UNREACHABLE: "+str(e.errno)
    print "RESPONSE: "+content
    return content
```

main.py:

```
#! /usr/bin/python
__author__ = 'Rodrigo Jimenez Rios TID'

import json
import datetime
from RestInterface import httpReq

with open("/home/tid/ABNO/ejemplo_de_request_auto.json","r") as file:
    requests=json.loads(file.read())
i=0
results=[]
for r in requests:
    #Montar JSON
    js={"service-port":[{"service-end-point": r['src']},{"service-
end-point": r['dst']}]}

    #Montar URL
    url="http://192.168.1.200:4445/config/context/connectivity-
service/1/"
    #Tiempo antes de ejecutar
    init=datetime.datetime.now()

    #Ejecucion
    resp=httpReq(url,r['method'],json.dumps(js))

    #Tiempo despues de ejecutar
    end=datetime.datetime.now()

    #Lo guardamos en la variable results. TODO Comprobar el formato
de la respuesta
    results.append({"request":r,"time":str(end-
init),"response":json.loads(resp)})

f=open("results.json",'w')
f.write(json.dumps(results))
f.close()
```