Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER THESIS

TITLE: Design, deployment and validation of SDN controller for metro/access optical switching nodes

MASTER DEGREE: Master of Science in Telecommunication Engineering & Management (MASTEAM)

AUTHOR: Rafael Montero Herrera

DIRECTOR: Salvatore Spadaro

DATE: September, 26th 2015

Título: Diseño, despliegue y validación de un controlador SDN para nodos de conmutación óptica en redes metro/acceso.

Autor: Rafael Montero Herrera

Director: Salvatore Spadaro

Fecha: Septiembre, 26 del 2015

Resumen

El actual documento presenta el diseño y la implementación de OPTNODE, una nueva aplicación SDN que corre sobre la capa de abstracción de Opendaylight, la cual está definida por modelos (MD-SAL). La aplicación tiene como propósito manejar nodos de conmutación óptica en el sentido de capturar información del nodo como el ID, la dirección IP, el número de puertos, datos de puertos y datos de longitudes de onda, entre otros.

La aplicación está a su vez proyectada para proveer *Remote Procedure Calls* (RPC) a las aplicaciones que se encuentren corriendo sobre el controlador (*Northbound Apps*) para que estas sean capaces de ejecutar operaciones específicas, tales como leer cierta información operacional del nodo o configurar conexiones sobre el mismo.

Para ejecutar estas operaciones, es necesaria la interacción de OPTNODE con otras aplicaciones/plugins para el intercambio de información específica, este intercambio se realiza a través de subscripciones por parte de la aplicación en calidad de consumidor o proveedor dependiendo de la acción requerida. Dentro del controlador, las principales interacciones de OPTNODE están definidas hacia la aplicación *Openflowplugin* y a la librería de *Opendaylight*.

Debajo del controlador (*Southbound*), el protocolo OpenFlow es empleado para la comunicación con el nodo óptico, que a su vez está controlado por una FPGA. La interconexión entre la FPGA y el controlador está hecha a través de la implementación de un agente OpenFlow, el cual se encarga de traducir la información que va desde la FPGA hacia el controlador y viceversa.

Dado que el transporte de datos requerido se quiere administrar de una manera simple (entre el agente y el controlador), se eligió la versión 1.0 del protocolo OpenFlow para ser implementada en el agente, donde la información referente al nodo es enviada al controlador dentro de un mensaje OF_FEATURES_REPLY y el agente recibe las peticiones de configuración desde el controlador a través de mensajes OF_FLOW_MOD. Estos últimos son traducidos y enviados hacia la FPGA.

Title: Design, deployment and validation of SDN controller for metro/access optical switching nodes

Author: Rafael Montero Herrera

Director: Salvatore Spadaro

Date: September, 26th 2015

Overview

This document presents the design and implementation of OPTNODE. OPTNODE is a new SDN application that runs on top of Opendaylight's abstraction layer. The application has been designed using a set of models (MD-SAL), which define its configuration and behaviour. OPTNODE is intended to manage optical switching nodes, that is, capturing the node's most important information such as ID, IP address, number of ports, port data, and wavelength data, among others.

The application is also projected to provide *Remote Procedure Calls* (RPC) so other applications running on top of the controller (*Northbound Apps*) would be able to call specific operations, such as reading operational information from the node or configuring cross-connections.

In order to execute these operations, the OPTNODE application interacts with other applications/plugins of the controller to exchange specific information. In particular, it can register itself as a consumer of a service provided by another plugin or as a producer of a service. The main interactions for this implementation are established with the *Openflowplugin* and the *Opendaylight* Inventory modules.

In the *Southbound* (below the controller), the OpenFlow protocol is used in order to interact with the optical node, which at the same time is controlled by an FPGA. The interconnection between the FPGA and the controller is realized through the deployment of an OpenFlow agent, which is capable of translating information from the FPGA to the controller and in the opposite direction, as well.

Since the required transportation of the data is intended to be managed as simple as possible (between agent and controller), the version 1.0 of the OpenFlow protocol has been chosen to be implemented in the agent. In this way, the optical node's information is sent to the controller via a OF_FEATURES_REPLY message and the agent receives configuration requests from the controller through OF_FLOW_MOD messages. The agent is responsible for translating this information and sending it to the FPGA.

INDEX

INDEX OF FIGURES

INDEX OF TABLES

INDEX OF CODE

INTRODUCTION

The breakthrough of software defined networking (SDN) over the past years has brought new ideas to the networking world, where arguments like “programmability”, “centralized-view” and “separation of planes” are the most identifiable ones. Among the different features that a software defined approach can offer, the ability to establish a centralized control platform that does not require physical centralization allows the flexibility to use the data plane as it was always intended to be, as a resource.

A number of implementations of this new networking paradigm have appeared recently. SDN aims to support different southbound protocols to communicate the control plane and the data resources. Among these protocols, the use of OpenFlow has shown a useful application in packetized operating networks.

Regarding optical applications, it has taken some time to see use cases applied to this area. Although some extensions have been defined for the OpenFlow protocol in versions 1.3 and 1.4, OpenFlow still lacks of well-defined support for optical transmission. This means that there are still some barriers related to the application of SDN in optical networks.

Therefore, this document presents a proposal to allow optical switching nodes to connect to an SDN controller in order to be managed and exposed, so a client could set up configurations or read specific data from them. In order to achieve this integration, the design and deployment of an SDN application and an agent between the controller and the physical node has been considered. The application should handle inside controller interaction with other apps/plugins in order to allow the flow of information from the agent to the client and back. On the other hand, the agent should be able to perform simple translation of data between an FPGA controlled node (e.g., an optical switching node) and a protocol plugin inside the controller. The document follows this structure:

- **Chapter 1** provides a review of the main concepts related to SDN and the OpenFlow protocol, considering their possible applications.
- **Chapter 2** presents a characterization of different SDN controllers and the selection of the most appropriated one for the project.
- **Chapter 3** describes the architecture of the Opendaylight controller.
- **Chapter 4** presents the design of an SDN application and an OpenFlow Agent for a proposed scenario.
- **Chapter 5** presents the deployment, validation and testing of the complete scenario.
- **Chapter 6** summarizes the main conclusions.

CHAPTER 1. SDN & OPENFLOW

This chapter provides an overview of the general concepts of Software Defined Networking, regarding important topics such as understanding, application, and adoption of this new kind of networking. Likewise, the relation of SDN with the OpenFlow protocol is also defined in order to understand how both can work, together or not, and what advantages can provide the use of this protocol.

1.1. SDN Basics

Before jumping into definitions, it is important to consider how the SDN concept was created and to understand that it is not just some modern idea trying to change how networking is performed today but, instead, it is meant to solve issues that are present in current networks. After reviewing this, it would be possible to comprehend how SDN works and how it can be applied to specific use cases.

1.1.1. The Need for Network Innovation

Conventional networks present a static architecture that show a lack of adaptation in front of dynamic storage and computing needs of today´s networking environments. Conventional networks, here, refer to the ones built on a hierarchical basis, commonly arranging Ethernet switches on a tree structure and supporting a client-server computing design that has been predominant in the last years. This model is not able to handle the changes that new trends like virtualization, cloud services and the explosion of mobile content and devices bring to the networking industry.

As it can be seen in the next figure, specialized components of the standard network infrastructure (see [1]) have created a vertically complex, closed and proprietary model that is difficult to program, operate or troubleshoot.

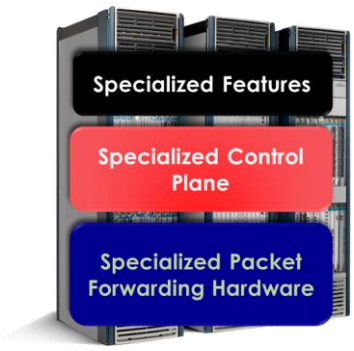

Fig. 1.1. Current Network Limitations [1]

[1] Krishnaswamy, U, “Innovation in SDN Tools and Platforms”, ON.LAB, 4-7 (2013).

In this kind of model, it is also very difficult to integrate new features or services or, as it was exposed before, to support the needs that actual trends require (see [2]). Some of these trends include:

- **Cloud Services:** The rise of these types of services has brought users with on-demand needs of access to IT resources such as applications or infrastructure. The provisioning of public or private clouds not only requires a highly secure environment, but also an elastic scaling of computing, storage and network resources.

- **Big Data:** The amount of data that needs to be handled nowadays requires massive processing in thousands of interconnected servers, which fuels a constant demand for additional network capacity.

- **IT "Consumerization":** The increase in the use of personal mobile devices pressures IT to provide networks that are capable to connect all these devices without compromising security and flexibility.

- **Changing Traffic Patterns:** Data Centres have changed their traffic model from a "north-south" pattern to a more "east-west" one, where there is more machine-to-machine traffic. Users have also begun to access content and applications from any device at any place and time, which has also changed networking patterns.

In front of these new requirements, conventional networks are beginning to suffer from architectural limitations that set up the need for a change in the current way of networking. Some of these limitations include:

- **Complexity:** The current applied networking technology makes it difficult to make networking changes like adding or moving devices, policies, features or services. These types of actions also risk service interruption and are time consuming.

- **Scalability:** The time-demanding link oversubscription commonly used to scale the network does not apply for dynamic traffic patterns of today's virtualized networks. At carrier level, this limitation becomes even more critical, as service providers are required to offer differentiated services to customers implementing key operations in order to stay competitive.

- **Vendor Dependence:** Vendor's equipment product cycles limit the network ability to respond to new user demands. This is potentiated also by the lack of standard open interfaces, which does not let operators to adapt the network to specific environments.

[2] ONF White Paper, "Software-Defined Networking : The New Norm for Networks", Open Networking Foundation, 3-7 (2012).

1.1.2. Fundamentals of SDN

The proposal that SDN brings to the networking world is based on fundamental characteristics, which were established (see [3]) in order to fight against the current network limitations exposed in the previous section.

1.1.2.1. Separation of Planes

The first characteristic consists on the separation of the control and data planes. While forwarding functionality such as tables and logic remain in the data plane, the intelligence is kept in the control plane. Hence, fundamental actions like forward, drop, consume or replicate packets are the ones performed at the data plane by determining the correct action after looking at an address table. On the other hand, the protocols, logic and algorithms that are used to program the data plane stay at the control plane, since they may require a global knowledge of the network in order to operate. Otherwise, in traditional networks, each switch has its own control plane that runs routing or switching protocols in order to synchronize forwarding tables between all devices in the network.

In SDN, the control plane is migrated from the switching node to a logically centralized controller so they no longer co-reside in the same device.

1.1.2.2. Centralized Control

Following the previous characteristic, the idea of moving the control software from devices to a centralized controller allows simplifying the management of the network. By using this approach, a software-based centralized controller is able to manage the network having a global view of it and using higher-level policies.

It is also important to understand that centralized control does not mean physical control centralization, which would lead to a single point of failure in the network and difficulties for horizontal scalability. Instead, SDN proposes a logically centralized control that would be able to provide management of the network without compromising security or scalability.

1.1.2.3. Openness

Another characteristic of the SDN proposal is that interfaces should be exposed in a well-documented open standard, avoiding the establishment of proprietary ones. This openness will allow the community to research into innovative methods of network operation and will potentiate the customization of SDN platforms to specific environments.

---

[3] Goransson, P. and Black, C, “How SDN Works”, Software Defined Networking - A Comprehensive Approach, 59-61 (2014).

However, the most important goal of openness is vendor interoperability. To do this, it will be necessary to secure both southbound and northbound open interfaces. In this way, a competitive environment will appear and this should lead to reduce the cost of network equipment.

1.1.2.4. Simplification of Devices

As control functionalities would be taken out of switching devices, they can become simpler, allowing the reduction of equipment costs and letting the central control to handle devices only as resources, basically ordering these how to behave.

Although this may seem an appropriate solution, it may take time to be applied, especially in commercial equipment where a stage of adaptation has to be considered. This stage may include the use of hybrid equipment that could operate, depending on the use case, using the control plane inside the switching device or not, in order to support a transition to SDN.

1.1.2.5. Network Automation & Virtualization

The abstraction of the distributed state, forwarding and configuration plays an important role in the programmability of SDN networks; it allows setting up complex actions through programming abstractions, specifying forwarding behaviours without the need of knowing vendor-specific hardware and applying actions in the network without the need of considering how the data plane is going to implement them.

By using southbound interfaces the control plane is able to communicate to data resources. One common example of an implementation of the southbound interface is the OpenFlow protocol. On the other hand, northbound interfaces provide access to applications, which can be programmed to manage specific network resources in certain way, allowing automating operations without the need of knowing individual characteristics of devices. This abstraction also considers an approach to virtualization, where devices connected through southbound interfaces can be virtual or physical, and applications managing these resources will be unaware of this fact.

1.1.3. Architecture

The Open Networking Foundation (ONF) [4], an organization devoted to the development and standardization of SDN, provides a graphical representation of the SDN architecture that can be seen below. This architecture is basically defined, as explained in previous sections, by the separation of planes and the interaction between them through the northbound and the southbound open interfaces.

[4] Open Networking Foundation <www.opennetworking.org>

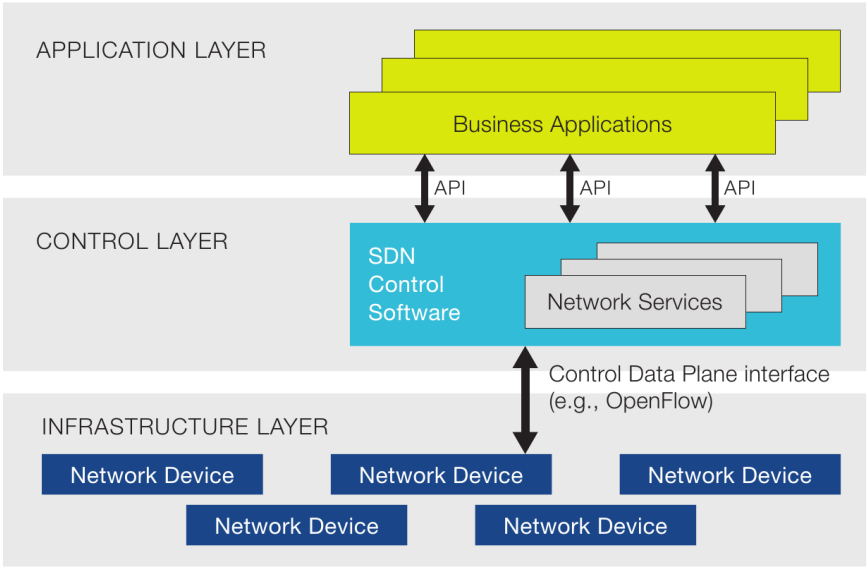

Fig. 1.2 SDN Architecture [5]

The architecture defines three separate layers, where the infrastructure layer, or more recently known as data plane, comprises network devices that expose their capabilities to the control layer via a southbound interface. By knowing data plane capabilities, the control layer is then able to optimize performance, provide granular control over network resources and to deliver relevant information and services up to the SDN applications running con the application layer. These ones, on the other hand, connect to the control layer via northbound interfaces in order to let the control know about their network requirements.

According to the ONF, this SDN proposed architecture has the following characteristics that meet the fundamentals reviewed in the previous section:

- **Direct Programmability:** As the control plane is decoupled from the data plane forwarding functions, it is possible to directly program network control.

- **Agility:** Network-wide traffic flow can be dynamically managed in order to be adjusted to changing needs.

- **Central Management:** A global view of the entire network is logically centralized at the SDN controller and can be used by applications or policy engines running on top of the control plane.

- **Programmatic Configuration:** Dynamic and automated SDN programs let network managers to configure, optimize, manage and secure network resources without the need of proprietary software.

[5] ONF - SDN 3 Layers <www.opennetworking.org/sdn-resources/sdn-definition>

- **Open Standards Based:** Open standards implementation avoids the need of vendor-specific devices and protocols while simplifying the network design and operation.

1.1.4. SDN Components

In order to better understand how SDN works, it is necessary to characterize the role of each of its components inside the architecture and how these components relate to each other (see [6]).

1.1.4.1. SDN Application

An SDN application can be defined as a program running on top of the controller and is connected to it through a northbound interface. An application may retrieve or consume the abstracted view of the network from the controller, so it can make internal decisions based on this information.

The application is conformed of a logic base and a northbound agent/driver that enables the communication to the controller. An SDN application may also represent some level of abstraction in network control, providing flexibility and programmability. Types of SDN applications may include programs for network virtualization, network monitoring, intrusion detection (IDS) and flow balancing, among many other possibilities.

1.1.4.2. SDN Controller

The SDN controller is basically a logically centralized unit that is in charge of:

- Translating requirements from SDN applications down to the SDN devices at the data plane.
- Providing an abstract view of the network to the applications, considering also events and statistics.

A controller, as it can be seen in the next figure, may be composed of several northbound interface agents that can offer accessibility in different ways, through REST interfaces, or through Java/Python APIs to make an example.

The controller logic resides below these interfaces, where several modules can offer functionalities like network and topology discovery, device management and specific content related databases. Southbound interface agents are the final components of the controller, providing connection to the data resources or also called SDN devices, in this case the OpenFlow Plugin shows an example of a southbound protocol agent.

[6] ONF, “SDN Architecture Overview”, Version 1.0, Open Networking Foundation, (2013).

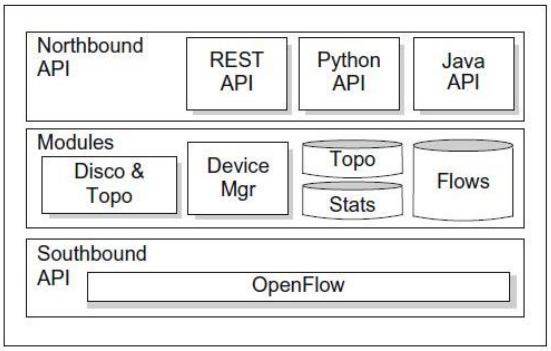

Fig. 1.3 SDN Controller Architecture [7]

Even though the controller architecture seems to show a “north-south” design, it also allows setups like federation of multiple controllers, hierarchical connections, communication interfaces between different controllers or virtualization/slicing of network resources.

Depending on the controller and, as it will be seen in next chapters of the present document, components can vary. Southbound interfaces may offer other protocol interfaces like OF-Config, NetConf or OVSDB, controller modules can show different or new functionalities and finally northbound interfaces can allow a variety of northbound APIs for SDN applications interconnection or, in some cases, just a single one.

1.1.4.3. SDN Device

An SDN device is a networking device, logical or physical, that is capable of exposing visibility and control over its forwarding and data processing capabilities to the upper control layer. Depending on the type of SDN devices, its composition can vary:

- **SDN Logical Device:** A logical, virtual or software device, is composed of an SDN agent that allows to connect the device to the controller, an abstraction layer that contains flow tables allowing to perform actions on incoming traffic or, in the case of no matches, sent it to the controller for further processing. If the traffic is matched, software-based packet processing functions at the bottom of the architecture should handle packets.

- **SDN Physical Device:** A physical device as it can be seen in the next figure shows a similar composition to the logical ones, where instead of packet processing software, the packet processing functions are embodied in the hardware for packet-processing logic.

[7] Goransson & Black, “Software Defined Networks - A Comprehensive Approach”, 1st Edition, Chapter 4, (2014).

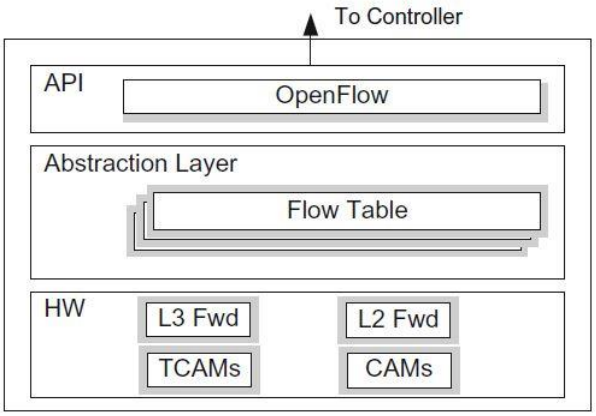

Fig. 1.4 SDN Physical Device Architecture [7]

Besides physical and logical SDN devices, there exists another type considered as the hybrid approach, which allows supporting both legacy an SDN networks. Details about this type of device will be further analysed in the OpenFlow protocol section.

1.1.4.4. SDN Northbound Interface

Northbound interfaces work between the applications and the SDN controller, providing network view abstraction and direct communication of network requirements through actions such as events or methods as it can be seen in the next figure.

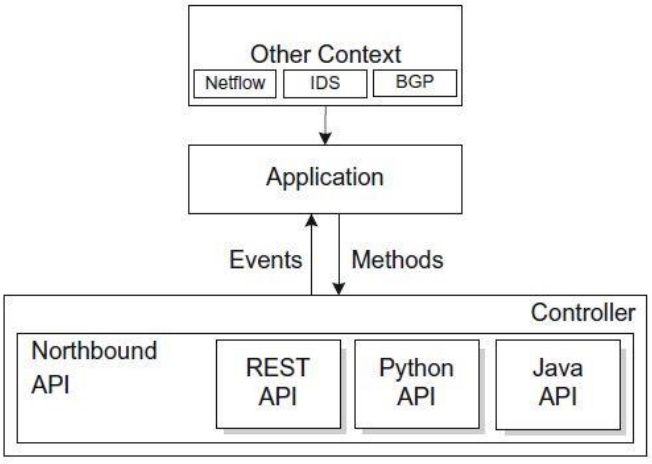

Fig. 1.5 SDN Northbound Interface Connection [7]

Besides providing connection to SDN applications, northbound interfaces are also used to integrate the controller with automation stacks, like Puppet or Chef, as well as with orchestration platforms such as OpenStack.

The northbound interface is nowadays a crucial topic in seek for SDN standardization, since an open interface is required to guarantee vendor-neutrality and interoperation between the applications and the controller.

Even though efforts have been taken to solve this matter, there is still no open interface reaching for standardization on the northbound of the SDN architecture. Among the many possibilities, a Representational State Transfer (RESTful) interface has been widely used by applications lately, but it is still far from being a standard.

1.1.4.5. SDN Southbound Interface

The southbound interface connects the controller and the SDN network devices, providing functions such as:

- Programmatic control of forwarding operations.
- Capabilities advertisement.
- Statistics reporting.
- Event notification.

The Southbound interface is able to support many different protocols. However, it is important to take special consideration on the OpenFlow protocol, an open source protocol designed to provide open connection in the southbound and that is seeking for standardization along SDN since the very beginning. Further analysis on this protocol and its open interface is done in the next section.

1.1.4.6. SDN Drivers & Agents

Interfaces are implemented on devices, applications and the controller by means of an interface driver/agent, capable of providing connection to its correspondent “north” or “south” pair.

Although these drivers/agents usually come already integrated in the components, there exist cases, as the present project, where there is no agent in the network component, so an external driver/agent is required to connect the device to the SDN framework.

1.2. OpenFlow Basics

OpenFlow is a standard protocol used to implement the Southbound interface that interconnects the control and data plane in the SDN architecture. It provides direct access to forwarding functionalities of network devices, physical or virtual, allowing moving control out of the devices to the logically centralized control software.

1.2.1. OpenFlow Implementation

The protocol specifies basic set of instructions that can be used by external applications to directly program the forwarding functions of a network device. These “instructions” define specific rules according to match/action tasks.

As it can be seen in the next figure, OpenFlow is implemented on both sides of the southbound interface by means of drivers/agents. A secure channel is established to set up the connection of the device to the controller.

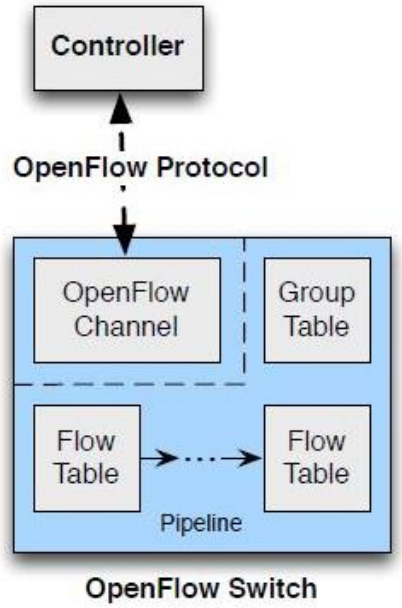

Fig. 1.6 OpenFlow Implementation [8]

Inside the device, there exists a flow table and a group table (the latter is not present in OF version 1.0, see [8]), which allow packet lookups and forwarding. OpenFlow uses the concept of flows in order to identify network traffic based on match rules, the previously mentioned “instructions”, which can be programmed and managed directly by the controller.

[8] ONF, “OpenFlow Switch Specification”, Version 1.4.0, Open Networking Foundation, (2013).

This per-flow basis programming provides extremely granular control that enables responding to real-time changes at the application, user and session levels. IP-based routing does not currently support this kind of response.

OpenFlow is nowadays the only standardized SDN protocol that allows direct manipulation of the forwarding plane of network devices, becoming one of the key enablers for SDN adoption.

1.2.2. Benefits of OpenFlow-Based SDN Networks

OpenFlow-based SDN technologies allow enterprises and carriers to set up competitive networks that consider benefits such as:

- **Centralized control of multi-vendor environments:** SDN Control software allows managing devices from different vendors, and using orchestration and management tools to deploy actions through the entire network.

- **Reduced complexity through automation:** OpenFlow-based networks allow the use of automation tools able to carry out tasks that are currently done manually.

- **Higher rate of innovation:** SDN accelerates innovation by allowing programming the network in real-time to meet specific requirements.

- **Increased network reliability and security:** Higher level configurations and policies can be defined by using SDN and implemented through the OpenFlow protocol. A complete visibility of the network also enhances security, traffic engineering, quality of service (QoS) and access control.

- **Granular network control:** Policies can be applied at a very granular level enabling cloud operators to support multi-tenancy while maintaining traffic isolation, security and elastic resources for users that share the same infrastructure.

- **Better user experience:** Centralization of network control and availability of state information to higher-level applications allows rapid adaptation of the network to the dynamic user needs.

1.2.3. OpenFlow Versions

Since the release of OpenFlow v1.0.0 in 2009, the protocol has seen releases that have brought new features such as:

- **OF v1.1.0:** Support for MPLS, VLANs, multipath, multiple tables, group tables and logical ports.

- **OF v1.2.0:** Support for extensible headers (in_match, packet_in, set_field), IPv6.

- **OF v1.3.0:** Support for tunnelling, per-flow traffic meters, provider backbone bridging.

- **OF v1.4.0:** Support for optical port properties, synchronized tables, more extensible wire protocol, flow monitoring, and vacancy events.

- **OF v1.5.0:** Support for egress tables, extensible flow entry statistics, packet type aware pipeline, TCP flags matching and scheduled bundles.

Although it is possible to see support for optical devices since v1.4.0, this support is very limited and still requires to be enhanced. Later in this document, it will be analysed how OpenFlow would be used in order to meet the requirements of the present proposal.

1.2.4. OpenFlow Match/Action Functionalities

In an SDN/OpenFlow network, OF capable devices have a flow table with flow entries as shown in the figure below. The controller updates this flow table by adding or removing flows in order to configure forwarding functionalities.

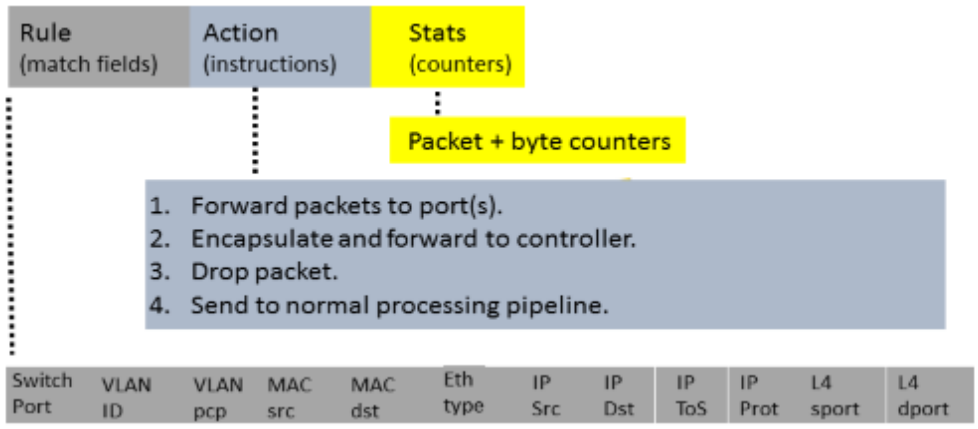

Fig. 1.7 OpenFlow Flow Table Entries

Each flow entry consists of three fields:

- **Rule:** Defines a matching condition to a specific flow.

- **Action:** Defines the action to be applied for a specific flow.

- **Counters:** Count the rule occurrence for management purposes.

After a packet arrives to an SDN device, it would be matched by a flow entry in the flow table. According to this entry, a specific action will be executed and a counter will be updated. In the case of no match, the packet would be sent to the controller for further processing.

In a flow table, priorities can also be set, where a higher number means a higher priority. If a packet matches to several flow entries, the action corresponding to the one with higher priority would be applied.

1.2.5. OpenFlow Messages

In order to operate, OpenFlow uses different types of messages for specific purposes. Messages are transmitted between the controller and the SDN device or in the opposite way depending on the case. In this document basic OpenFlow v1.0.0 messages will be analysed (see [9]), as are the ones that best accommodate to the requirements of the present proposal. These messages can be divided in three types:

1.2.5.1. Symmetric Messages

This type of messages is sent in both directions, without solicitation:

- **Hello:** Exchanged between SDN network device and controller upon connection start up.

- **Echo:** Used to indicate latency, bandwidth or aliveness of a connection, in this case an echo request must be answered by an echo reply.

1.2.5.2. Asynchronous Messages

These messages are sent from network devices to the controller without a previous solicitation from it:

- **Packet-in:** The network device sends this packet to the controller even when there is no-match at the flow table or when the packet matches an entry that explicitly specifies sending the packet to the controller.

- **Flow-removed:** Advices of a flow removal at the device's flow table when flows are deleted because of a lack of activity or a of hard timeout value that indicates time of removal.

- **Port-status:** Advices port configuration state changes. The device is expected to send these messages in front of any port change.

- **Error:** Notifies the controller about problems in the device.

[9] ONF, "OpenFlow Switch Specification", Version 1.0.0, Open Networking Foundation, (2009).

1.2.5.3. Controller-to-Switch Messages

These messages are sent by the controller. Some of them may require a response:

- **Features:** After the session establishment, the controller sends a features request message to the network device, so the device can respond with a features reply containing its supported capabilities.

- **Configuration:** Controller can set or request configuration parameters in the device that only responds to requests.

- **Modify-state:** Allow the management of states in the device by adding, deleting or modifying flows at the flow tables or by setting port properties.

- **Read-state:** Used by the controller to collect statistics from flow tables, ports and individual flow entries.

- **Send-packet:** Used to send packets out of a specific port on the device.

An example of a typical OpenFlow message exchange scenario can be found in **ANEXES [I]**, where it is possible to identify how the interaction of previously analysed types of messages allows the communication in the southbound interface.

CHAPTER 2. BENCHMARKING OF AVAILABLE SDN CONTROLLERS

This chapter characterizes the available controllers in the SDN market, evaluating provided functionalities such as supported northbound/southbound interfaces and internal controller modules. For the characterization, only benefits and drawbacks of open controllers are considered, as vendor-defined proprietary controllers are seen not suitable for the implementation of the present project.

2.1. SDN Controllers

Before analysing the possible design of the OTPNODE application that is intended to be used as an optical switching node manager, it is necessary to choose an implementation environment. Therefore, a controller must be selected, where the application will run and use the controller functionalities to interact with other applications in the northbound and to the optical node agent in the southbound.

The following table presents some of the most used SDN controllers, presenting a brief description of each one.

Table 2.1 SDN Controllers

SDN Controller	Description
Opendaylight [odl]	Industry-led collaborative open source SDN platform hosted by the Linux Foundation. www.opendaylight.org
Floodlight [floodl]	Enterprise-class, Apache-licensed, Java-based SDN Controller http://www.projectfloodlight.org/floodlight/
OpenContrail Controller [openc]	Apache 2.0-licensed project built using standards-based protocols to provide all the necessary components for an SDN controller http://www.opencontrail.org/
Ryu [ryu]	SDN framework that provides software components with well-defined API that makes it easy for developers to create new network management and control applications, with support of OpenFlow, Netconf, OF-config. http://osrg.github.io/ryu/
ONOS [onos]	New carrier-grade SDN network operating system designed for high availability, performance, scale-out and well-defined northbound and southbound abstractions and interfaces. http://onosproject.org/

2.1.1. Opendaylight

OpenDaylight [odl] is an open source platform for network programmability hosted by the Linux Foundation (see [10]) that enables SDN and NFV through a combination of components including a fully pluggable controller, interfaces, protocol plug-ins and applications; it supports networks of any size and scale.

The core part of the project, the controller, is implemented in Java and can be contained within its own Java Virtual Machine (JVM), so that it can be run on any platform that supports Java. Besides, the northbound and southbound interfaces have clearly defined and documented APIs. The combination of these elements allows vendors, service providers, customers and application developers to utilize a standards-based and widely supported SDN platform.

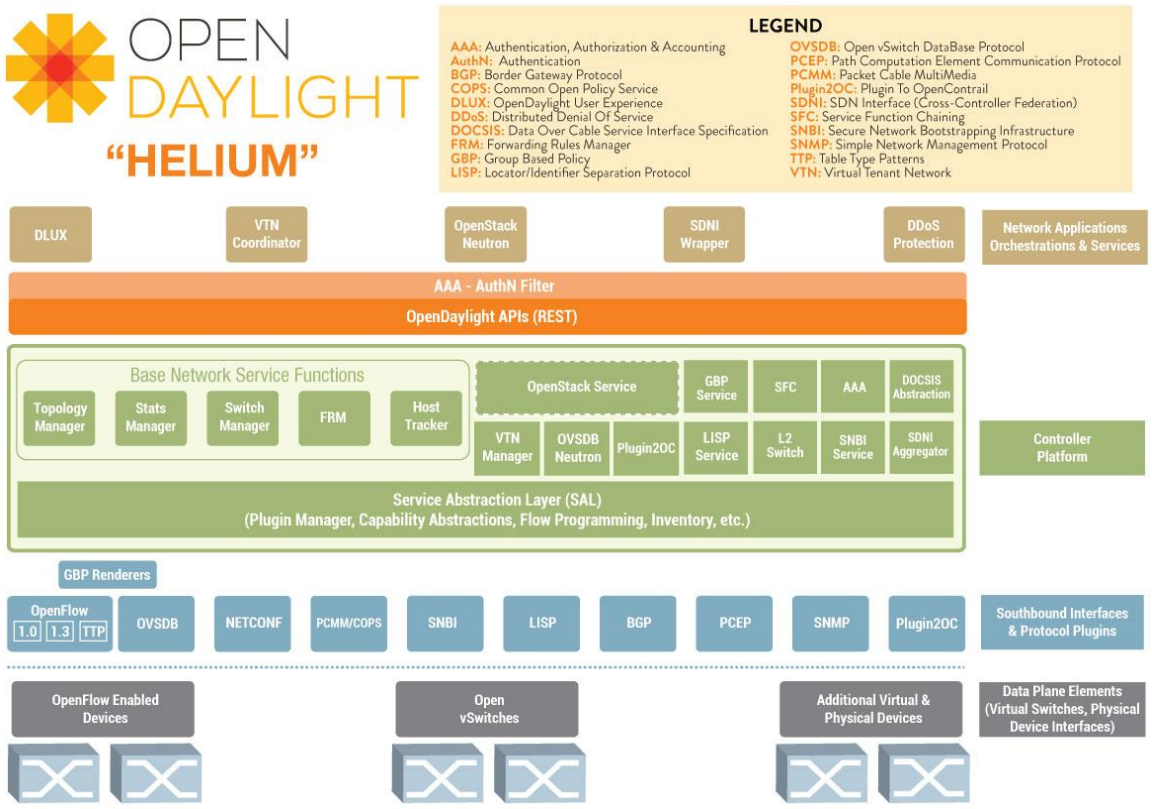

Fig. 2.1 Opendaylight Helium Architecture [10]

The figure above shows the community's second release "Helium", which comes with several additional plugins, new applications, south-bound interfaces and features, and also includes a new user interface and a much simpler and customizable installation process than the first release "Hydrogen", thanks to the use of the Apache Karaf container. Some of the features and capabilities of the "Helium" controller release are:

[10] The Opendaylight Platform <www.opendaylight.org>

- **Northbound Interfaces:** The Opendaylight northbound framework is open and flexible to either extend already existing APIs, or create new REST APIs to expose new user-defined internal network services capabilities.

- **Controller Modules:** It is composed by a collection of dynamically pluggable modules to perform a variety of network tasks and services. In this version, a new form of abstraction defined by models (the Model-Driven Abstraction Layer, MD-SAL) is presented. The MD-SAL interconnects different modules of the controller.

  The SDN controller platform offers a set of base network services such as the Topology Manager, the Statistics Manager, the Switch Manager, the Host Tracker, etc. Each of these modules exposes its own northbound APIs to let network applications and orchestration services program the network and the controller behaviour.

- **Southbound Interfaces:** Opendaylight supports multiple southbound interfaces through a heterogeneous set of dynamically loadable plugins, allowing implementing a wide set of southbound protocols, such as OpenFlow v1.0, OpenFlow v1.3, BGP Link State (BGP-LS), Open vSwitch Database (OVSDB), PCEP, NETCONF, etc.

2.1.2. Floodlight

The Floodlight Open SDN Controller (see [11]) is an enterprise-class, Apache-licensed, Java-based OpenFlow controller, which is supported by an open community and is the core part of the Big Switch Network's commercial solution.

The controller offers a module loading system that makes it simple to extend and enhance. It is easy to set up with minimal dependencies, and supports a broad range of virtual and physical OpenFlow switches. It is also able to handle mixed OpenFlow and non-OpenFlow networks and can manage multiple "islands" of OpenFlow hardware switches. Some of the features and capabilities of the Floodlight controller's most recent release are:

- **Northbound Interfaces:** The interface is based on RESTful web services. It provides capabilities for querying the network topology, setting up OF rules in devices, setting up virtual networks (VNF) and defining firewall rules. It also shows support for OpenStack cloud orchestration platform through a dedicated python-based Neutron plugin.

- **Controller Modules:** The controller is composed of a core that provides base network functions (such as Topology Manager, Link Discovery, Device Manager, Performance Monitor, among others) and a set of user-defined Java module applications for complex network services, which

[11] Floodlight OpenFlow Controller - Project Floodlight <http://www.projectfloodlight.org/floodlight/>

are interfaced to the core controller through a westbound Java API. These additional modules are loaded via a separate module system when the Floodlight controller starts, enabling the deployment of different editions of the controller.

- **Southbound Interfaces:** The southbound interface only provides support for Openflow v1.0 and Openflow v1.3. No plans are envisaged in the short term to extend the support to other protocols.

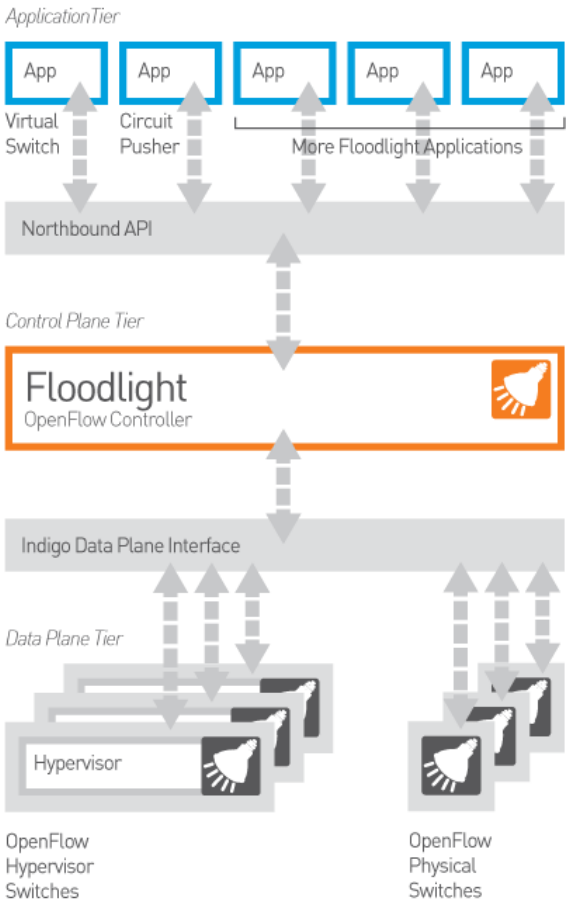

Fig. 2.2 Floodlight v1.0 Architecture [11]

The figure above shows the architecture of Floodlight v1.0, the latest version of the controller, released on December 2014.

2.1.3. Open Contrail

OpenContrail System (see [12]) is an extensible platform for Software Defined Networking (SDN). The system is an open source Apache 2.0 licensed project that is built around standards-based protocols and provides flexible, reliable and highly scalable solutions that can be used for multiple networking use cases

[12] OpenContrail <http://www.opencontrail.org/>

such as Cloud Networking and Network Function Virtualization (NFV) in Service Provider Networks.

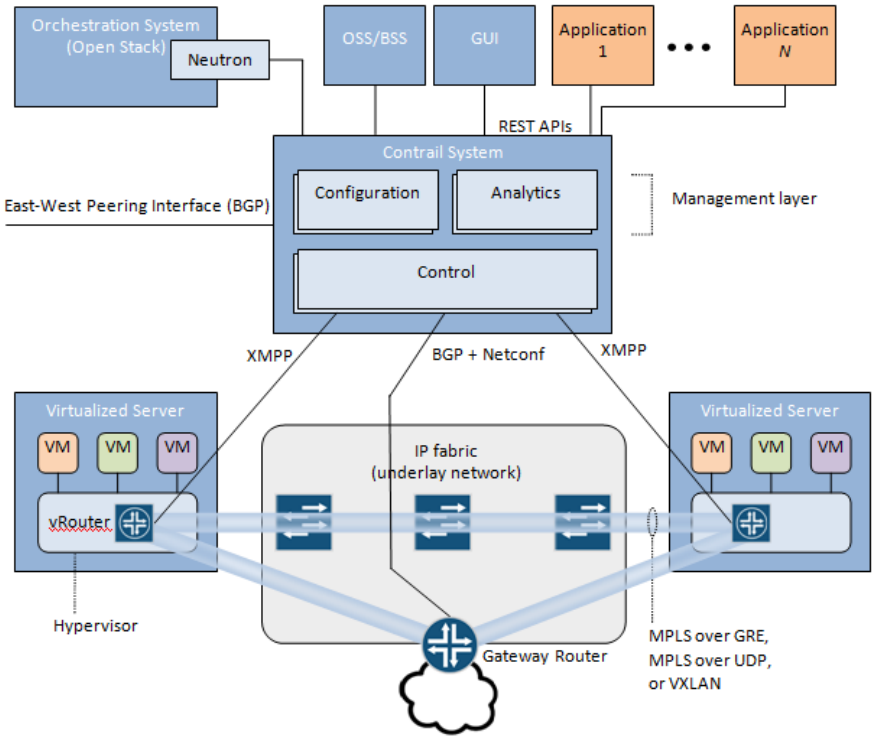

Fig. 2.3 OpenContrail System Architecture [12]

The OpenContrail System, as seen in the figure above, consists of two parts: a logically centralized but physically distributed controller and a set of virtual routers (vRouters) that serve as software forwarding elements implemented in the hypervisors of general purpose virtualized servers. In this case some of the features and capabilities of the OpenContrail controller are:

- **Northbound Interfaces:** The configuration nodes in the OpenContrail Controller provide a northbound REST API to the provisioning or orchestration system. These northbound REST APIs are automatically generated from the formal high-level data model allowing every service to be provisioned and controlled through them.

- **Controller Modules:** The controller architecture consists of three types of nodes: Configuration nodes, Control nodes and Analytics nodes. Configuration nodes are responsible for translating the high-level data model into a lower level form suitable for interacting with network elements. Control nodes are responsible for propagating this low level state to and from network elements and peer systems in an eventually consistent way. Finally, Analytics nodes are responsible for capturing real-time data from network elements, abstracting it and presenting it in a form suitable for applications to consume.

- **Southbound Interfaces:** The control nodes are responsible for realizing the desired state of the network as described by the low-level technology data model using a combination of southbound protocols. In the system, these south-bound protocols include Extensible Messaging and Presence Protocol (XMPP) to control the OpenContrail vRouters as well as a combination of the Border Gateway Protocol (BGP) and the Network Configuration (NetConf) protocols to control physical routers.

2.1.4. Ryu

Ryu is a component-based SDN framework (see [13]) composed of open source code available under the Apache 2.0 license. It is fully written in phyton and provides software components with a well-defined API that makes it easy for developers to create new network management and control applications. Some of the features and capabilities of the Ryu controller are:

- **Northbound Interfaces:** Ryu offers a RESTful API with JSON data format for access to information and configuration, where every module can offer its own REST API. Ryu has also an OpenStack Neutron plug-in that supports both GRE based overlay and VLAN configurations.

- **Controller Modules:** Existing components inside the controller include southbound protocols support, event management, messaging, in-memory state management, application management, infrastructure services and a series of reusable libraries. Additionally, applications like layer 2 switch, firewall, IDS (Snort), GRE tunnel abstractions, VRRP, as well as services, like topology discovery and statistics, are available.

- **Southbound Interfaces:** Ryu supports various protocols for managing network devices, such as OpenFlow, Netconf, OF-config, etc. About OpenFlow, Ryu supports versions 1.0, 1.2, 1.3, and 1.4.

Figure below shows the main Ryu framework architecture:

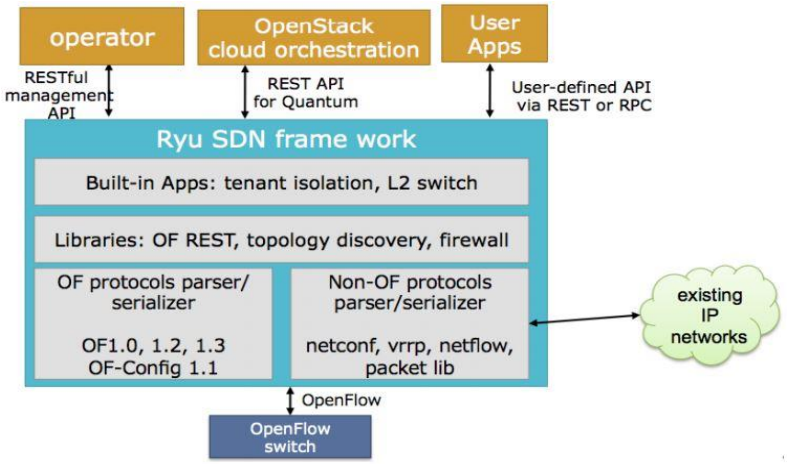

Fig. 2.4 Ryu Architecture [13]

[13] Ryu SDN Framework <http://osrg.github.io/ryu/>

2.1.5. ONOS

The Open Network Operating System (ONOS) is the first open source SDN network operating system targeted specifically at the Service Provider and mission critical networks (see [14]). ONOS was built to provide the high availability, scale-out, and performance that these networks demand. In addition, ONOS has created useful northbound abstractions and APIs to enable easier application development and southbound abstractions and interfaces to allow for control of OpenFlow-ready and legacy devices.

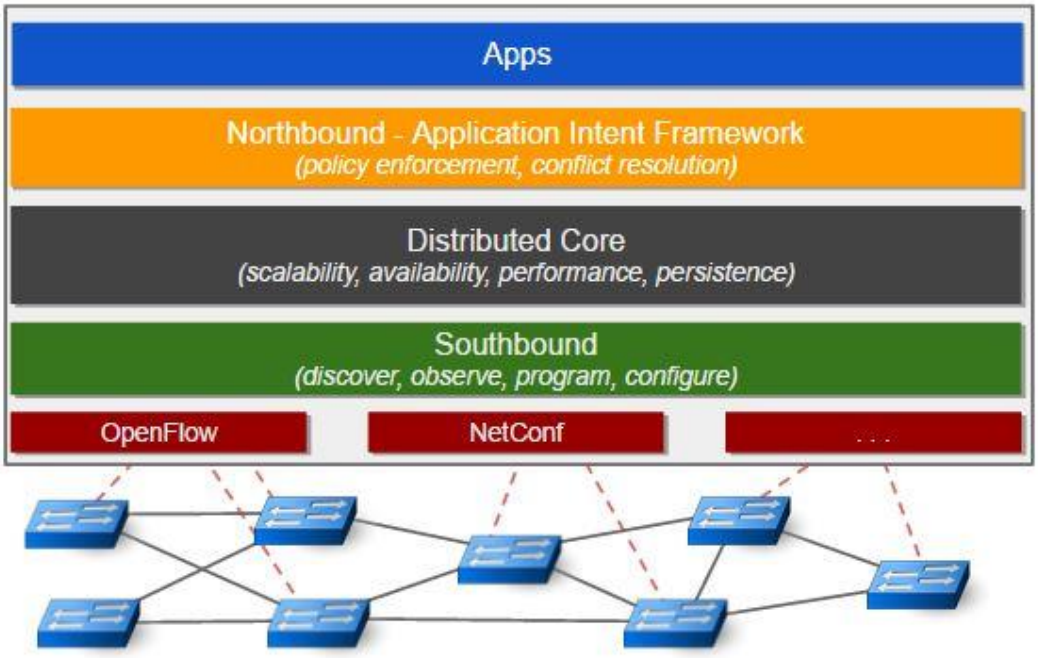

Fig. 2.5 ONOS Architecture [14]

The figure above shows the basic architecture of “Blackbird”, the second release of ONOS, where it is possible to analyse some of its features and capabilities such as:

- **Northbound Interfaces:** The northbound provides two important abstractions: the Intent Framework that allows an application to request a service from the network without having to know details of how the service will be performed, and the Global Network View that provides a view of the Network, where an application can program this network view through APIs.

- **Controller Modules:** ONOS modules run over a distributed core showing well-defined relationships, a basis for customization and the avoidance of cyclic dependencies between them.

[14] ONOS Project <http://onosproject.org/>

- **Southbound Interfaces:** ONOS and its southbound abstraction allow plugins for various southbound protocols and devices, where a plugin maps or translates generic network element description or operation on the device to the specific and vice versa. The southbound also enables ONOS to control or manage multiple diverse devices, even if they use different protocols (OpenFlow, NetConf, OVSDB, etc.).

2.2. Qualitative Comparison of Controllers

After analysing each controller individually, it is possible to compare their main features. Table 2.2 shows the pros and cons of the aforementioned controllers. The most appropriated one needs to be selected according to the project's requirements.

For this project the main requirements are:

- The need of an Application tutorial and the necessary well-documented resources to implement it in the selected controller.
- Open southbound interface with support of OpenFlow v1.0 protocol.
- Open northbound interface with support of REST APIs.
- High-level abstraction at the controller in order to allow a flexible integration of the application with other modules, services.
- Large open source community for feedback request.

According to the requirements and to the comparison show in the table, the Opendaylight (ODL) controller has been chosen as the reference SDN controller for the project. The controller shows a modular and extensible architecture that offers many already deployed services, applications and APIs. It also provides a well-defined abstraction through the use of YANG models at the controller level and a wide support from both scientific and industry communities that encourages new projects to implement innovative solutions through SDN.

Further considerations of why ODL controller is the most appropriated one for the implementation of the OPTNODE application will be seen in CHAPTERS 3 and 4, where a deeper analysis of the project's requirements will be performed in order to explain, for example, the reason of choosing OpenFlow v1.0 and REST APIs as interfaces for the application.

Table 2.2 Qualitative Comparison of SDN Controllers

Features	Opendaylight	Floodlight	Open Contrail	Ryu	ONOS
External Apps Support	Via northbound REST APIs.	Supported through REST APIs.	Northbound REST APIs to configure the system or collect operational data.	RESTful API with JSON data format for access to information and configuration.	Through the Intent Framework and Global Network View northbound abstractions.
Resource Abstraction	Model Driven Service Abstraction Layer (MD-SAL).	Supports abstractions for various network resources.	Declarative high-level service data models to describe services.	Application management, infrastructure services and a series of reusable libraries.	Through the deployment of a distributed core.
Application Tutorial	Yes, very detailed with many examples.	Yes	No	Yes	Yes, limited information
Popularity	Very high. Industry-led project with high interest and expected impact in the SDN/NFV community	Medium-Low. Big Switch Networks-led project.	Medium-High. Juniper-led project.	Medium. NTT Labs-led project.	Medium-High. Open Networking Lab-led new project.
OpenFlow Support	Yes, v.1.0 and v1.3.	Yes, v1.0 and v1.3.	No	Yes, all versions.	Yes, v.1.0 and v1.3.
Other SB Protocols	SNMP, NETCONF, PCEP, BGP-LS, OVSDB, LISP, among others.	No, no plans to extend support.	XMPP, BGP, NETCONF.	OF-CONFIG, OVSDB, NETCONF.	NETCONF, OVSDB.

CHAPTER 3. OPENDAYLIGHT INFRASTRUCTURE & ENVIRONMENT

3.1. ODL Infrastructure

Following the selection of ODL as the reference controller for the project, this section deeply analyses how the controller works and which of its components would be used in order to create, integrate and deploy the OPTNODE application.

3.1.1. MD-SAL

The Model-driven Service Abstraction Layer (MD-SAL) presents a way to unify both the northbound and the southbound APIs and the data structures used in various services and components of the SDN Controller. In order to describe the data structure provided by the controller components, MD-SAL uses the YANG data modelling language for service and data abstractions. Some of the benefits of the proposed language are:

- Modelling the structure of XML data and functionality provided by controller components.
- Defining semantic elements and their relationships.
- Modelling all the components as a single system.

Utilizing a schema language simplifies the development of controller components and applications. In this case, a developer of a module that provides some functionality (a service, data, function or procedure) can define a schema and thus create simpler, statically typed APIs for the provided functionality. The SAL architecture can be seen in the figure below.

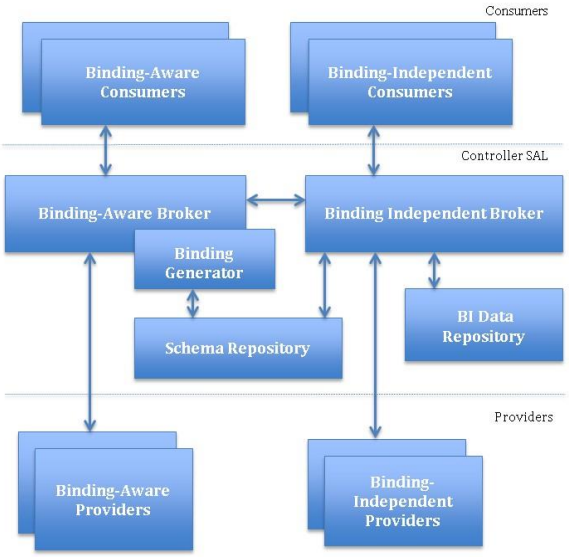

Fig. 3.1 SAL Architecture [15]

The SAL system shows a provider-consumer model conformed by a series of subsystems (see [15]), where each one defines a specific functionality:

- **Binding-Independent Broker:** The core component of the MD-SAL. It routes RPCs, notifications and data changes between various Providers and Consumers.

- **Binding-Aware Broker:** Provides programmatic APIs and Java language support to both Consumers and Providers. It works as a proxy built on top of the Binding Independent Broker that simplifies access to data and services provided by Binding Providers.

- **BI Data Repository:** Is a binding-independent infrastructure component of SAL that is responsible for the storage of configuration and transient data.

- **Binding Schema Repository:** An infrastructure component, responsible for storing specifications of YANG–Java relationships and mapping between language-binding APIs to binding-independent API calls.

- **Binding Generator:** A SAL infrastructure component, which generates implementations of binding interfaces and data mappers to the binding-independent format.

The providers are defined as components that expose its functionality to applications and to other providers (plugins) through its northbound API. A provider can be a consumer of other providers, as well. There are two types of providers:

- **Binding Independent Providers**: their functionality is exposed in the binding-independent Data DOM format.

- **Binding Aware Providers:** their functionality is exposed in a format compiled against one or more generated binding interfaces

The consumers are defined as components that consume a functionality provided by one or more providers. There are two types:

- **Binding Independent Consumers:** The functionality is consumed in the binding-independent Data DOM format

- **Binding Aware Consumers:** The functionality is consumed via one or more generated binding interfaces

In this project, the OPTNODE application is defined as a provider of functionalities that will expose specific information and operations of the optical node. The application will also be required to consume functionalities from other applications and to interact with them inside the controller through the MD-SAL.

---

[15] SAL Architecture Overview <wiki.opendaylight.org/view/OpenDaylight_Controller:_SAL_Architecture_Overview>

3.1.2. OpenFlow Plugin

The OpenFlow Plugin project presents a plugin (see [16]) to support implementations of the OpenFlow specification aiming to handle versions 1.0 and 1.3.x. It also can be extended to add support for subsequent OpenFlow specifications. The plugin is based on the MD-SAL architecture. The next figure shows the basic architecture of the OpenFlow Plugin.

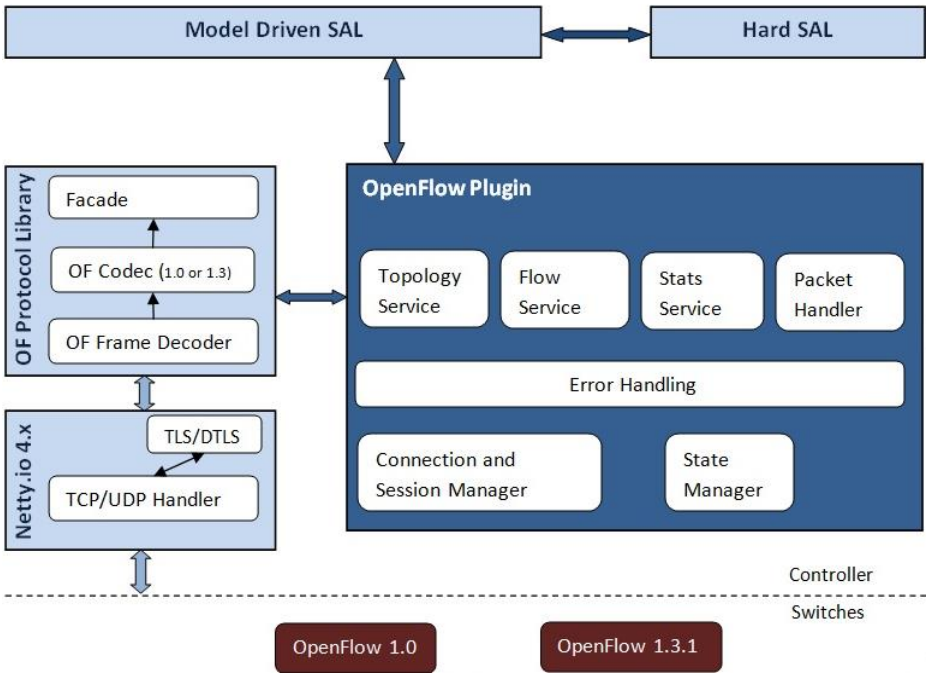

Fig. 3.2 OpenFlow Plugin Architecture [16]

The plugin shows a direct integration with OF Protocol and Netty.io libraries in order to provide a southbound plugin able to handle an OpenFlow connection while offering services such as: Topology Service, Flow Service, Stats Service and Packet Handler. It also provides message and event handling with propagation to upper layers through the MD-SAL.

For the OPTNODE application, interaction with the OpenFlow Plugin will be crucial in order to listen for an optical node's new connection to the controller and to retrieve data notifications published to the MD-SAL by the OF Plugin. Later chapters of the present document will show how this interaction is managed by the application and how optical node's capabilities are retrieved.

3.1.3. YANG Tools

YANG Tools is a infrastructure project aiming to develop necessary tooling and libraries providing support of NETCONF and YANG for Java (JVM-language

[16] Plugin Design <https://wiki.opendaylight.org/view/OpenDaylight_OpenFlow_Plugin:Overview_Architecture>

based) projects and applications, such as MD-SAL for the ODL Controller (which uses YANG as it's modelling language) and Netconf / OFConfig plugins. In Opendaylight, YANG Tools is used to generate or map Java classes from Yang models. In the OPTNODE application, it will automate the creation of SAL APIs containing the application provided data defined in the YANG data model. Further information about how Yang Tools is involved in the creation of MD-SAL plugins can be found in **ANEXES [II]**.

3.1.4. REST APIs & RESTCONF

REST allows for a minimum amount of data to be passed using the same well-established mechanisms that define the web, it works by exposing resources to a program via a URL. The program can access that URL and receive data about the resource. Well-designed RESTful APIs include additional links the program can follow to request related information.

The REST APIs in ODL are also derived from models. It is not necessary for the designer to write any code for them. The controller implements the RESTCONF protocol, which defines access to yang-formatted data through REST.

After defining the service in a model, the designer just needs to expose that model to the SAL. REST access to modelled data will then be provided by the SAL infrastructure. However, it is also possible to create a specific REST API (for example, to be compliant with an existing API).

RESTCONF then, allows access to data stores in the controller, where request and response data can be in XML or JSON format. There are two existent data stores inside the controller:

- **CONFIGURATION** – Usually contains data inserted via controller or data that has both read and write permissions.

- **OPERATIONAL** – Usually contains data obtained from the network or data with read-only permission.

In the case of the OPTNODE application, both data stores will be used in order to expose data of the optical node, which will be available through REST so other applications will have access to it by means of a REST client.

3.1.5. RPCs

In YANG, Remote Procedure Calls (RPCs) are used to model any procedure call implemented by a Provider, which exposes its functionality to Consumers. In MD-SAL terminology, the term 'RPC' is used to define the input and output for a function that is to be provided by a Provider, and adapted by the MD-SAL.

In the context of the MD-SAL, there are three types of RPCs (RPC services) defined in various API types:

- **Java Generated APIs:** Where Providers expose their implementation of the service by registering their implementation to RpcProviderRegistry and Consumers get it from RpcConsumerRegistry.

- **DOM APIs:** RPCs are identified by QName. Providers expose their implementation of RPC identified by QName registering their RpcImplementation to RpcProvisionRegistry and Consumers get it from RpcConsumerRegistry.

- **REST APIs:** RPCs are identified by the model name and their name. Consumers invoke RPCs by invoking POST operation to /restconf/operations/model-name:rpc-name.

In the case of the OPTNODE application, REST APIs will be exposed, so consumers above or inside the controller will be able to access to the provided operations. Further analysis on these operations will be shown in Chapter 4.

3.1.6. Notifications

In YANG, Notifications represent asynchronous events, published by providers for listeners. There exist RPCs in various API types:

- **Java Generated APIs:** For each model, there is a Listener interface and transfer object for each notification. Providers publish notifications by invoking the publish method on NotificationPublishService. To receive notifications, consumers register their implementation of Listener to NotificationBrokerService.

- **DOM APIs:** Notifications are represented only by XML Payload. Providers publish notifications by invoking the publish method on NotificationPublishService. To receive notifications, consumers register their implementation of Listener to NotificationBrokerService.

- **REST APIs:** Notifications are currently not supported.

In Opendaylight, a provider may use specific types of Notifications in order to interact with other applications (consumers) or may also create new types of notifications for a more particular use.

For this project, and as it will be seen in the next chapter, the OPTNODE application will subscribe/listen to specific types of Notifications published by the OpenFlow Plugin in order to retrieve the optical node’s information.

3.2. ODL Environment

This section presents the description of several components that work along with the controller in order to provide the desired functionalities and to simplify deployment of applications.

3.2.1. OSGi & Apache Karaf

The Open Services Gateway Initiative (OSGi) defines an architecture for developing and deploying modular applications (see [17]). The OSGi specification defines a set of services that an OSGi container must implement and a contract between the container and an application. Developing on the OSGi platform requires first building an application using OSGi APIs and then deploying it in an OSGi container. OSGi offers the following advantages:

- Ability to install, uninstall, start, and stop different modules of an application dynamically without restarting the container.
- An application can have more than one version of a particular module running at the same time.
- OSGi provides very good infrastructure for developing service-oriented applications, as well as embedded, mobile, and rich internet apps.

Any framework that implements the OSGi standard provides an environment for the modularization of applications into smaller bundles. The OSGi layered model can be seen in the next figure where many layers are defined:

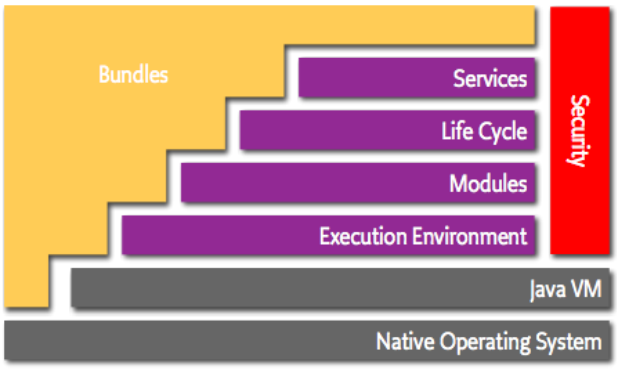

Fig. 3.3 OSGi Service Gateway Architecture [17]

- **Bundles:** Bundles are the OSGi components made by the developers.
- **Services:** The services layer connects bundles in a dynamic way by offering a publish-find-bind model for plain old Java objects.

[17] OSGi Alliance - The OSGi Architecture <www.osgi.org/Technology/WhatIsOSGi>

- **Life-Cycle:** The API to install, start, stop, update, and uninstall bundles.
- **Modules:** Layer that defines how a bundle can import and export code.
- **Security:** The layer that handles the security aspects.
- **Execution Environment:** Defines what methods and classes are available in a specific platform.

An OSGi application is defined as a set of OSGi bundles. An OSGi bundle is a regular jar file, with additional metadata in the jar MANIFEST. In OSGi, a bundle can depend on other bundles. Therefore, to deploy an OSGi application, most of the time, it is first necessary to deploy other bundles required by the application.

Apache Karaf (see [18]) is a small OSGi based runtime that provides a lightweight container onto which various OSGi components and applications can be deployed. With Karaf, users can pick the high-level features they need and build their own versions of, for example, the Opendaylight controller.

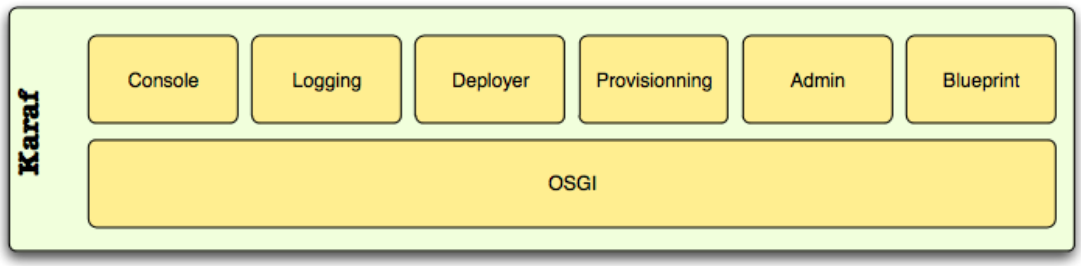

Fig. 3.4 The Apache Karaf Container [18]

In Apache Karaf, the application provisioning is simple and flexible by defining Apache Karaf "features", where a feature contains all the requirements (bundles and configurations) to be deployed for an application. A feature describes an application as:

- a name
- a version
- an optional description (eventually with a long description)
- a set of bundles
- optionally a set configurations or configuration files
- optionally a set of dependency features

When a feature is installed, Apache Karaf installs all resources described in the feature, therefore automatically resolving and installing all bundles, configurations, and dependency features described in the feature.

[18] Karaf 4.0.1 - The Apache Software Foundation <karaf.apache.org/>

OpenDaylight has been an OSGi based project from the beginning and now, with Helium distribution, the idea is to improve its base environment experience via Karaf. Specifically, some of the key reasons for the distribution are: ease of deployment, friendly console CLI, superior PAX exam integration and a supportive community.

The ODL Apache Karaf distribution is a customized Karaf distribution on top of which key ODL components are deployed. The base Karaf container has been configured to support OpenDaylight's specific needs, including Equinox core, key component features available by default, custom commands, and branding; thus providing a lightweight container where different apps can run and the ability to plug and play these apps.

ODL “Helium” distribution’s currently supported ports for Apache Karaf are:

- **RESTCONF:** PORT 8181 (Yang, Dlux, MD-SAL API's)
- **REST:** PORT 8080 (AD-SAL API)

3.2.2. Maven & Archetypes

Apache Maven is a software project management and comprehension tool that is based on the concept of a project object model (POM), which is the fundamental unit of the entire Maven system.

Maven simplifies and standardizes the project build process. It handles compilation, distribution, documentation, team collaboration and other tasks seamlessly. Maven increases reusability and takes care of most of build related tasks by addressing two aspects of building software: First, it describes how software is built, and second, it describes its dependencies.

Maven primary goal is to provide to the developer:

- A comprehensive model for projects which is reusable, maintainable, and easier to comprehend.
- Plugins or tools that interact with this declarative model.

The POM or pom.xml file is the core of a project's configuration in Maven. It is a single configuration file that contains the majority of information required to build a project in a specific way.

In order to create a simple project using maven, it is possible to use the Maven Archetype Plugin. This plugin allows the creation of an "archetype", which in maven, is the general skeleton structure (or template) of a project.

In the case of the present project, an archetype named “opendaylight-startup-archetype” will be used to setup the skeleton of the project, more details on how this setup is performed can be found in **ANEXES [III]**.

3.2.3. Snapshots

A snapshot represents a frozen image of a volume, where the source of a snapshot is called an "original". When a snapshot is created, it is exactly like the original at that point in time. As changes are made to the original, the snapshot remains the same and looks exactly like the original at the time the snapshot was created.

Snapshotting allows keeping a volume online while a backup is created. This method is much more convenient than a data backup where a volume must be taken offline to perform a consistent backup. When snapshotting, a snapshot of the volume is created and the backup is taken from the snapshot, while the original remains in active use.

A snapshot version in Maven represents a version that has not been released. The idea is that before a 1.0 release is done, there commonly exists a 1.0-SNAPSHOT. That version could be considered basically as a "1.0 under development". The difference between a "real" version and a snapshot version is that snapshots might get updates, so these can vary in time.

3.2.4. Yang Language & Models

YANG is a data modelling language used to model configuration and state data manipulated by the NETCONF protocol. The data modelling language can be used to model configuration data, state data of network elements, RPCs and notifications. It represents data structures in an XML tree format. Some of the capabilities provided by YANG are:

- Human readable, easy to learn representation.
- Hierarchical configuration data models.
- Reusable types and groupings (structured types).
- Extensibility through augmentation mechanisms.
- Supports the definition of operations (RPCs).
- Formal constraints for configuration validation.
- Data modularity through modules and sub modules.
- Versioning rules and development support.

YANG defines the data definition model, which is used by MD-SAL to provide a variety of SAL functions required for adaptation between Providers and Consumers. The implementation of these functions requires the use of two types of SAL Plugin APIs, DOM and Binding Aware. Data models in both API types can be modelled by YANG.

- **Binding Aware APIs:** This format is a specific YANG to Java language binding, which specifies how Java Data Transfer Objects (DTOs) and APIs are generated from YANG model. The API definition for these

DTOs, interfaces for invoking / implementing RPCs, interfaces containing Notification call-backs are generated at compile time.

- **DOM APIs:** This format is a representation of YANG trees. It is suitable for generic components, such as the data store, the NETCONF Connector, RESTCONF, which can derive behaviour from a YANG model itself.

The data handling functionality is divided into two distinct brokers: a **Binding-Independent DOM Broker** that interprets YANG models at runtime and is the core component of the MD-SAL runtime, and a **Binding-Aware Broker** that exposes Java APIs for plugins using binding-aware representation of data (Java DTOs).

The next figure shows the basic data building blocks used to define a YANG model (see [19]). In this case the example includes only data blocks for configuration and state data.

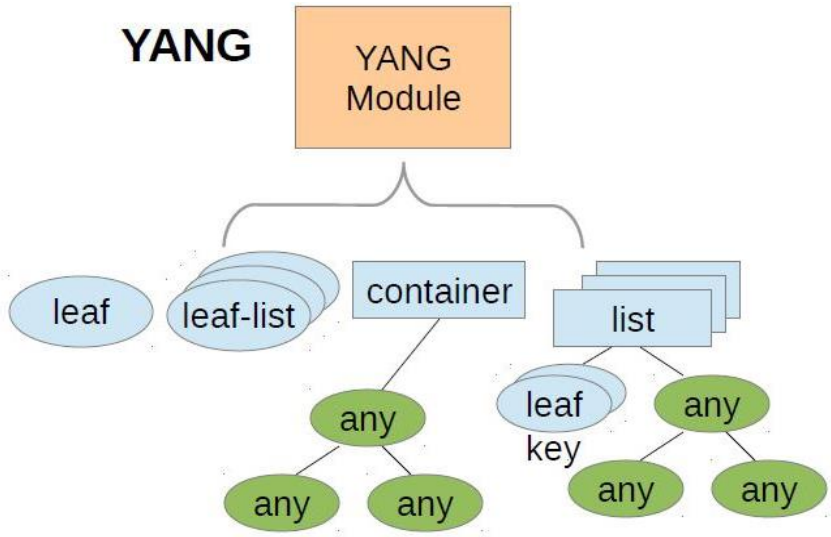

Fig. 3.5 Basic YANG model [19]

In the OPTNODE application, methods will be created from a base yang model, defined in the API folder after the generation of the project by using the *opendaylight-startup-archetype*. The model will allow the creation of state and configuration data for the optical node as well as RPCs to expose the information and provide specific functions to other external applications.

The YANG model implemented for the OPTNODE application will use different types of data building blocks as the ones seen in the figure above. Detailed description of the types and characteristics of data blocks can be found in **ANEXES [IV]**.

[19] Andy, B, "A Guide to NETCONF for SNMP Developers", IEEE 802 Plenary, v0.6,17-24 (2014).

CHAPTER 4. OPTNODE PROJECT DESIGN

This chapter exposes the main scenario for project implementation, as well as its scope and requirements. It also provides the general design and expected behaviour of the MD-SAL application and the OpenFlow Agent.

4.1. Scenario & Project Requirements

As introduced in early chapters, the main goal of the project is to provide management of non-OpenFlow capable optical switching nodes. In order to achieve this it is first necessary to establish a specific scenario where the design and implementation of the project can be performed. The next figure describes the proposed scenario for the project:

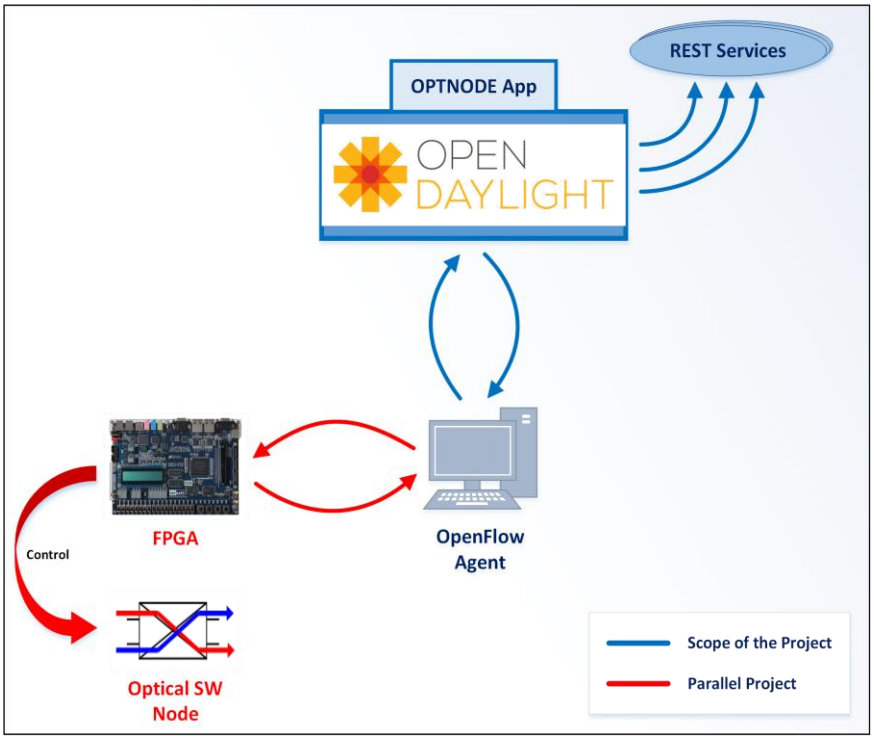

Fig. 4.1 Proposed General Scenario

As described in the figure above, the scenario includes the development of the control of optical switching nodes through an FPGA. It is worth to mention that such FPGA-based controller is being developed in another Master Thesis. The purpose of such project is to implement the communication between the FPGA and an Agent in order to transmit the node's capabilities and receive configuration data by means of an open or proprietary protocol.

Consequently, the project scope is focused on determining how the information once received from the FPGA at the agent, will be sent to the ODL controller by means of the OpenFlow protocol, and then how this information will be retrieved by the OPTNODE application in order to be exposed through REST services.

The connection between the ODL Controller and Agent has been selected to be managed by the OpenFlow v1.0 protocol, as this connection is intended to be handled as simple as possible in order to transfer data inside specific OF messages. This version of the protocol is natively supported by the controller and it is relatively simple to be implemented at the Agent by means of available OpenFlow libraries. It was also taken in consideration that OpenFlow is one of the major promoters for the standardization of SDN.

Summarizing the concepts just reviewed over the proposed scenario, the project will be required to accomplish the following:

- Design and implement an application running on top of Opendaylight's MD-SAL that will be able to interact with other apps inside the controller and expose functionalities to external apps through REST, allowing the management and request of specific information of the optical node.

- Design and implement an Agent capable of sending information to the Opendaylight controller through the use of the OpenFlow protocol, finding a way to parse the node's information inside OpenFlow messages so it can be received by the MD-SAL application.

- Integrate the created MD-SAL application with a current ODL distribution in order to test the provided RPC services on the scenario, considering different stages of integration.

4.2. OPTNODE MD-SAL App Design

The design of the OPTNODE application will be based on the requirements seen for the project, where the application must be able to:

- Interact with other apps through the MD-SAL.

- Parse information sent by the Agent to the controller.

- Use MD-SAL data stores to store operational and configuration data.

- Expose operational data through REST RPCs.

- Provide configuration functionalities via REST RPCs.

- Send configuration data back to the Agent.

In order to accomplish this behaviour, the tools seen in chapter 3 were used at the design stage, such as the base archetype to generate the application skeleton, a reference YANG model, the use of YANG Tools and Maven, the Java-based implementation code and the integration of generated bundles/features in ODL's Apache Karaf distribution.

Since the use of the "*opendaylight-startup-archetype*" was already covered in the previous chapter and can be found in **ANEXES [III]**, the following stage in the design of the app will be the construction of a reference YANG model.

4.2.1. The OPTNODE YANG Model

The usage of the YANG language to construct a reference YANG model will be the basis of the application design, where auto-generated classes, bundles and features will depend on the structure of this model.

The next figure shows the basic tree structure for the application's model, considering only in this case configuration and state data, which is dependent of a main optnode-properties container created under the OPTNODE module.

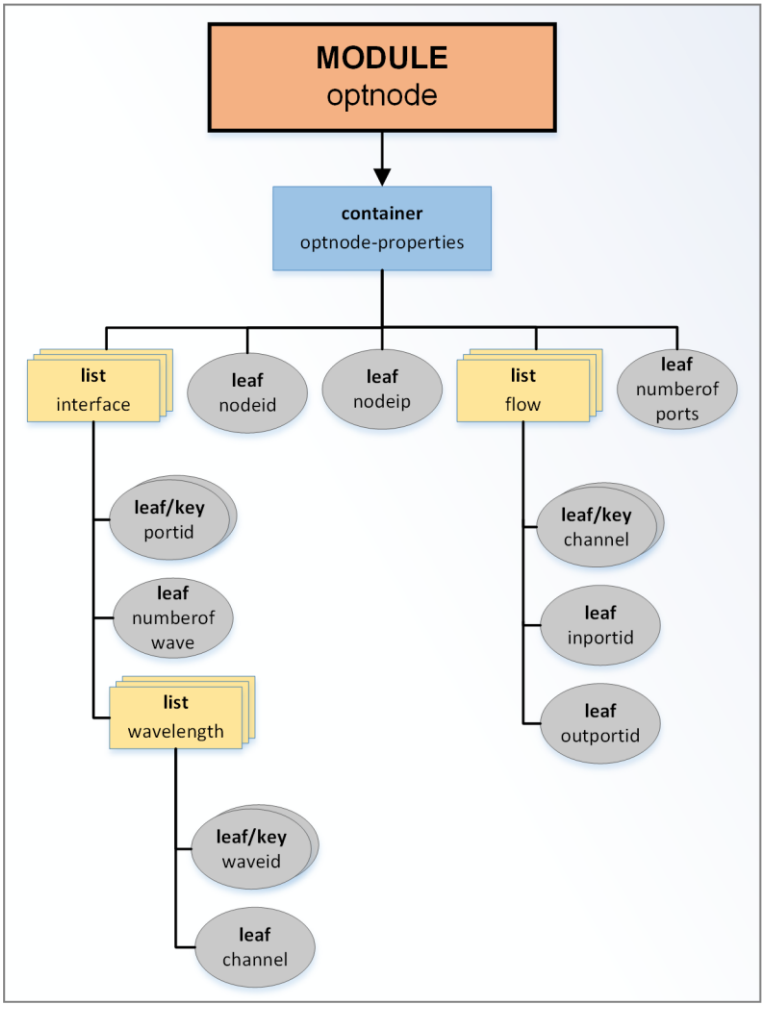

Fig. 4.2 OPTNODE YANG Model-Container

State data represented in this part of the model contains:

- **leaf - nodeid:** One simple value of type String, representing the ID of the connected optical node.

- **leaf - nodeip:** One simple value of type String, representing the IP of the connected optical node.

- **leaf - numberofports:** One simple value of type int32, representing the total number of ports of the connected node.

- **list - interface:** A sequence of list entries, each one representing an interface of the connected node. The list contains:

    - **leaf/key - portid:** A unique identifier value of type uint32, representing the ID of an specific port.

    - **leaf - numberofwave:** One simple value of type int32, representing the number of supported wavelengths in the port.

    - **list - wavelength:** A sequence of list entries, each one representing a wavelength of the interface. The list contains:

        - **leaf/key - waveid:** A unique identifier value of type int32, representing the ID of an specific wavelength.

        - **leaf - channel:** One simple value of type String, representing the supported channel.

Configuration data represented in this part of the model contains:

- **list - flow:** A sequence of list entries, each one representing a flow to be programed in the node. The list contains:

    - **leaf/key - channel:** A unique identifier value of type String, representing the channel to be used for a cross connection at the optical switching node.

    - **leaf - inportid:** One simple value of type int32, representing the input port to be used for the cross connection.

    - **leaf - outportid:** One simple value of type int32, representing the output port to be used for the cross connection.

After defining the configuration and state data, it is necessary to model the RPCs that will expose the data information and provide configuration RPCs to other applications (could be external or running inside the controller).

The next figure shows the second part of the same reference OPTNODE YANG model, considering in this case the definition of the RPC operations which is dependent directly from the OPTNODE module.

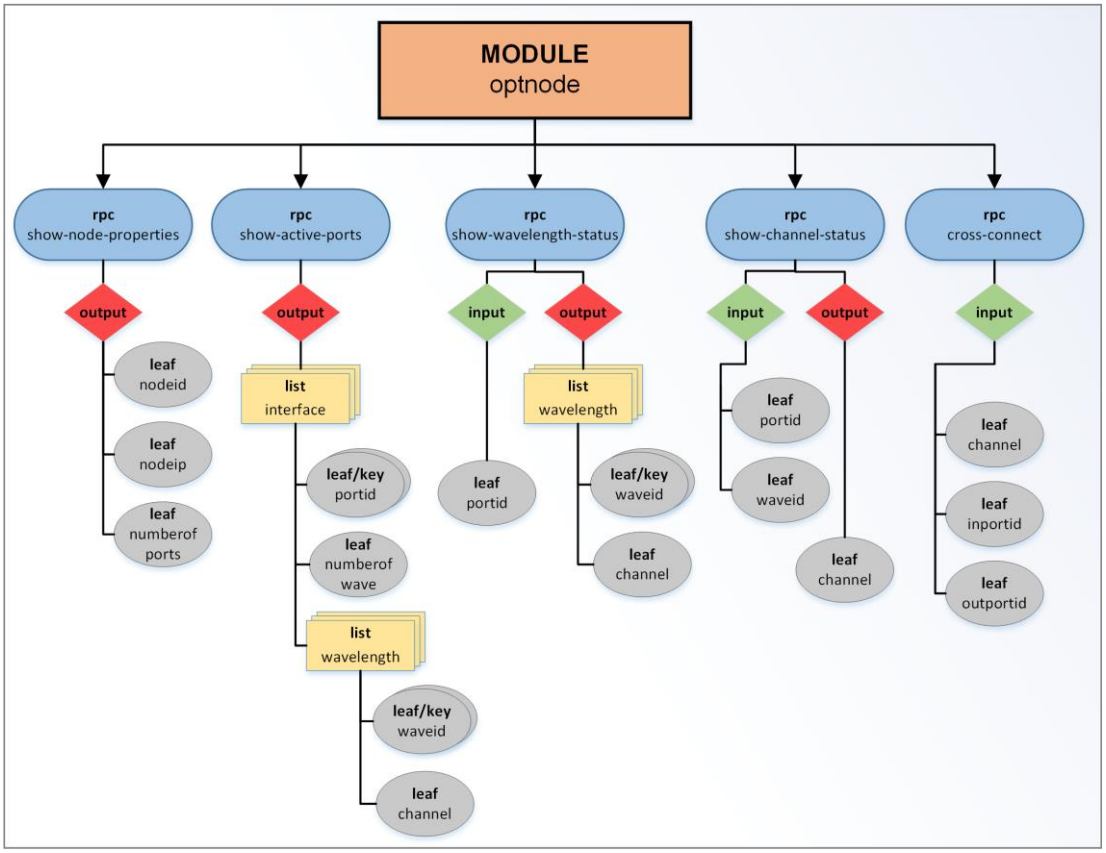

Fig. 4.3 OPTNODE YANG Model-RPCs

As seen in the figure above, five operations are defined, where each one provides a specific function and may require/provide input or output data:

- **RPC show-node-properties:** Provides state information regarding the optical node's ID, IP and number of ports.
- **RPC show-active-ports:** Provides state information regarding the available ports at the optical node their correspondent capabilities.
- **RPC show-wavelength-status:** Requires as an input the ID of a specific port, provides state information of the supported wavelengths of this port.
- **RPC show-channel-status:** Requires as an input the ID of a specific wavelength/port, provides state information of the supported channel of this wavelength/port.
- **RPC cross-connect:** Requires as an input an specific channel and the ID of both desired input and output port in order to generate configuration data that will be sent to the optical node by the application.

The main optnode.yang file used for the project can be found in **ANEXES [V]**, where the same structure analysed in this section can be observed after its translation to the YANG language.

4.2.2. Generated JAVA Classes

After defining the OPTNODE YANG model, it must be placed inside the /api folder of the structure generated by using the archetype. Then it is possible to run the *mvn clean generate-sources* command over this folder in order to compile it. By doing this, YANG Tools will identify the .yang file allocated inside the /api folder and will auto-generate JAVA classes based on this model.

The next figure shows the generated packets resulting of running YANG Tools over the OPTNODE .yang model.

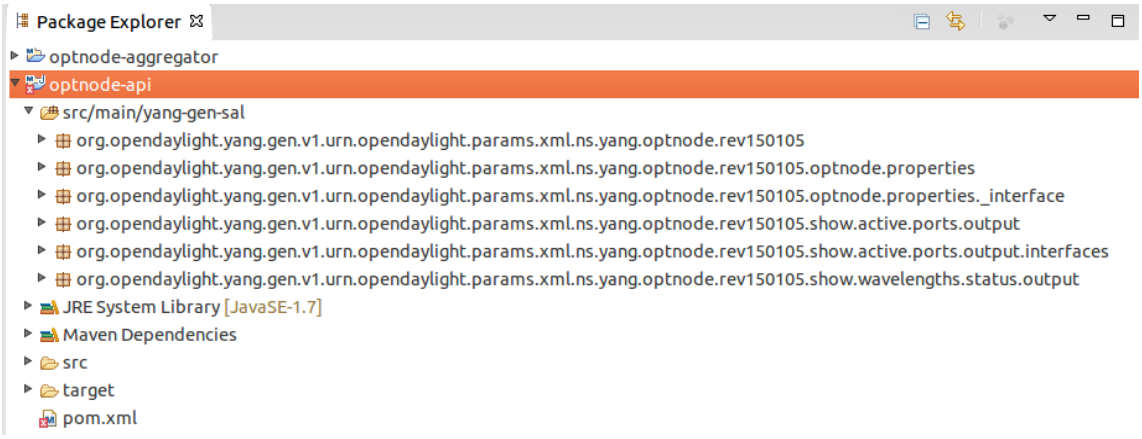

Fig. 4.4 YANG Tools Generated Packages

Inside each package it is possible to identify the auto-generated JAVA classes. In the example of the next figure, the classes found in the package "optnode.rev150105" represent input/output interfaces of each RPC, containing also a Builder helper class for each input/output interface.

Fig. 4.5 YANG Tools Generated Classes

Other important identifiable classes found in this package are:

OptondeProperties: An interface that represents the optnode-properties container with methods to obtain the leaf node data.

OptnodePropertiesBuilder: A helper class that builds immutable Data Transfer Objects (DTOs) of type optnode-properties.

OptnodeData: an interface that represents the top-level optnode module with one method getOptnode() that returns the singleton optnode instance.

$YangModelBindingProvider, $YangModuleInfoImpl: Classes which are used internally by MD-SAL to wire the optnode module for use.

These auto-generated classes will be used in the implementation Java code seen in the next section. The model, when loaded in the optnode-api feature/bundle into the controller, will allow the MD-SAL data store to interpret optnode configuration write and read requests.

4.2.3. JAVA Implementation Modules

Once the classes are generated by YANG Tools, these can be used to start developing the implementation JAVA code in order to define the behaviour of the application. The next figure shows the different modules that will constitute the structure of the OPTNODE application:

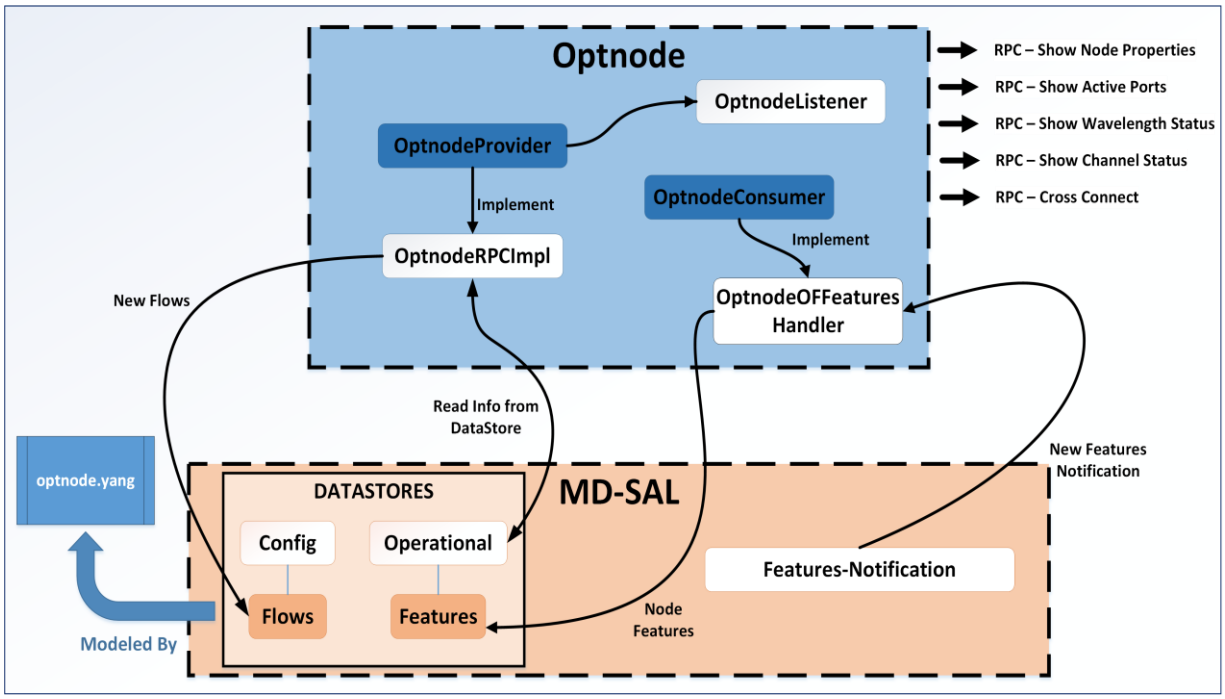

Fig. 4.6 OPTNODE Application Design

Inside the OPTNODE application it is possible to identify five different modules, where each one will provide a specific function:

- **OptnodeProvider:** This module will setup the main implementation of the application, it provides registration to different SAL services, creates an instance of optnode after the application is launched and launches parallel modules such as OptnodeRPCImpl and OptnodeListener.

- **OptnodeRPCImpl:** This module defines the behaviour of all the RPCs provided by the application, it interacts directly with the operational and configuration data stores.

- **OptnodeListener:** This module listens to any change that is made to the configuration data store.

- **OptnodeConsumer:** This module setups the consumer implementation part of the application, it is in charge of registering the application to notifications provided by other applications and launching the OptnodeOFFeaturesHandler module.

- **OptnodeOFFeaturesHandler:** This module is in charge of parsing the information received in the MD-SAL notifications from other applications. After parsing, it populates the operational data store with the new connected optical node features.

Although the provider module in theory is also able to handle consumer implementations, for this project it was selected to be handled separately by two different modules in order to give better structure to the application.

After the project skeleton is generated by the opendaylight archetype, an empty OptnodeProvider class is generated under the IMPL folder. This class in handled by another generated classes such as the OptnodeModule and OptnodeModuleFactory as it can be seen in the next figure.

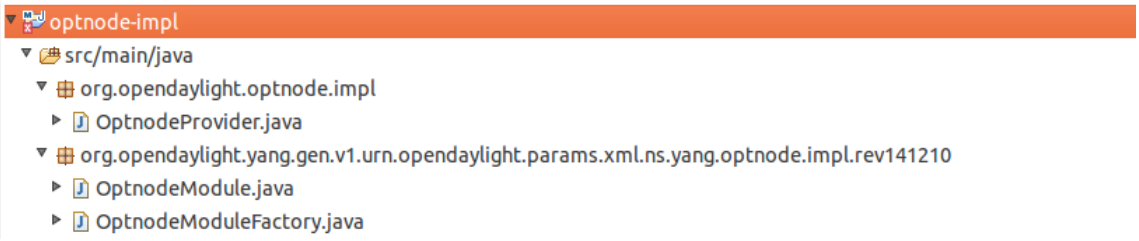

Fig. 4.7 Generated Implementation Classes

This /impl folder would be used to implement the five modules analysed before, so these can be run through maven in order to be integrated in the optnode-impl bundle. The next figure shows the folder after the implementation of the optnode modules:

Fig. 4.8 Implementation Classes

The source code of the five IMPL classes can be found in **ANEXES [VI]**, where it is possible to observe how the behaviour of each one and the interaction between modules, the MD-SAL and other applications is defined.

4.2.4. Features/Bundles Generation

After the generation of the JAVA classes is performed by the YANG Tools at the /api folder and the implementation code using these resources is developed at the IMPL folder, then it is possible to construct with maven the entire project from the main /optnode folder by using the *mvn clean install* command.

The result of running maven over the project will show a success build at the end of the compilation as it can be seen in the next figure:

```
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] optnode-api ....................................... SUCCESS [01:03 min]
[INFO] optnode-impl ...................................... SUCCESS [ 29.677 s]
[INFO] optnode-features .................................. SUCCESS [ 51.421 s]
[INFO] optnode-karaf ..................................... SUCCESS [03:57 min]
[INFO] optnode-artifacts ................................. SUCCESS [  1.935 s]
[INFO] optnode ........................................... SUCCESS [ 11.234 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 06:51 min
[INFO] Finished at: 2015-06-13T16:41:27+01:00
[INFO] Final Memory: 84M/222M
[INFO] ------------------------------------------------------------------------
rafael@UbuntuSDN:~/optnode$
```

Fig. 4.9 Maven Project Building

As seen in the figure above maven builds not only the IMPL and API folders, but also the FEATURES, KARAF and ARTIFACTS folders; integrating all the previous developed classes and models with these other modules. The main purpose of this build is the maven generation of the optnode-api and optnode-impl OSGi bundles.

The generated bundles are then automatically installed into the apache karaf in form of features, which include not only the optnode bundles but also the related dependencies and configuration files required to run the OPTNODE application, the next figure shows the installed features running in the Apache Karaf distribution built from the archetype.

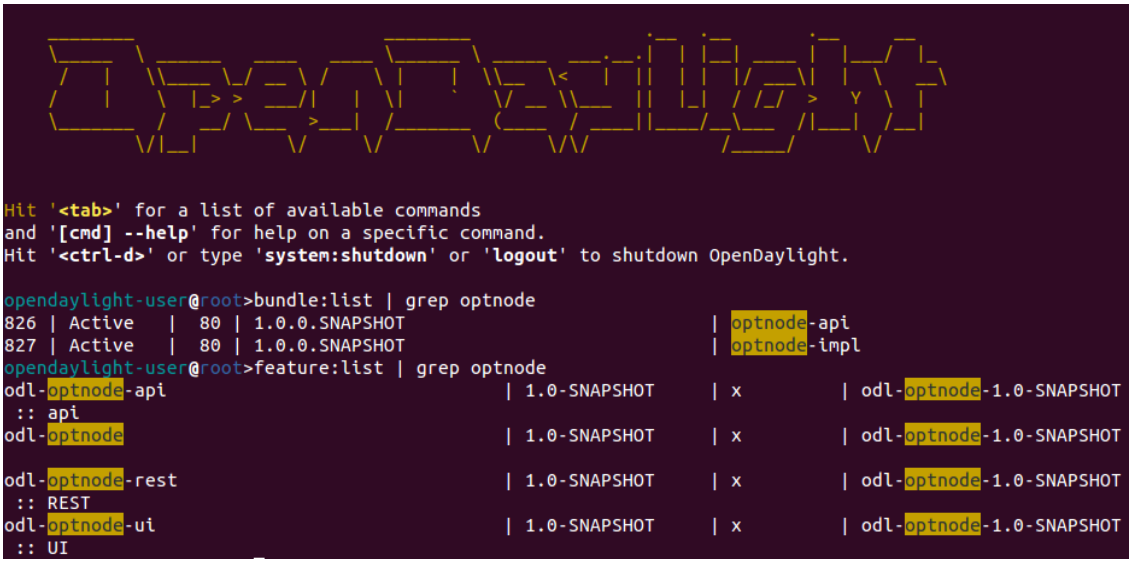

Fig. 4.10 Apache Karaf OPTNODE Features

As seen in the figure, both optnode-api and optnode-impl bundles show an “Active” state in the console, meaning that are installed and already running in Karaf. Optnode features on the other hand show an X state showing a correct installation, each feature may represent different dependencies and bundle requirements:

- **odl-optnode-api:** Contains dependencies to the odl-yangtools-models feature and includes the optnode-api bundle.
- **odl-optnode:** Contains dependencies to the odl-mdsal-broker, odl-optnode-api and odl-yangtools-models features and includes the model-inventory, model-topology, model-flow-base, model-flow-statistics, model-flow-service and optnode-impl bundles
- **odl-optnode-rest:** Contains dependencies to the odl-optnode and odl-restconf features.
- **odl-otpnode-ui:** Contains dependencies to the odl-optnode-rest, odl-mdsal-apidocs and odl-mdsal-xsql features.

4.3. OF Agent Design

In order to send the information received from the optical switching node to the controller, it is necessary to design an agent capable of encapsulating this information inside OF messages and being able also to handle messages coming from the controller. In summary the agent must be able to:

- Create a new TCP session and connect to the ODL controller, authenticate by using the OpenFlow protocol.
- Encapsulate the Optical switching node capabilities inside a specific type of OF message.
- Handle the reception of configuration information sent from the controller inside a specific type of OF message.

The next figure shows the basic structure of the JAVA based designed OF Agent, where it is possible to appreciate the interaction with the ODL controller.

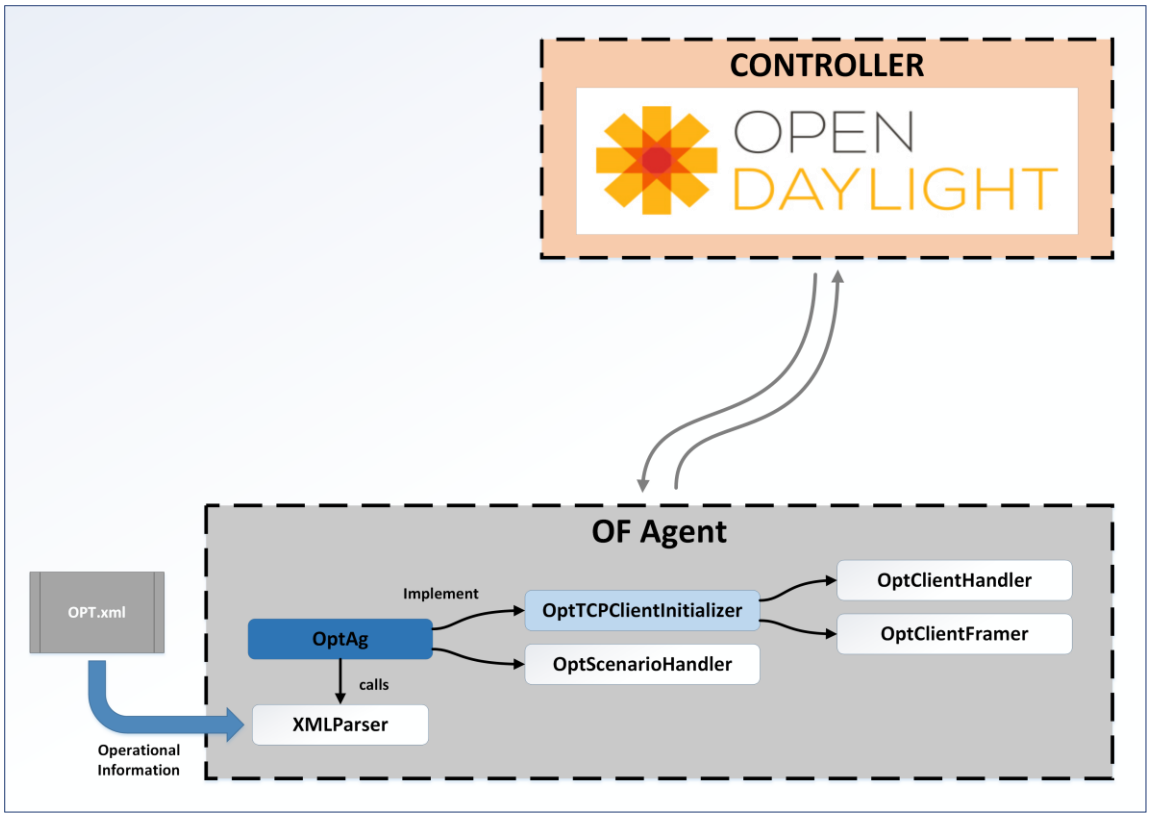

Fig. 4.11 OF Agent Design

Among the many modules implemented inside the OF Agent, it is possible to identify six main modules, where each one will provide a specific function:

- **OptAg:** The main OF Agent implementation module, it starts the agent and implements the OptTCPClientInitializer and OptScenarioHandler, it also calls the XMLParser to obtain the node's "operational information".

- **OptTCPClientInitializer:** This module is responsible for starting a new TCP session with the controller, by implementing the OptClientHandler and OptClientFramer modules.

- **OptClientHandler:** This module listens to all OF messages that come through the channel, derives the messages to the OptScenarioHandler.

- **OptClientFramer:** This module decodes OF messages from the bytes received through the channel, identifies type of received OF message.

- **OptScenarioHandler:** This module is in charge of handling received OF messages, deploying a specific action dependent on the type of message, actions could significate sending a new message or parsing the information that the message contains.

- **XMLParser:** This module parses the XML information from the OPT.xml file and sends it to the OptAg so it can be sent to the controller after the connection is established.

It is important to consider that in this case, as the project is focused on sending the node's information from the agent to the controller, this agent does not handle the connection to the FPGA that controls the optical switching node. Instead, the agent assumes that the state information has been already received and reads it from the OPT.xml file that represents such information.

4.3.1. Connecting to the ODL Controller

In order to connect to the ODL controller, the OF Agent must open a new session in order to exchange OF messages with the controller. As OpenFlow support is required also at the controller, the connection must be pointed to the OpenFlow Plugin module, an ODL module capable of handling OF messages and maintaining session with switches/agents via this protocol.

The OpenFlow Plugin shows its OF-switch-connection-provider service through port 6633, which is the port to be used by the OF Agent to establish the connection. After the TCP session is opened, an exchange of OFTP_HELLO messages must be performed in order to negotiate the OpenFlow version for the message exchange.

The next figure shows how after a new TCP connection is established towards port 6633 (ODL – OpenFlow Plugin module), an OFTP_HELLO message trying to negotiate version 1.3 of the protocol is sent from the controller to the OF Agent (in this case port 43121).

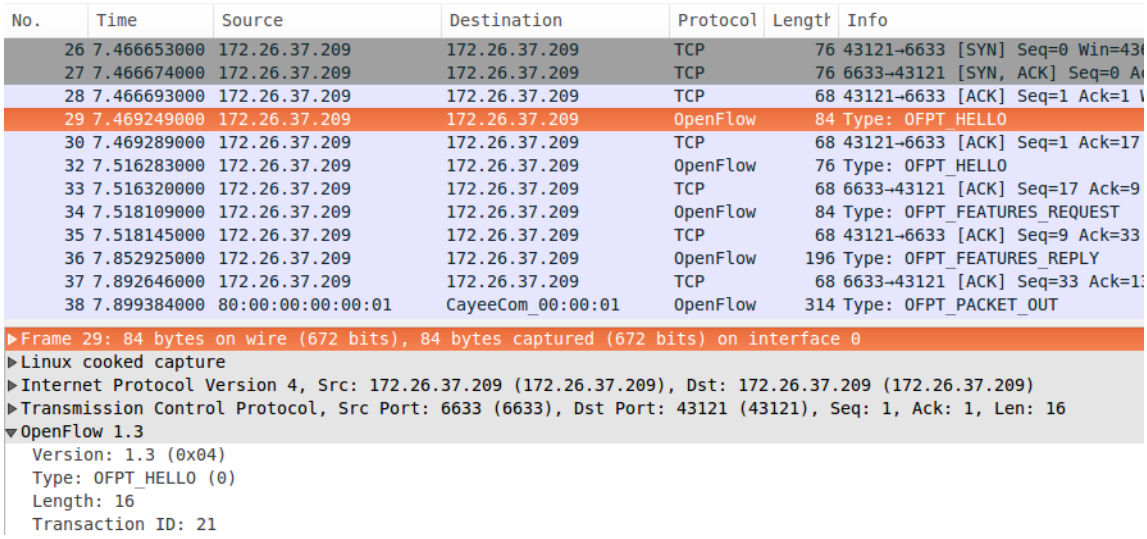

Fig. 4.12 OF Agent Connection Setup 1

This first OFPT_HELLO message is discarded by the OF Agent, as the agent is not intended to work with the 1.3 version of the OpenFlow protocol.

On the other hand, after the TCP connection is established, the OF Agent sends also an OFTP_HELLO message to try to negotiate in this case the OpenFlow version 1.0 as it can be seen in the next figure.

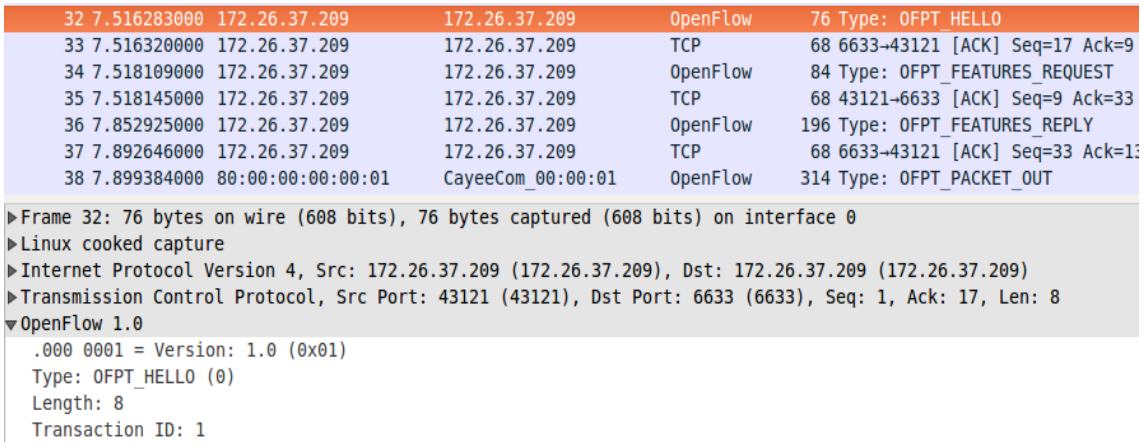

Fig. 4.13 OF Agent Connection Setup 2

As soon as the OpenFlow Plugin recognizes a switch capable of talking OpenFlow version 1.0, it sends back to it an OFTP_HELLO message with the negotiated version along with an OFTP_FEATURES_REQUEST message in order to request for the switch/agent capabilities.

The next figure shows how the controller sends both messages to the OF Agent, from port 6633(controller) to port 43121(Of Agent):

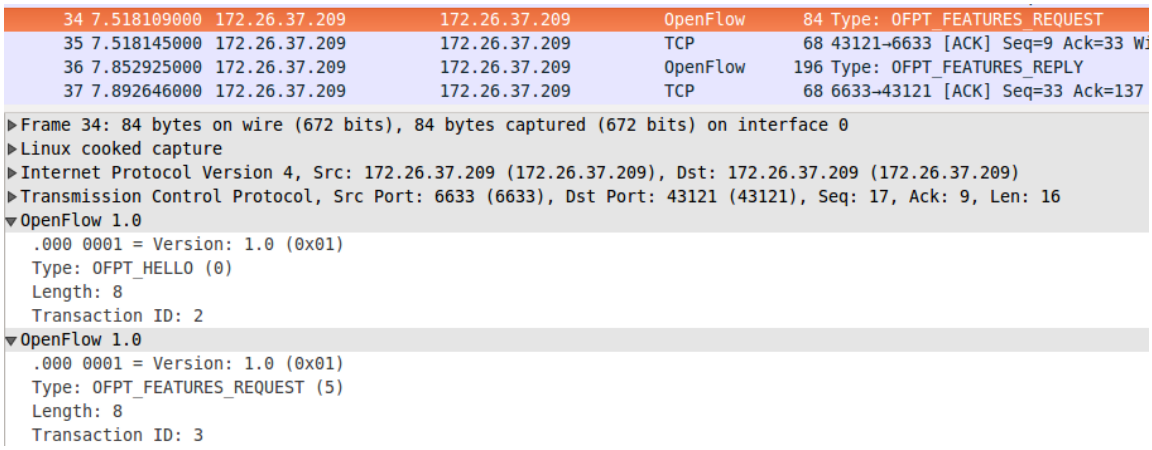

Fig. 4.14 OF Agent Connection Setup 3

4.3.2. Encapsulating State Information

After the TCP session is established and the OpenFlow version is negotiated between the OF Agent and the controller, the controller must sent an OFTP_FEATURES_REPLY containing the optical node state information, in this case this information is retrieved from the OPT.xml file which can be found in **ANEXES [VII]**.

The next figure shows the console of the OF Agent, where the information that the agent retrieves from the XML file can be observed:

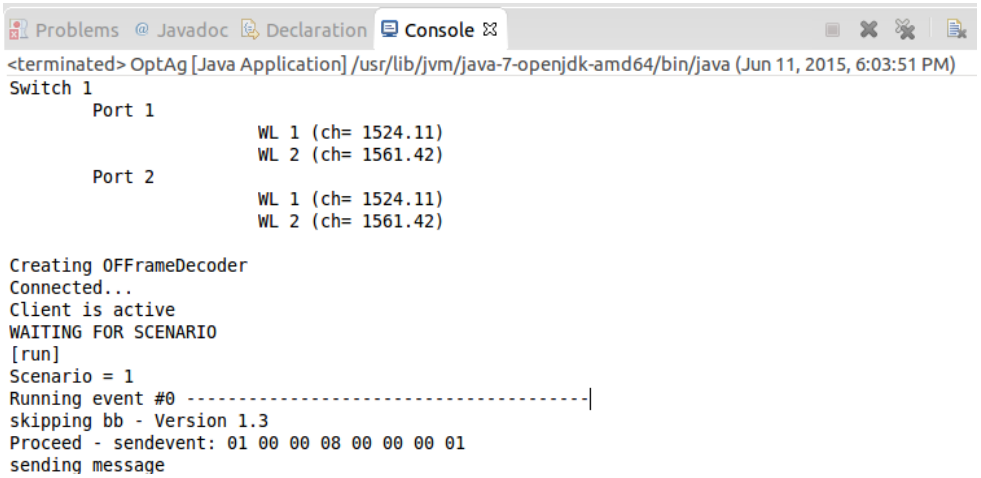

Fig. 4.15 OF Agent Sending State Info 1

Once the agent retrieves the state information, it can be sent inside the OFTP_FEATURES_REPLY message as explained before. The information inside the OF message is show in the next figure:

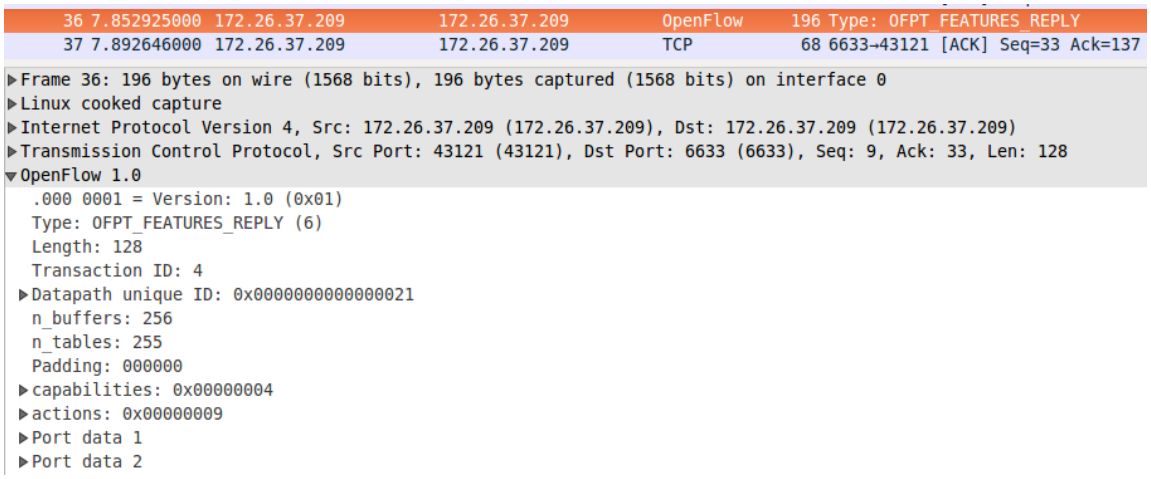

Fig. 4.16 OF Agent Sending State Info 2

The state information sent by means of this message is the one regarding port and wavelength capabilities and is encapsulated inside the OF message by the OptScenarioHandler module. The constructed message holds:

- **Port ID:** Encapsulated as Port Number/Name, inside port data field.
- **Wavelength ID & Channel:** Encapsulated as the HW Address, inside port data field. The specific method for encapsulating this state information inside a MAC address can be found in **ANEXES [VIII]**.

Next figure shows the encapsulated state information inside the port data field:

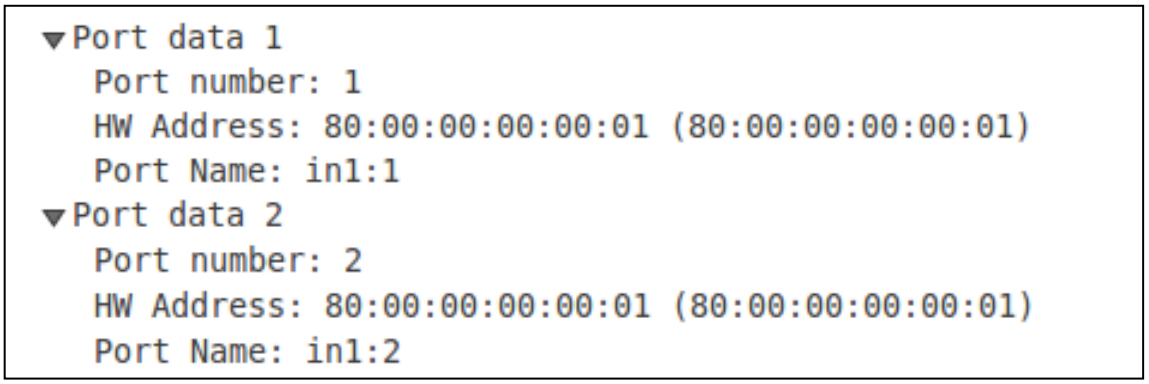

Fig. 4.17 OF Agent Sending State Info 2

After the information is sent by the OF Agent, the OpenFlow Plugin is in charge of retrieving the information that comes inside the message and publish it to the MD-SAL , from where it will be gathered by the OPTNODE application. More information regarding this interaction between both MD-SAL modules/apps can be found in the next chapter that delivers the implementation of the project.

4.3.3. Handling Configuration Information

The last function of the OF Agent is to provide support for incoming messages from the controller, sent via the OpenFlow Plugin and containing configuration information. In this case, OFTP_FLOW_MOD messages will be received by the OptAg module and handled by the OptScenarioHandler module.

The next figure shows the complete interaction between the ODL controller, including the OpenFlow Plugin, and the OF Agent.

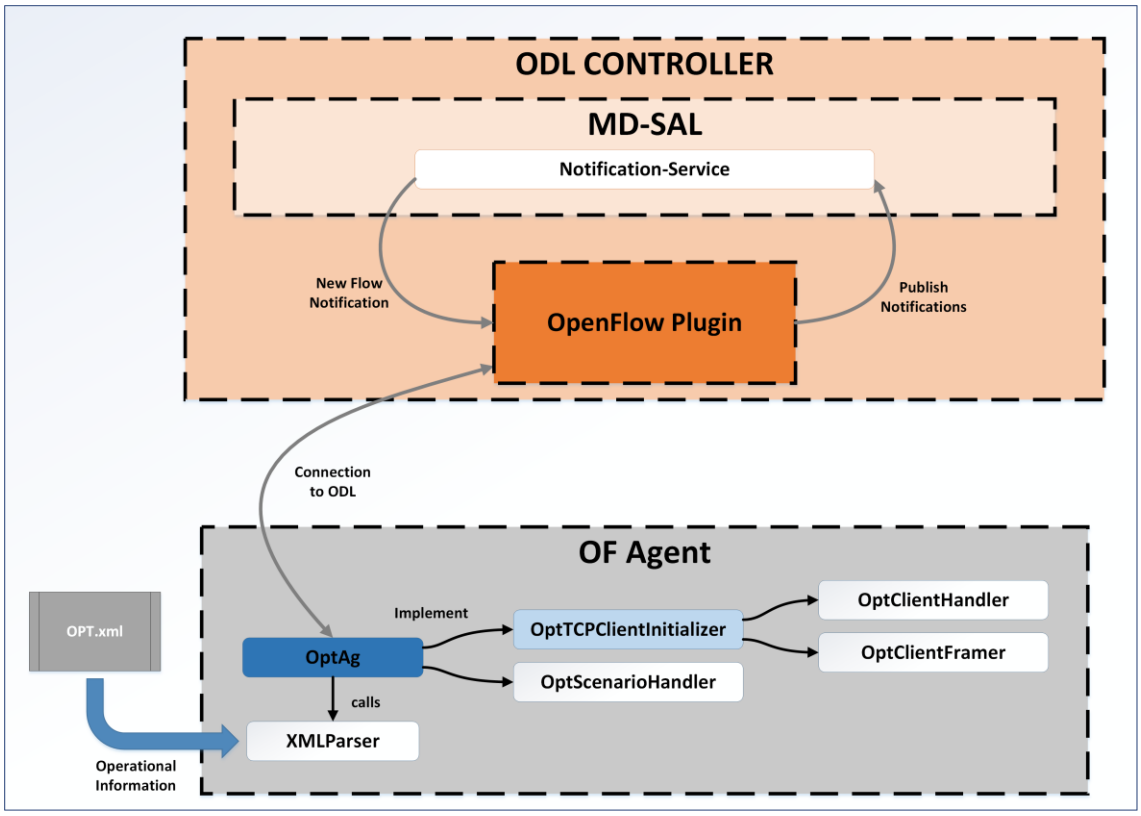

Fig. 4.18 OF Agent Connection

The OFPT_FLOW_MOD messages received at the agent will hold:

- **Matches:** Containing Input Port and Channel matches to be configured in the switch; in this case the channel will be encapsulated inside a Mac Address match in the same way as when sending channel operational data seen in the previous section.

- **Actions:** Containing Output Port action to be configured in the switch.

How this information is received by the OF Agent, once an RPC triggers an OFPT_FLOW_MOD message will be seen in more detail in the next chapter that holds the implementation part of the project. The source code of the main OF Agent modules can be found in **ANEXES [IX]**.

CHAPTER 5. OPTNODE PROJECT IMPLEMENTATION & TESTING

This chapter delivers the implementation part of the project, considering the interaction of the OPTNODE application with the OF Agent in different phases, showing several steps in the development of the application modules and the integration with the OpenFlow Plugin ODL distribution.

5.1. Implementation Phase 1

The first phase of implementation considers only the provider part modules of the OPTNODE application, where as it can be seen in the next figure only the OptnodeProvider, OptnodeListener and OptnodeRPCImpl are implemented. As there is no interaction at this point with other modules inside the controller or with any connected switch/agent, there exists a provisional OptNode1 module that allows populating the operational data store with the default node features. This module simulates the reception of the node's operational information, so the provider can implement the read-from-datastore RPCs.

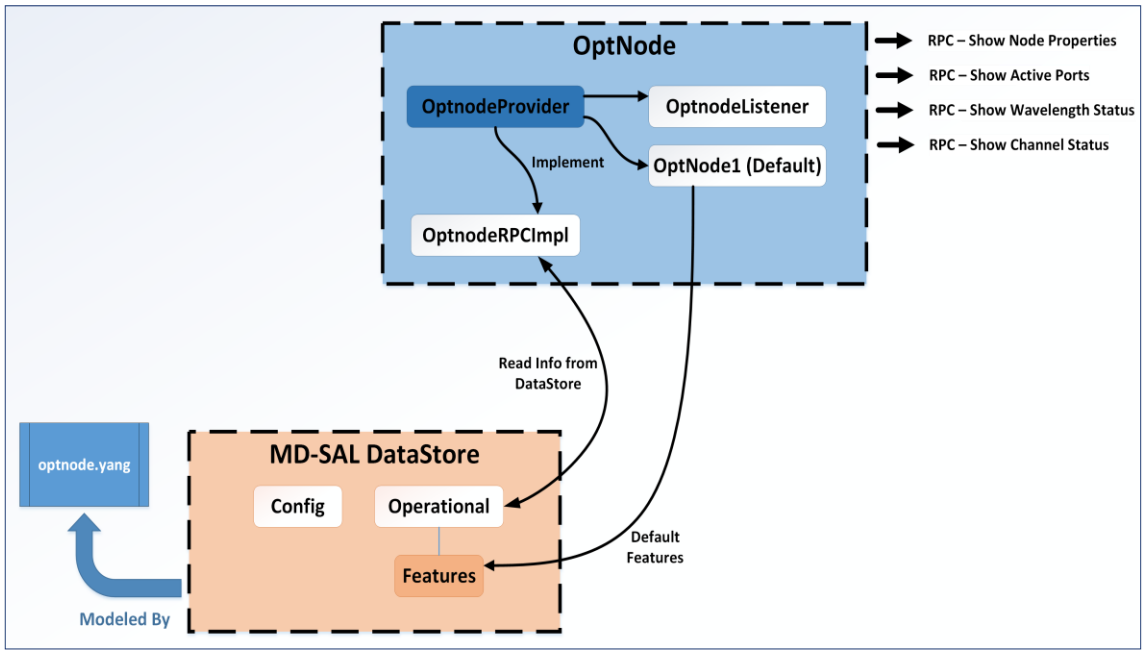

Fig. 5.1 OPTNODE Implementation Phase 1

The steps of operation for this phase of implementation are the following:

1) The OptnodeProvider module starts the OPTNODE application and implements the OptNode1 module in order to populate the operational datastore with the node's default state information.

2) The OptnodeProvider module implements the OpnodeListener module that works as a listener over the Configuration datastore, at this stage this module works in a passive way as configuration information is not yet deployed over the configuration datastore.

3) The OptnodeProvider module implements the OptnodeRPCImpl module, which deploys only the read-from-datastore RPCs, allowing testing these services by using a REST client to use them to retrieve the operational node's information from the datastore. In this case the last RPC that allows the generation of flows is not implemented yet.

This phase allows setting up an initial environment in order to test some operations and see how the interaction between datastore, application and rest clients is accomplished.

5.2. Implementation Phase 2

The second phase of implementation considers both the provider and consumer modules of the OPTNODE application, where the OpnodeConsumer and OptnodeOFFeaturesHandler modules are added to the previous phase of implementation as it can be seen in the next figure.

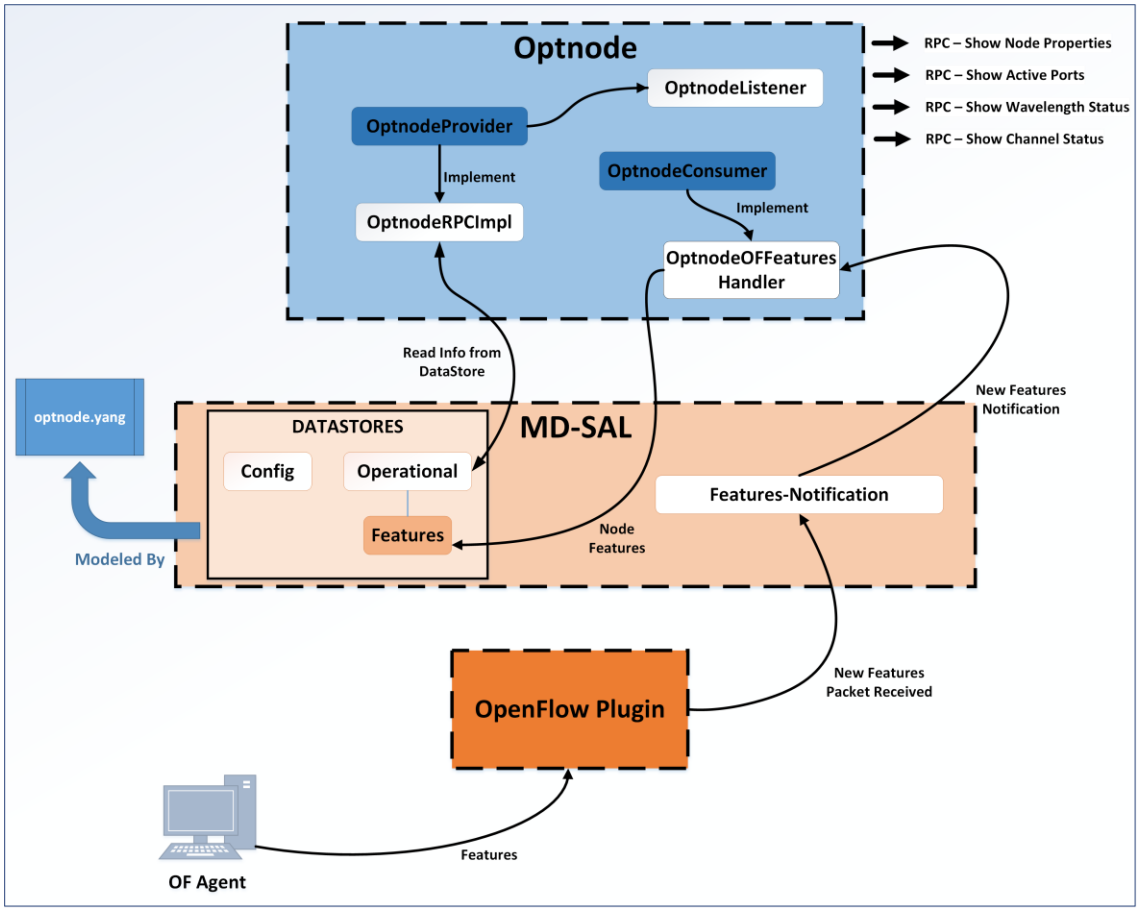

Fig. 5.2 OPTNODE Implementation Phase 2

This phase also considers the integration of the application with the OpenFlow Plugin and the OF Agent. In order to integrate the application, this one is exported into the OpenFlow Plugin available ODL distribution, which already contains the plugin running in Apache Karaf and listening to port 6633. The exporting procedure can be found in **ANEXES [X]**.

The steps of operation for this phase of implementation are the following:

1) The OptnodeProvider module starts the OPTNODE application and launches in parallel the Opnode Consumer module.

2) The OptnodeProvider module implements the OpnodeListener module that works as a passive listener over the Configuration datastore.

3) The OptnodeConsumer module registers to the MD-SAL and starts listening to Features notifications, in order to handle this type of notifications it implements the OptnodeOFFeaturesHandler.

4) The OptnodeProvider module implements the OptnodeRPCImpl module, which deploys only the read-from-datastore RPCs.

5) The OF Agent is started, the agent opens a new TCP session and exchanges OF messages with the OpenFlow Plugin, it sends the node's operational information inside an OFTP_FEATURES_REPLY message.

6) After the node's features are received at the plugin, it publishes a new Features notification to the MD-SAL.

7) The OptnodeConsumer listens to the new notification and derives it to the OptnodeOFFeaturesHandler, which populates the operational datastore with the information contained in the notification.

8) The OptnodeRCPImpl module now allows the access to operational information via read-only RPCs.

It is possible to observe how in this phase the interaction with the MD-SAL allows a full north-south integration between REST clients, the OPTNODE app, the operational datastore, the OpenFlow Plugin and the OF Agent.

5.3. Implementation Phase 3

The last phase of implementation is manly focused on the deployment of the configuration RPC that allows the population of the Configuration datastore and sending this information to the switch/agent via the OpenFlow Plugin.

The next figure shows the complete and final scenario for the OPTNODE application, where it can be observed how the use of this last RCP triggers the creation of flows in the database and in the MD-SAL in form of a notification.

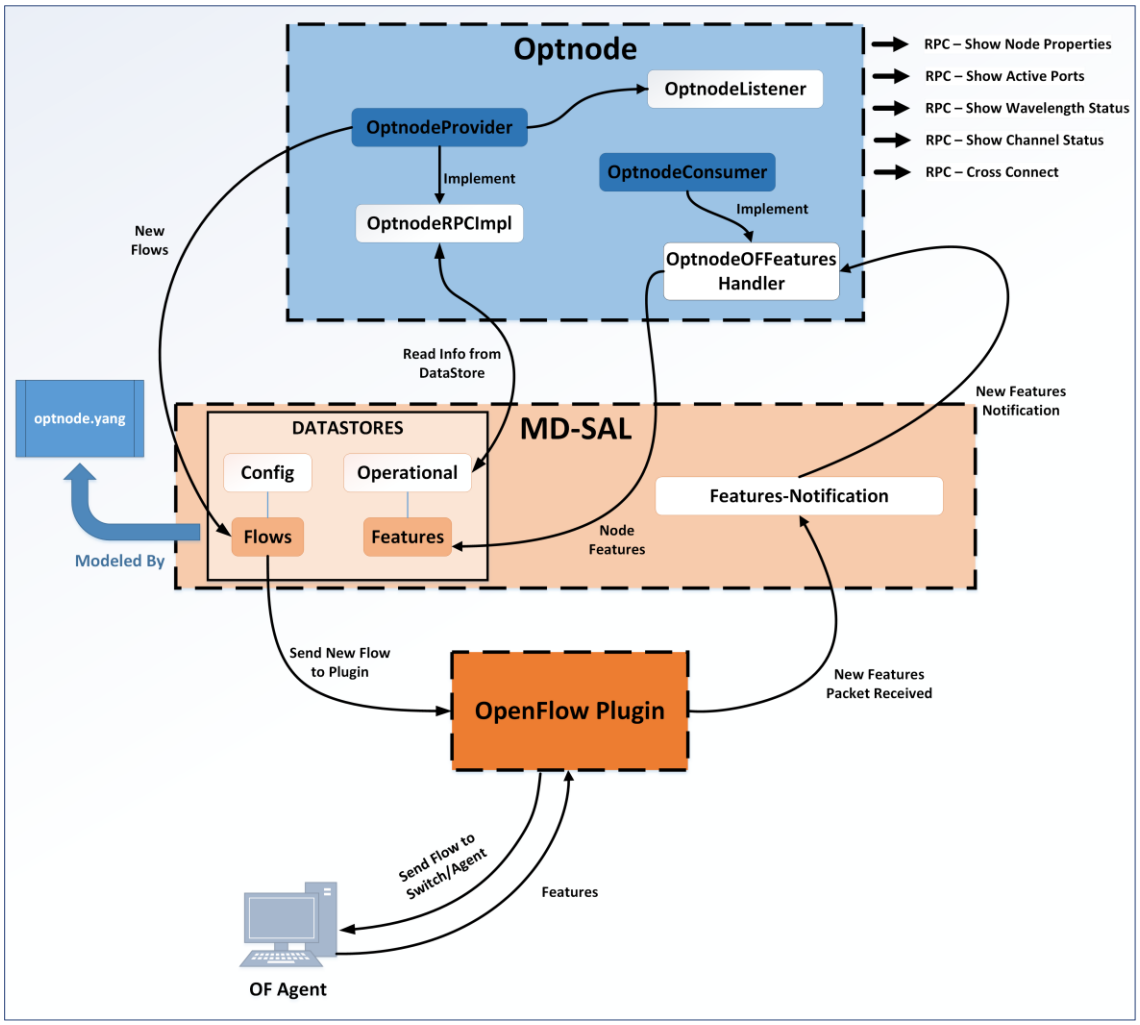

Fig. 5.3 OPTNODE Implementation Phase 3

The steps of operation for this phase of implementation are the following:

1) The OptnodeProvider module starts the OPTNODE application and launches in parallel the Opnode Consumer module.

2) The OptnodeProvider module implements the OpnodeListener module that works as a passive listener over the Configuration datastore and the OptnodeRPCImpl that deploys the RPCs.

3) The OptnodeConsumer module registers to the MD-SAL and starts listening to Features notifications, in order to handle this type of notifications it implements the OptnodeOFFeaturesHandler.

4) The OF Agent is started, the agent opens a new TCP session and exchanges OF messages with the OpenFlow Plugin, it sends the node's operational information to the plugin and this one publishes a new Features notification to the MD-SAL.

5) The OptnodeConsumer listens to the new notification and derives it to the OptnodeOFFeaturesHandler, which populates the operational datastore with the information contained in the notification.

6) The OptnodeRCPImpl module now allows the access to operational information via read-only RPCs and it able to provide a configuration RPC to allow the deployment of flows (cross-connections) at the node.

7) After a REST client uses the configuration RPC to create a new cross-connection, the OptnodeRPCImpl populates the configuration datastore with the new information and constructs a new OFTP_FLOW_MOD message that is published in form of a notification.

8) The OpenFlow Plugin listens to the new flow notification and redirects the OFTP_FLOW_MOD message to the switch/agent.

9) The OF Agent receives the OFTP_FLOW_MOD message retrieves the contained configuration information so it can be passed to the switch in a future stage of the project.

As reviewed in this last phase of implementation, a complete support for operational and configurational RPCs is accomplished, allowing not only the retrieval of node's capabilities by other applications, but also the creation of new cross-connection over the optical switching node.

5.4. RPC Testing

After exposing the final stage of implementation, it is necessary to test all the provided RPCs in order to show the correct operation of the OPTNODE application. Testing was performed using a REST client in order to call each specific RPC via the REST protocol.

5.4.1. RPC show-node-properties

The show-node-properties RPC provides a read-only operation over this particular datastore, it does not require an input and provides an output containing the node ID, IP and number of ports.

The next figure shows the request of this operation by the REST client, where it is possible to observe how the client sends an empty POST message to the REST operation, in this case the URL /restconf/operations/optnode:show-node-properties that sends back the requested state information in JSON format.

Fig. 5.4 RPC show-node-properties

Inside the controller it is also possible to observe how the call to this operation is shown as a LOG message in the console as seen next.

```
2015-06-11 17:24:35,170 | INFO  | qtp867754909-702 | OptnodeRPCImpl
|827 - org.opendaylight.optnode.impl - 1.0.0.SNAPSHOT | Show Node
Properties!!!

2015-06-11 17:24:35,172 | INFO  | qtp867754909-702 | OptnodeRPCImpl
|827 - org.opendaylight.optnode.impl - 1.0.0.SNAPSHOT | SUCCESS

2015-06-11 17:24:35,172 | INFO  | qtp867754909-702 | OptnodeRPCImpl
|827  -  org.opendaylight.optnode.impl  -  1.0.0.SNAPSHOT  |  ID:
openflow:33  IP: 172.26.37.209  Ports: 2
```

5.4.2. RPC show-active-ports

The show-active-ports RPC provides also a read-only operation; it does not require an input and it provides an output containing the list of node's interfaces with their capabilities. The console shows the following LOG messages:

```
2015-06-11 17:23:53,382 | INFO  | qtp867754909-702 | OptnodeRPCImpl
| 827 - org.opendaylight.optnode.impl - 1.0.0.SNAPSHOT | Show Active
Ports!!!

2015-06-11 17:23:53,392 | INFO  | qtp867754909-702 | OptnodeRPCImpl
| 827 - org.opendaylight.optnode.impl - 1.0.0.SNAPSHOT | SUCCESS

2015-06-11 17:23:53,394 | INFO  | qtp867754909-702 | OptnodeRPCImpl
| 827 - org.opendaylight.optnode.impl - 1.0.0.SNAPSHOT | Ports:
[Interface{getNumberofwavelengths=2, getPortid=2,
getWavelength=[Wavelength{getChannel=1561.42, getWaveid=2,
augmentations={}}, Wavelength{getChannel=1524.11, getWaveid=1,
augmentations={}}], augmentations={}},
Interface{getNumberofwavelengths=2, getPortid=1,
getWavelength=[Wavelength{getChannel=1561.42, getWaveid=2,
augmentations={}}, Wavelength{getChannel=1524.11, getWaveid=1,
augmentations={}}], augmentations={}}]
```

The next figure shows the request of the RPC by the REST client and the received list of interfaces in JSON format.

Fig. 5.5 RPC show-active-ports

5.4.3. RPC show-wavelength-status

The show-wavelength-status RPC provides a read-only operation that requires as an input the port ID and provides as an output the list of supported wavelengths of that port. The console shows the following LOG messages:

```
2015-06-11 17:23:09,296 | INFO  | qtp867754909-702 | OptnodeRPCImpl
| 827 - org.opendaylight.optnode.impl - 1.0.0.SNAPSHOT | Show
Wavelengths of Port: ShowWavelengthsStatusInput{getPortid=2,
augmentations={}}

2015-06-11 17:23:09,303 | INFO  | qtp867754909-702 | OptnodeRPCImpl
| 827 - org.opendaylight.optnode.impl - 1.0.0.SNAPSHOT | SUCCESS

2015-06-11 17:23:09,304 | INFO  | qtp867754909-702 | OptnodeRPCImpl
| 827 - org.opendaylight.optnode.impl - 1.0.0.SNAPSHOT |
Wavelengths: [Wavelength{getChannel=1561.42, getWaveid=2,
augmentations={}}, Wavelength{getChannel=1524.11, getWaveid=1,
augmentations={}}]
```

The next figure shows the request from the REST client, containing the required port ID in JSON format and the output received containing the list of wavelengths supported for that specific port.

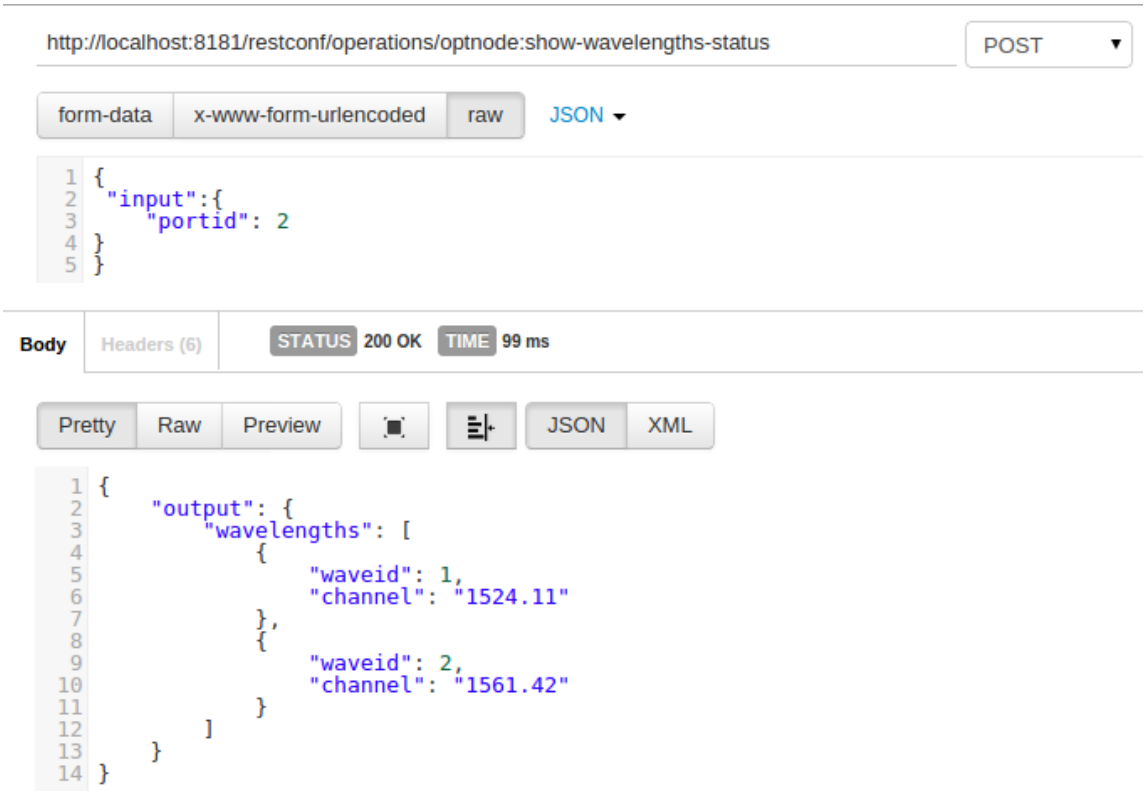

Fig. 5.6 RPC show-wavelength-status

5.4.4. RPC show-channel-status

The show-channel-status RPC provides a read-only operation that requires as an input both a port ID and a wavelength ID, it returns as an output the supported channel that correspond to the input variables. The console shows in this case the following LOG messages.

```
2015-06-11 17:22:12,312 | INFO  | qtp867754909-702 | OptnodeRPCImpl
| 827 - org.opendaylight.optnode.impl - 1.0.0.SNAPSHOT |
Show Channel of Wavelength/Port: ShowChannelStatusInput{getPortid=2,
getWaveid=1, augmentations={}}

2015-06-11 17:22:12,315 | INFO  | qtp867754909-702 | OptnodeRPCImpl
| 827 - org.opendaylight.optnode.impl - 1.0.0.SNAPSHOT | SUCCESS

2015-06-11 17:22:12,316 | INFO  | qtp867754909-702 | OptnodeRPCImpl
| 827 - org.opendaylight.optnode.impl - 1.0.0.SNAPSHOT | Channel:
1524.11
```

The next figure shows the request by the RPC client containing the input required variables and obtaining as an output the supported channel. In this

case the request is made by means of a POST message towards /restconf/operations/optnode:show-channel-status URL.

Fig. 5.7 RPC show-channel-status

5.4.5. RPC cross-connect

The cross-connect RPC is the main configuration operation that provides to the client the ability to configure cross-connections in the optical switching node via the OPTNODE application. It requires as input variables the channel to be used in the cross connection, as well as the input and output ports.

In the case that the client sends a request to this RPC with the same port ID in both input and output port variables, the operation will not be executed and the console will show the following LOG message:

```
2015-06-11 17:50:47,592 | INFO  | tp867754909-1082 | OptnodeRPCImpl
| 827 - org.opendaylight.optnode.impl - 1.0.0.SNAPSHOT |
CrossConnect Channel/Wavelength/Ports:
CrossConnectInput{getChannel=1561.42, getInportid=1, getOutportid=1,
augmentations={}}

2015-06-11 17:50:47,593 | WARN  | tp867754909-1082 | OptnodeRPCImpl
| 827 - org.opendaylight.optnode.impl - 1.0.0.SNAPSHOT | Output Port
cannot be the same as Input Port
```

It is also possible that the client sends a request via REST to cross connect to ports using a channel that may not be supported by one of these ports. In this case the OPTNODE application uses a function to check in the operational datastore if the channel is currently supported by both ports. If it is not, then the console shows the following LOG message.

```
2015-06-11 17:52:24,613 | INFO  | tp867754909-1082 | OptnodeRPCImpl
| 827 - org.opendaylight.optnode.impl - 1.0.0.SNAPSHOT |
CrossConnect Channel/Wavelength/Ports:
CrossConnectInput{getChannel=1561.40, getInportid=1, getOutportid=2,
augmentations={}}

2015-06-11 17:52:24,614 | WARN  | tp867754909-1082 | OptnodeRPCImpl
| 827 - org.opendaylight.optnode.impl - 1.0.0.SNAPSHOT | Wavelenght
not supported by Input port: 1
```

If the input information is correct and the checking functions do not return errors, then the OPTNODE application populates the datastore with this new configuration information as shown in the next LOG message.

```
2015-06-11 17:04:38,297 | INFO  | qtp867754909-410 | OptnodeRPCImpl
| 827 - org.opendaylight.optnode.impl - 1.0.0.SNAPSHOT |
CrossConnect Channel/Wavelength/Ports:
CrossConnectInput{getChannel=1561.42, getInportid=2, getOutportid=1,
augmentations={}}

2015-06-11 17:04:38,365 | INFO  | qtp867754909-410 | OptnodeRPCImpl
| 827 - org.opendaylight.optnode.impl - 1.0.0.SNAPSHOT | FLOW ADDED:
AddFlowInput [_instructions=Instructions [_instruction=[Instruction
[_instruction=ApplyActionsCase [_applyActions=ApplyActions
[_action=[Action [_action=OutputActionCase
[_outputAction=OutputAction [_outputNodeConnector=Uri [_value=1],
augmentation=[]], augmentation=[]], _key=ActionKey [_order=1],
_order=1, augmentation=[]]], augmentation=[]], augmentation=[]],
_key=InstructionKey [_order=1], _order=1, augmentation=[]]],
augmentation=[]], _match=Match [_ethernetMatch=EthernetMatch
[ ethernetDestination=EthernetDestination [ address=MacAddress
[_value=80:00:00:00:00:00], augmentation=[]], augmentation=[]],
_inPort=Uri [_value=2], augmentation=[]], _node=NodeRef
[_value=KeyedInstanceIdentifier{targetType=interface
org.opendaylight.yang.gen.v1.urn.opendaylight.inventory.rev130819.no
des.Node,
path=[org.opendaylight.yang.gen.v1.urn.opendaylight.inventory.rev130
819.Nodes,
org.opendaylight.yang.gen.v1.urn.opendaylight.inventory.rev130819.no
des.Node[key=NodeKey [_id=Uri [_value=openflow:33]]]]}],
augmentation=[]]


2015-06-11 17:04:38,382 | INFO  | qtp867754909-410 | OptnodeConsumer
| 827 - org.opendaylight.optnode.impl - 1.0.0.SNAPSHOT | init
Optnode Config: Config Status Populated: OptnodeProperties
[_flow=[Flow [_channel=1561.42, _inportid=2, _key=FlowKey
[_channel=1561.42], _outportid=1, augmentation=[]]],
augmentation=[]]


2015-06-11 17:04:38,430 | INFO  | lt-dispatcher-16 | OptnodeListener
| 827 - org.opendaylight.optnode.impl - 1.0.0.SNAPSHOT |
onDataChanged - new Optnode config:
OptnodeProperties{getFlow=[Flow{getChannel=1561.42, getInportid=2,
getOutportid=1, augmentations={}}], augmentations={}}
```

In these LOG messages it is also possible to see how the OptnodeListener identifies that a new configuration has been saved in the Configuration datastore, and how a new flow notification is generated in parallel and published to the MD-SAL in order to be handled by the OpenFlow Plugin. The next figure shows the original request of the RPC by the REST client.

Fig. 5.8 RPC cross-connect

After the OpenFlow Plugin receives the new flow notification, it sends the OFTP_FLOW_MOD message to the OF Agent, containing the configuration information inside matches and actions, as explained in the previous chapter. The next figure shows the message sent from the controller (port 6633) to the OF Agent (port 43121).

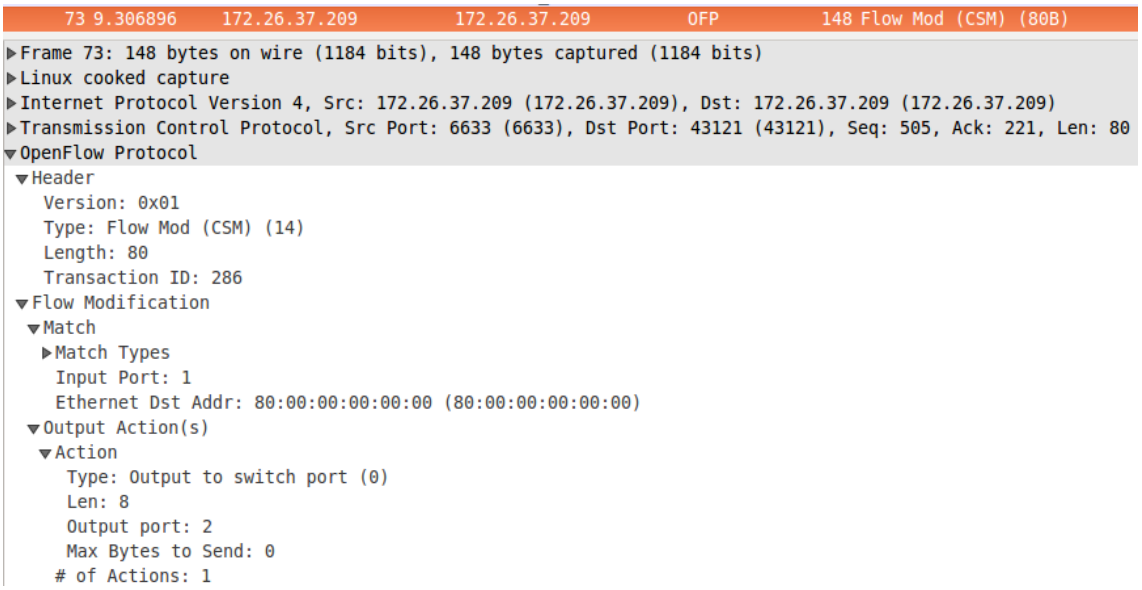

Fig. 5.9 New OFTP_FLOW_MOD message

Finally, this information is received and retrieved from the message at the OF Agent, so it can be passed on a future stage, to the optical switching node. The next figure shows the console of the agent, where a LOG message shows that the information of a new OFTP_FLOW_MOD message has been received.

```
----------------  NEW FLOW_MOD  -----------------

MATCH  ------> OFMatch[in_port=1,dl_dst=80:00:00:00:00:00]

ACTION   ----> OFActionOutput [maxLength=0, port=2, length=8, type=OUTPUT]
```

Fig. 5.10 Message Reception at the OF Agent

As seen in the picture above, the agent at the final stage of the cross-connect RPC implementation, receives a match field containing both the input port ID and the channel to cross connect, this last one encapsulated inside the destination Mac Address by using the same method as the one seen in **ANEXES [VIII]**.

In order to retrieve this channel data the same method of encapsulation is used in reverse mode. Then, the output port ID is also retrieved from the action field received inside the OF message.

CHAPTER 6. CONCLUSIONS

The OPTNODE app, as seen in the implementation chapter, delivers a fully runnable MD-SAL application, capable of handling the connection of an optical switch by means of the OF Agent. The final testing shows that the app is able to provide five different RPCs via REST to clients, delivering operational and configuration operations over the connected switch.

Regarding the application itself, its design allows a complete interaction with different elements of the controller, such as the MD-SAL Databases, MD-SAL Services and the OpenFlow Plugin, interactions that are crucial in order to setup a full north-south connectivity from the REST clients to the OF Agent.

On the other hand, the design of the OF agent allows the OpenFlow message exchange that is necessary to send the optical switch capabilities to the controller and to retrieve from it the configuration data. Integration of the project with the OpenFlow Plugin ODL distribution can be considered fundamental in order to allow this OFAgent-to-Controller OpenFlow connection.

The use of OpenFlow v 1.0 in the project shows that the protocol is fully capable of communicating the optical switching node's information via the OF Agent, regarding not only state data via an OFTP_FEATURES_REPLY message but also configuration data to the switch via OFTP_FLOW_MOD messages.

It is also important to consider that the development of the application over the Opendaylight SDN controller has shown an appropriated controller choice in terms of open community support, ease of implementation, controller modules and northbound/southbound open interfaces.

As with every project related with SDN, OPTNODE also works in an ongoing development model, where more intelligence and functionalities could be added to the application considering the future release of a new version of OPTNODE. Some next steps that could be taken towards a new version are presented next:

- Testing documentation regarding the integration with the parallel project, considering the connection to the optical switching node via the FPGA.

- Modifications to the YANG model, allowing the provision of new RPCs or the ability to handle more than one connected switch/agent at once.

- Design of an AngularJS based GUI (Graphical User Interface) able to run over the DLUX user interface of the ODL controller, allowing a more practical request of the provided RPCs and a better exposure of the operational data to the final user.

ANEXES

I. OpenFlow v1.0.0 Message Exchange

The following figure shows the basic transaction of OF messages between the controller and SDN network device, it is important to consider that this is the exchange seen in version 1.0.0 of the OpenFlow protocol, so it can vary in some aspects from newer versions.

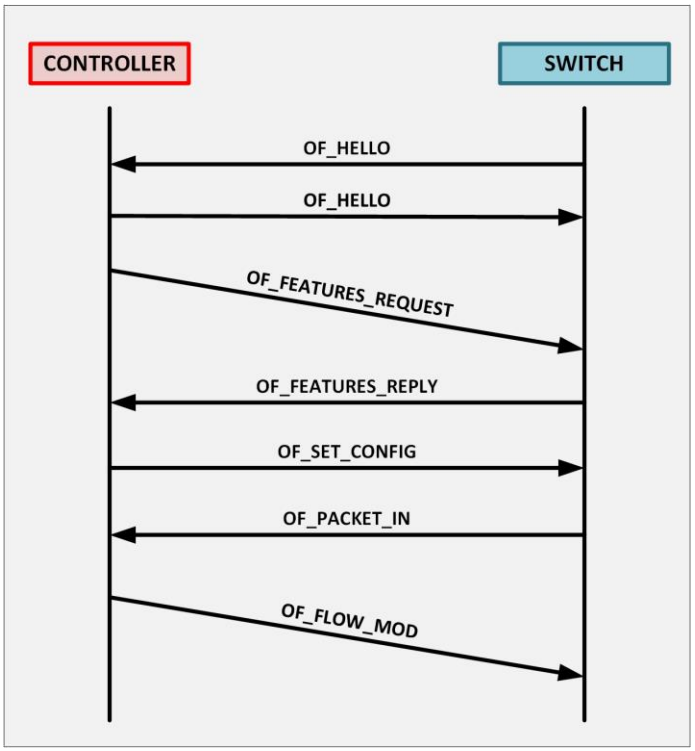

Figure 1 - OF v1.0 Message Exchange

When an TCP connection is first established, both switch and controller send OF_HELLO messages with the version set to the highest possible. The version then is negotiated, if is not the same one in both messages, and can result in another OF_HELLO or in an OF_ERROR if the version is not supported.

After negotiation, the controller sends an OF_FEATURES_REQUEST message to request for switching capabilities, which are sent back to the controller through an OF_FEATURES_REPLY message.

Then, the controller is sends an OF_SET_CONFIG message containing sets of flags and max bytes for packets, so the device can use this information when sending packets to the controller.

When a packet arrives to the switch and does not matches any entry in the switch's flow table, it is advised and sent to the controller using an OF_PACKET_IN message, so this one can process the packet and implement new flows in the switch through OF_FLOW_MOD messages. These flows can be sent in both proactive and reactive ways, depending on the configuration of the SDN controller.

II. MD-SAL Plugin Design

In order to design a provider MD-SAL plugin, the designer must follow specific steps:

1) First, the designer decides which data and how will be provided by the plugin and designs the data model for the provided data. The data model (expressed in yang) is then run through the yang tools, which generate the SAL APIs for the model.

2) Then, the implementation for the generated provider API, along with other plugin features and functionality, are developed. The resulting code is packaged in a "plugin" OSGI bundle. It is also possible to package the code of a subsystem in multiple plugins or applications that may communicate with each other through the SAL.

3) In parallel, the generated APIs and a set of helper classes are also built and packaged in an "API" OSGI bundle.

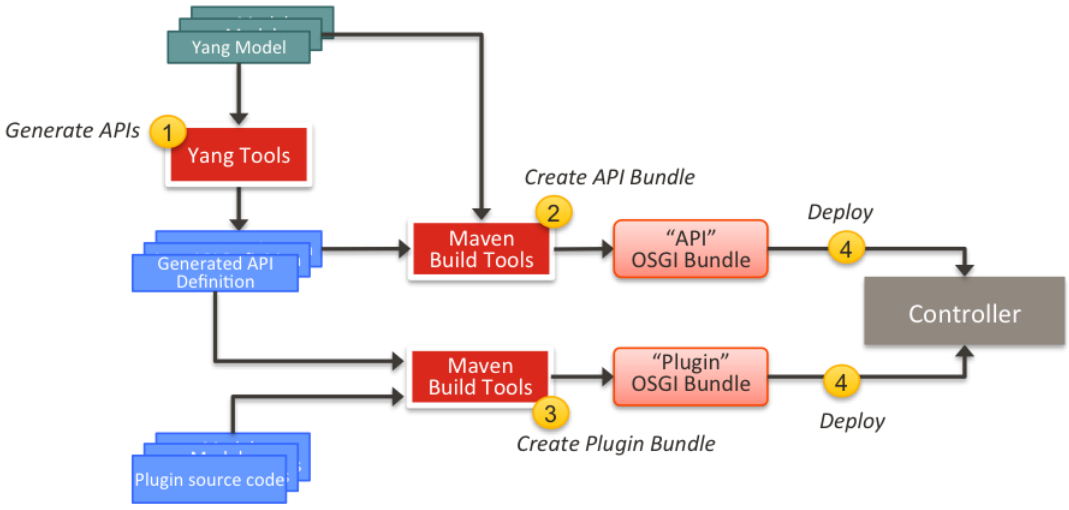

Figure 2 - MD-SAL Plugin Design

The plugin development process is shown in the figure above, where after the OSGI bundle of a MD-SAL plugin is loaded into the controller and activated, the plugin is ready for operation.

III. Opendaylight Start-up Archetype Setup

The OPTNODE application uses an archetype given by ODL in order to setup the skeleton of the project. Before this archetype can be run it is necessary to fulfil the following environment requirements:

- Maven 3.1.1 or higher version.
- Java 1.7.0 (Java 7 JDK).

Once the development machine is setup with Maven and Java, it is necessary to update the Maven settings.xml in order to let Maven know about the location of Opendaylight repositories and how to use them. This is required because OpenDaylight maintains its own repositories outside of Maven Central, so maven cannot resolve OpenDaylight artifacts by default.

The update on the settings.xml can be performed using the following command:

```
cp -n ~/.m2/settings.xml{,.orig} ; \wget -q -O -
https://raw.githubusercontent.com/opendaylight/odlparent/master/settings.xml >
~/.m2/settings.xml
```

After updating the settings.xml, it is also recommended to increase the amount of RAM that Maven can use by adding the following line in the *~/.bashrc* file:

```
export MAVEN_OPTS='-Xmx1048m -XX:MaxPermSize=512m'
```

Then, it is possible to setup the new OPTNODE project by calling the opendaylight archetype named “opendaylight-startup-archetype” with the mvn archetype:generate command as it can be seen next:

```
mvn archetype:generate -DarchetypeGroupId=org.opendaylight.controller -
DarchetypeArtifactId=opendaylight-startup-archetype
```

Finally, some information will be required at the prompt in order to generate the new project as seen next:

```
Define value for property 'groupId': : org.opendaylight.optnode
Define value for property 'artifactId': : optnode
Define value for property 'version': :  1.0-SNAPSHOT
Define value for property 'package': :  org.opendaylight.optnode
Define value for property 'copyright': : 2015 UPC-EETAC
```

Then, it will be possible to see the auto-generated skeleton at the /optnode folder where the following structure would be found:

```
api/
artifacts/
features/
impl/
karaf/
pom.xml
```

This generated structure will be used as the basis for the deployment of the OPTNODE application, where each generated folder will accommodate to a specific function:

- **api:** contains public API definitions for the OPTNODE application. It's formed of one or more yang models that will define OPTNODE's REST and Java APIs. The APIs define the application's functionality.

- **artifacts:** Not to be used in this project (still in development).

- **features:** At the Karaf container level, every app/plugin is a set of features that can be installed in the container. This folder contains definitions of OPTNODE's Karaf features and their dependencies, features that must be installed in the container for OPTNODE to work.

- **impl:** contains code and configuration for the provider and consumer part of OPTNODE application, it is where all the code implementation will be performed.

- **karaf:** the ODL custom distribution that contains OPTNODE and all its dependencies is built in this folder. It is possible to use this custom distribution or install OPTNODE in a pre-existing distribution.

IV. Data Modelling Basics

YANG defines different types of blocks for data modelling. According to the type each block may offer different characteristics:

- **Leaf Nodes:** A leaf node contains simple data like an integer or a string. It has exactly one value of a particular type and no child nodes.

- **Leaf-List Nodes:** A leaf-list is a sequence of leaf nodes with exactly one value of a particular type per leaf.

- **Container Nodes:** A container node is used to group related nodes in a sub-tree. A container has only child nodes and no value. A container may contain any number of child nodes of any type (including leafs, lists, containers, and leaf-lists).

- **List Nodes:** A list defines a sequence of list entries. Each entry is like a structure or a record instance, and is uniquely identified by the values of its key leafs. A list can define multiple key leafs and may contain any number of child nodes of any type (including leafs, lists, containers etc.).

YANG can model state data, as well as configuration data, based on the "config" statement. When a node is tagged with "config false", its sub-hierarchy is flagged as state data, to be reported using the <get> operation, not the <get-config> operation. Parent containers, lists, and key leafs are reported also, giving the context for the state data.

YANG has a set of built-in types, similar to those of many programming languages, but with some differences due to special requirements from the management domain. The following table summarizes the built-in types:

Table. 1 YANG Built-in Types

Name	Description
binary	Any binary data
bits	A set of bits or flags
boolean	"true" or "false"
decimal64	64-bit signed decimal number
empty	A leaf that does not have any value
enumeration	Enumerated strings
indentityref	A reference to an abstract identity
instanceidentifier	References a data tree node
int8	8-bit signed integer
int16	16-bit signed integer
int32	32-bit signed integer
int64	64-bit signed integer
leafref	A reference to a leaf instance
string	Human-readable string

uint8	8-bit unsigned integer
uint16	16-bit unsigned integer
uint32	32-bit unsigned integer
uint64	64-bit unsigned integer
union	Choice of member types

YANG allows the definition of RPCs. The operations' names, input parameters, and output parameters are modelled using YANG data definition statements. Finally, YANG also allows the definition of notifications. YANG data definition statements are used to model the content of the notification.

V. The OPTNODE YANG File

The base .yang file code for the OPTNODE module can be seen next, where leafs, lists and keys are declared inside a container, and where RPCs are also defined in parallel to the container.

Code 1 - OPTNODE .yang

```
module optnode {
    yang-version 1;
    namespace "urn:opendaylight:params:xml:ns:yang:optnode";
    prefix "optnode";

      revision "2015-01-05" {
        description "Initial revision of optnode model";

container optnode-properties {
      leaf nodeid {
            type string;
            config false;
      }
      leaf nodeip {
            type string;
            config false;
      }
      leaf numberofports {
            type int32;
            config false;
      }
      list interface {
            key portid;
            leaf portid {
                  type uint32;
                  config false;
            }
            leaf numberofwavelengths {
                  type int32;
                  config false;
            }
```

```
            list wavelength {
                  key waveid;
                  leaf waveid {
                        type int32;
                        config false;
                  }
                  leaf channel {
                        type string;
                        config false;
                  }
            }
      }
      list flow {
            key channel;
            leaf channel {
                  type string;
            }
            leaf inportid {
                  type int32;
            }
            leaf outportid {
                  type int32;
            }
      }
}
rpc show-node-properties {
      output {
            leaf nodeid {
                  type string;
            }
            leaf nodeip {
                  type string;
            }
            leaf numberofports {
                  type int32;
            }
      }
}
rpc show-active-ports {
      output {
                  list interfaces {
                        key portid;
                        leaf portid {
                              type uint32;
                        }
                        leaf numberofwavelengths {
                              type int32;
                        }
                        list wavelength {
                              key waveid;
                              leaf waveid {
                                    type int32;
                              }
                              leaf channel {
                                    type string;
                              }
                        }
                  }
      }
}
```

```
rpc show-wavelengths-status {
       input {
              leaf portid {
                     type uint32;
                     description
                     "port which wavelength-status will be shown";
              }
       }
       output {
                     list wavelengths {
                            key waveid;
                            leaf waveid {
                                   type int32;
                                   config false;
                            }
                            leaf channel {
                                   type string;
                                   config false;
                            }
                     }
       }
}
rpc show-channel-status {
       input {
              leaf portid {
                     type uint32;
              }
              leaf waveid {
                     type int32;
                     description
                     "wavelength which channel-status will be shown";
              }
       }
       output {
              leaf channel {
                     type string;
              }
       }
}
rpc cross-connect {
       input {
              leaf inportid {
                     type int32;
                     description
                     "input port to be cross-connected";
              }
              leaf channel {
                     type string;
                     description
                     "input channel to be cross-connected";
              }
              leaf outportid {
                     type int32;
                     description
                     "output port to be cross-connected";
              }
        }
}
}
```

VI. OTPNODE Implementation Modules

The classes used for the implementation of the OPTNODE application are the OptnodeProvider, OptnodeRPCImpl, OptnodeListener, OptnodeConsumer and OptnodeOFFeaturesHandler, the code developed for each is the following:

Code 2 - OptnodeProvider.java

```
package org.opendaylight.optnode.impl;

import org.opendaylight.controller.md.sal.binding.api.DataBroker;
import org.opendaylight.controller.md.sal.binding.api.DataChangeListener;
import
org.opendaylight.controller.md.sal.common.api.data.AsyncDataBroker.DataChan
geScope;
import
org.opendaylight.controller.md.sal.common.api.data.AsyncDataChangeEvent;
import
org.opendaylight.controller.md.sal.common.api.data.LogicalDatastoreType;
import org.opendaylight.controller.sal.binding.api.BindingAwareBroker;
import
org.opendaylight.controller.sal.binding.api.BindingAwareBroker.ProviderCont
ext;
import org.opendaylight.controller.sal.binding.api.BindingAwareProvider;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.flow.service.rev130819.SalFlo
wService;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.OptnodeProperties;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.OptnodeService;
import org.opendaylight.yangtools.concepts.ListenerRegistration;
import org.opendaylight.yangtools.yang.binding.DataObject;
import org.opendaylight.yangtools.yang.binding.InstanceIdentifier;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;


public class OptnodeProvider implements BindingAwareProvider,
DataChangeListener, AutoCloseable {

    private static final Logger LOG =
LoggerFactory.getLogger(OptnodeProvider.class);

    private BindingAwareBroker.RpcRegistration<OptnodeService>
optnodeService;

    private ListenerRegistration<DataChangeListener> dcReg;

    public static DataBroker dataService;
    public static SalFlowService flowProgrammer;
    public static final InstanceIdentifier<OptnodeProperties> OPTNODE_IDD =
InstanceIdentifier.builder(OptnodeProperties.class).build();
```

```
@Override
    public void onSessionInitiated(ProviderContext session) {
      OptnodeProvider.dataService =
session.getSALService(DataBroker.class);
      dcReg =
dataService.registerDataChangeListener(LogicalDatastoreType.CONFIGURATION,
OPTNODE_IDD, new OptnodeListener(), DataChangeScope.SUBTREE);
      OptnodeProvider.flowProgrammer =
session.getRpcService(SalFlowService.class);

      optnodeService = session.addRpcImplementation(OptnodeService.class,
new OptnodeRPCImpl());

        LOG.info("OptnodeProvider Session Initiated");
    }

@Override
    public void close() throws Exception {

        LOG.info("OptnodeProvider Registrations Closed");
        dcReg.close();
        optnodeService.close();
    }

      @Override
      public void onDataChanged(
                    AsyncDataChangeEvent<InstanceIdentifier<?>, DataObject>
change) {
              // TODO Auto-generated method stub
      }
}
```

Code 3 - OptnodeRPCImpl.java

```
package org.opendaylight.optnode.impl;

import java.nio.ByteBuffer;
import java.util.concurrent.Future;
import java.util.ArrayList;
import java.util.List;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import
org.opendaylight.controller.md.sal.common.api.data.LogicalDatastoreType;
import org.opendaylight.openflowjava.util.ByteBufUtils;
import org.opendaylight.optnode.impl.util.ITUGridC;
import
org.opendaylight.yang.gen.v1.urn.ietf.params.xml.ns.yang.ietf.inet.types.re
v100924.Uri;
import
org.opendaylight.yang.gen.v1.urn.ietf.params.xml.ns.yang.ietf.yang.types.re
v100924.MacAddress;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.action.types.rev131112.action
.list.Action;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.action.types.rev131112.action
.action.OutputActionCase;
```

```
import
org.opendaylight.yang.gen.v1.urn.opendaylight.action.types.rev131112.action
.action.OutputActionCaseBuilder;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.action.types.rev131112.action
.action.output.action._case.OutputAction;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.action.types.rev131112.action
.action.output.action._case.OutputActionBuilder;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.action.types.rev131112.action
.list.ActionBuilder;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.flow.service.rev130819.AddFlo
wInput;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.flow.service.rev130819.AddFlo
wInputBuilder;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.flow.types.rev131026.flow.Ins
tructionsBuilder;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.flow.types.rev131026.flow.Mat
chBuilder;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.flow.types.rev131026.instruct
ion.instruction.ApplyActionsCase;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.flow.types.rev131026.instruct
ion.instruction.ApplyActionsCaseBuilder;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.flow.types.rev131026.instruct
ion.instruction.apply.actions._case.ApplyActions;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.flow.types.rev131026.instruct
ion.instruction.apply.actions._case.ApplyActionsBuilder;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.flow.types.rev131026.instruct
ion.list.Instruction;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.flow.types.rev131026.instruct
ion.list.InstructionBuilder;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.inventory.rev130819.NodeConne
ctorId;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.model.match.types.rev131026.e
thernet.match.fields.EthernetDestinationBuilder;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.model.match.types.rev131026.m
atch.EthernetMatchBuilder;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.OptnodeService;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.OptnodeProperties;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.CrossConnectInput;
```

```
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.ShowActivePortsOutput;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.ShowActivePortsOutputBuilder;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.ShowChannelStatusInput;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.ShowChannelStatusOutput;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.ShowChannelStatusOutputBuilder;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.ShowNodePropertiesOutput;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.ShowNodePropertiesOutputBuilder;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.ShowWavelengthsStatusInput;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.ShowWavelengthsStatusOutput;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.ShowWavelengthsStatusOutputBuilder;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.optnode.properties.Interface;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.optnode.properties._interface.Wavelength;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.optnode.properties.Flow;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.optnode.properties.FlowBuilder;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.optnode.properties.InterfaceKey;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.show.active.ports.output.Interfaces;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.show.active.ports.output.InterfacesBuilder;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.show.wavelengths.status.output.Wavelengths;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.show.wavelengths.status.output.WavelengthsBuilder;
import org.opendaylight.yangtools.yang.common.RpcResult;
import org.opendaylight.yangtools.yang.common.RpcResultBuilder;
import org.opendaylight.controller.md.sal.binding.api.ReadOnlyTransaction;
```

```
import com.google.common.base.Optional;
import com.google.common.util.concurrent.FutureCallback;
import com.google.common.util.concurrent.Futures;
import com.google.common.util.concurrent.ListenableFuture;

public class OptnodeRPCImpl implements OptnodeService {

	private static final Logger LOG1 =
LoggerFactory.getLogger(OptnodeRPCImpl.class);
	public static List<Flow> OPTNODE_FLOWS = new ArrayList<Flow>();
	public static int found = 0;

	@Override
	public Future<RpcResult<ShowNodePropertiesOutput>>
showNodeProperties() {
		LOG1.info("Show Node Properties!!! ");

		ReadOnlyTransaction readTx =
OptnodeProvider.dataService.newReadOnlyTransaction();
		ListenableFuture<Optional<OptnodeProperties>> dataFuture = null;
		dataFuture = readTx.read(LogicalDatastoreType.OPERATIONAL,
OptnodeProvider.OPTNODE_IDD);
		final ShowNodePropertiesOutputBuilder propBuilder = new
ShowNodePropertiesOutputBuilder();
		Futures.addCallback(dataFuture, new
FutureCallback<Optional<OptnodeProperties>>() {

			@Override
			public void onSuccess(Optional<OptnodeProperties>
result) {
				if(result.isPresent()){
					OptnodeProperties opt1 = result.get();
					propBuilder.setNodeid(opt1.getNodeid());
					propBuilder.setNodeip(opt1.getNodeip());

	propBuilder.setNumberofports(opt1.getNumberofports());
					LOG1.info("SUCCESS");
					LOG1.info("ID: "+opt1.getNodeid()+"  IP:
"+opt1.getNodeip()+"  Ports: "+opt1.getNumberofports());
				}
				else{
					LOG1.info("No Data on Datastore");
				}
			}
	@Override
			public void onFailure(Throwable t) {
				LOG1.info("FAILURE");
			}
		});
		return
	RpcResultBuilder.success(propBuilder.build()).buildFuture();
	}

	@Override
	public Future<RpcResult<ShowActivePortsOutput>> showActivePorts() {
		LOG1.info("Show Active Ports!!! ");
		ReadOnlyTransaction readTx =
OptnodeProvider.dataService.newReadOnlyTransaction();
		ListenableFuture<Optional<OptnodeProperties>> dataFuture2 =
readTx.read(LogicalDatastoreType.OPERATIONAL, OptnodeProvider.OPTNODE_IDD);
```

```
        final ShowActivePortsOutputBuilder actBuilder = new
ShowActivePortsOutputBuilder();
        Futures.addCallback(dataFuture2, new
FutureCallback<Optional<OptnodeProperties>>() {

                    @Override
                    public void onSuccess(Optional<OptnodeProperties>
result) {
                          if(result.isPresent()){
                                OptnodeProperties opt2 = result.get();

                                List<Interface> listOfInterfaces =
opt2.getInterface();
                                List<Interfaces> outList = new
ArrayList<Interfaces>();

                                for(Interface i : listOfInterfaces){
                                      List<Wavelength> listOfWavelengths =
i.getWavelength();

      List<org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yan
g.optnode.rev150105.show.active.ports.output.interfaces.Wavelength>
outList2 = new
ArrayList<org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.
optnode.rev150105.show.active.ports.output.interfaces.Wavelength>();
                                      for(Wavelength w :
listOfWavelengths){
                                            outList2.add(new
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.show.active.ports.output.interfaces.WavelengthBuilder().setWaveid(w
.getWaveid()).setChannel(w.getChannel()).build());
                                      }
                                      outList.add(new
InterfacesBuilder().setPortid(i.getPortid()).setWavelength(outList2).build(
));
                                }
                                actBuilder.setInterfaces(outList);
                                LOG1.info("SUCCESS");
                                LOG1.info("Ports: "+opt2.getInterface());
                          }
                          else{
                                LOG1.info("No Data on Datastore");
                          }
                    }
              @Override
                    public void onFailure(Throwable t) {
                          LOG1.info("FAILURE");
                    }
        });
              return
RpcResultBuilder.success(actBuilder.build()).buildFuture();
      }

      @Override
      public Future<RpcResult<ShowWavelengthsStatusOutput>>
showWavelengthsStatus(
                        final ShowWavelengthsStatusInput input) {
              LOG1.info("Show Wavelengths of Port: {}", input);

              InterfaceKey key1 = new InterfaceKey(input.getPortid());
```

```
            ReadOnlyTransaction readTx =
OptnodeProvider.dataService.newReadOnlyTransaction();
            ListenableFuture<Optional<Interface>> dataFuture3 =
readTx.read(LogicalDatastoreType.OPERATIONAL,
OptnodeProvider.OPTNODE_IDD.child(Interface.class, key1));
            final ShowWavelengthsStatusOutputBuilder waveBuilder = new
ShowWavelengthsStatusOutputBuilder();
            Futures.addCallback(dataFuture3, new
FutureCallback<Optional<Interface>>() {

                  @Override
                  public void onSuccess(Optional<Interface> result) {
                        if(result.isPresent()){
                              Interface opt3 = result.get();
                              List<Wavelength> listOfWavelengths =
opt3.getWavelength();
                              List<Wavelengths> outList = new
ArrayList<Wavelengths>();

                              for (Wavelength w : listOfWavelengths){
                                    outList.add(new
WavelengthsBuilder().setWaveid(w.getWaveid()).setChannel(w.getChannel()).bu
ild());
                              }
                              waveBuilder.setWavelengths(outList);
                              LOG1.info("SUCCESS");
                              LOG1.info("Wavelengths: "+
opt3.getWavelength());
                        }
                        else{
                              LOG1.info("No Data on Datastore");
                        }
                  }
                  @Override
                  public void onFailure(Throwable t) {
                        LOG1.info("FAILURE");
                  }
        });
            return
RpcResultBuilder.success(waveBuilder.build()).buildFuture();
      }
      @Override
      public Future<RpcResult<ShowChannelStatusOutput>> showChannelStatus(
                  final ShowChannelStatusInput input) {
            LOG1.info("Show Channels of Wavelength/Port: {}", input);

            InterfaceKey key1 = new InterfaceKey(input.getPortid());

            ReadOnlyTransaction readTx =
OptnodeProvider.dataService.newReadOnlyTransaction();
            ListenableFuture<Optional<Interface>> dataFuture4 =
readTx.read(LogicalDatastoreType.OPERATIONAL,
OptnodeProvider.OPTNODE_IDD.child(Interface.class, key1));
            final ShowChannelStatusOutputBuilder channelBuilder = new
ShowChannelStatusOutputBuilder();
Futures.addCallback(dataFuture4, new FutureCallback<Optional<Interface>>()
{
                  @Override
                  public void onSuccess(Optional<Interface> result) {
                        if(result.isPresent()){
```

```
                                    Interface opt4 = result.get();
                                    List<Wavelength> listOfWavelengths =
opt4.getWavelength();
                                    for (Wavelength w : listOfWavelengths){
                                          if(w.getWaveid() ==
input.getWaveid()){

      channelBuilder.setChannel(w.getChannel());
                                                LOG1.info("SUCCESS");
                                                LOG1.info("Channel: "+
w.getChannel());
                                                break;
                                          }
                                    }
                              }
                              else{
                                    LOG1.info("No Data on Datastore");
                              }
                        }
                        @Override
                        public void onFailure(Throwable t) {
                              LOG1.info("FAILURE");
                        }
        });
            return
RpcResultBuilder.success(channelBuilder.build()).buildFuture();
      }

      @Override
      public Future<RpcResult<Void>> crossConnect(CrossConnectInput input)
{
            LOG1.info("CrossConnect Channel/Wavelength/Ports: {}", input);
            if (input.getOutportid() == input.getInportid()){
                  LOG1.warn("Output Port cannot be the same as Input
Port");
                  return Futures.immediateFuture(RpcResultBuilder.<Void>
failed().build());
            }
            if (this.checkWave(input.getInportid(), input.getChannel()) ==
            false){
                  LOG1.warn("Wavelenght not supported by Input port: " +
input.getInportid());
                  return Futures.immediateFuture(RpcResultBuilder.<Void>
failed().build());
            }
            OptnodeRPCImpl.found = 0;
            if (this.checkWave(input.getOutportid(), input.getChannel())
== false){
                  LOG1.warn("Wavelenght not supported by Output port: " +
input.getOutportid());
                  return Futures.immediateFuture(RpcResultBuilder.<Void>
failed().build());
            }
OptnodeRPCImpl.found = 0;

            MatchBuilder match = new MatchBuilder();
            EthernetMatchBuilder mac = new EthernetMatchBuilder();
            EthernetDestinationBuilder eth = new
EthernetDestinationBuilder();
```

```
            InstructionsBuilder ins = new InstructionsBuilder();
            List <Instruction> listofIns = new ArrayList<Instruction>();
            InstructionBuilder ins1 = new InstructionBuilder();
            ApplyActionsCaseBuilder aaa = new ApplyActionsCaseBuilder();
            ApplyActionsBuilder apply = new ApplyActionsBuilder();
            List <Action> listofAct = new ArrayList<Action>();
            ActionBuilder action = new ActionBuilder();
            OutputActionCaseBuilder out = new OutputActionCaseBuilder();
            OutputActionBuilder a = new OutputActionBuilder();

            NodeConnectorId nodeid =
NodeConnectorId.getDefaultInstance(input.getInportid().toString());
            LOG1.info("INPUT PORT ID: " + nodeid.getValue());

            Uri u =
Uri.getDefaultInstance(input.getOutportid().toString());
            a.setOutputNodeConnector(u);

            OutputAction action1 = a.build();

out.setOutputAction(action1);

            int akey = 1;
            int lkey = 1;

            OutputActionCase case1 = out.build();
            action.setAction(case1);
            action.setOrder(akey);
            Action action11 = action.build();
            listofAct.add(action11);
            apply.setAction(listofAct);
            ApplyActions apply1 = apply.build();
            aaa.setApplyActions(apply1);
            ApplyActionsCase apply2 = aaa.build();
            ins1.setInstruction(apply2);
            ins1.setOrder(lkey);
            Instruction instruction1 = ins1.build();
            listofIns.add(instruction1);

      /* ----------------------PARSE MAC---------------------------*/

            String channel1 = input.getChannel();
            double ch1 = Double.parseDouble(channel1);

            Long macBitmap = (long)0;
            ITUGridC itu = new ITUGridC();
            itu.initialize100G();
            macBitmap = itu.setLambdaByWave(macBitmap, ch1);
            byte[] bitmac = ByteBuffer.allocate(8).putLong(macBitmap >>
10).array();
            byte [] macBytes = new byte[6];
            for (int i=0; i<6; i++)
                  macBytes[i] = bitmac[i+2];

            String macSpace = ByteBufUtils.bytesToHexString(macBytes);
            String macStr = macSpace.replace(' ', ':');
            MacAddress macX = MacAddress.getDefaultInstance(macStr);

      /* ----------------------PARSE MAC---------------------------*/
```

```
            eth.setAddress(macX);
            mac.setEthernetDestination(eth.build());
            match.setEthernetMatch(mac.build());
            match.setInPort(nodeid);
            ins.setInstruction(listofIns);
            AddFlowInputBuilder flow = new AddFlowInputBuilder();
            flow.setNode(OptnodeOFFeaturesHandler.OPTNODE_NODEREF);
            flow.setMatch(match.build());
            flow.setInstructions(ins.build());
            AddFlowInput flow1 = flow.build();
            LOG1.info("FLOW ADDED:    "  + flow1.toString());
            OptnodeProvider.flowProgrammer.addFlow(flow1);

            OPTNODE_FLOWS.add(new
FlowBuilder().setChannel(input.getChannel()).setInportid(input.getInportid(
)).setOutportid(input.getOutportid()).build());

            OptnodeConsumer.initOptnodeConfig();
            return Futures.immediateFuture(RpcResultBuilder.<Void>
      success().build());
      }

      private boolean checkWave (int portid, final String channel){
            InterfaceKey key1 = new InterfaceKey((long)portid);
            ReadOnlyTransaction readTx =
OptnodeProvider.dataService.newReadOnlyTransaction();
            ListenableFuture<Optional<Interface>> dataFuture5 =
readTx.read(LogicalDatastoreType.OPERATIONAL,
OptnodeProvider.OPTNODE_IDD.child(Interface.class, key1));

            Futures.addCallback(dataFuture5, new
FutureCallback<Optional<Interface>>() {

                  @Override
                  public void onSuccess(Optional<Interface> result) {
                        if(result.isPresent()){
                              Interface opt4 = result.get();
                              List<Wavelength> listOfWavelengths =
opt4.getWavelength();
                              for (Wavelength w : listOfWavelengths)
                              {
                                    if (channel.equals(w.getChannel())){
                                          OptnodeRPCImpl.found = 1;
                                          break;
                                    }
                              }
                        }
                  }
                              List<Wavelength> listOfWavelengths =
                        opt4.getWavelength();
                              for (Wavelength w : listOfWavelengths)
                              {
                                    if (channel.equals(w.getChannel())){
                                          OptnodeRPCImpl.found = 1;
                                          break;
                                    }
                              }
                        }
                  }
```

```
		@Override
			public void onFailure(Throwable t) {
				LOG1.info("FAILURE");
				OptnodeRPCImpl.found = 0;
			}
		});

		if (OptnodeRPCImpl.found == 1){
			return true;
		}
		else{
			return false;
		}
	}
}
```

Code 4 - OptnodeListener.java

```
package org.opendaylight.optnode.impl;

import
org.opendaylight.controller.md.sal.common.api.data.AsyncDataChangeEvent;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.OptnodeProperties;
import org.opendaylight.yangtools.yang.binding.DataObject;
import org.opendaylight.yangtools.yang.binding.InstanceIdentifier;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class OptnodeListener extends OptnodeProvider {

	private static final Logger LOG3 =
LoggerFactory.getLogger(OptnodeListener.class);

	@Override
	public void onDataChanged(
			AsyncDataChangeEvent<InstanceIdentifier<?>, DataObject>
change) {
		DataObject dataObject = change.getUpdatedSubtree();

		if(dataObject instanceof OptnodeProperties){
			OptnodeProperties optnodeProperties =
(OptnodeProperties) dataObject;
			LOG3.info("onDataChanged - new Optnode config: {}",
optnodeProperties);
		}
		else{
			LOG3.warn("onDataChanged - not instance of Optnode {}",
dataObject);
		}

	}
}
```

Code 5 - OptnodeConsumer.java

```
package org.opendaylight.optnode.impl;

import org.opendaylight.controller.md.sal.binding.api.DataBroker;
import org.opendaylight.controller.md.sal.binding.api.WriteTransaction;
import
org.opendaylight.controller.md.sal.common.api.data.LogicalDatastoreType;
import
org.opendaylight.controller.sal.binding.api.BindingAwareBroker.ConsumerCont
ext;
import org.opendaylight.controller.sal.binding.api.BindingAwareConsumer;
import org.opendaylight.controller.sal.binding.api.NotificationService;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.OptnodeProperties;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.OptnodePropertiesBuilder;
import org.opendaylight.yangtools.concepts.ListenerRegistration;
import org.opendaylight.yangtools.yang.binding.NotificationListener;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class OptnodeConsumer implements BindingAwareConsumer, AutoCloseable
{

	private static final Logger LOG4 =
LoggerFactory.getLogger(OptnodeConsumer.class);
	public static DataBroker dataService2;
	private ListenerRegistration<NotificationListener> optListener;
	public static NotificationService notificationService;

	@Override
	public void onSessionInitialized(ConsumerContext session) {

		OptnodeConsumer.notificationService =
session.getSALService(NotificationService.class);
		optListener =
notificationService.registerNotificationListener(new
OptnodeOFFeaturesHandler());
		OptnodeConsumer.dataService2 =
session.getSALService(DataBroker.class);

		LOG4.info("OptnodeConsumer Session Initiated");
	}
	public static void initOptnodeOperational(){
	OptnodeProperties optnode = new
OptnodePropertiesBuilder().setNodeid(OptnodeOFFeaturesHandler.OPTNODE_ID).s
etNodeip(OptnodeOFFeaturesHandler.OPTNODE_IP).setNumberofports(OptnodeOFFea
turesHandler.OPTNODE_NUMPORTS).setInterface(OptnodeOFFeaturesHandler.OPTNOD
E_INTERFACES).build();
	WriteTransaction tx = dataService2.newWriteOnlyTransaction();
	tx.put(LogicalDatastoreType.OPERATIONAL, OptnodeProvider.OPTNODE_IDD,
optnode);
	tx.submit();
	LOG4.info("init Optnode Operational: Operational Status Populated:
{}", optnode);
    }
```

```
	public static void initOptnodeConfig(){
		OptnodeProperties optnode = new
OptnodePropertiesBuilder().setFlow(OptnodeRPCImpl.OPTNODE_FLOWS).build();
		WriteTransaction tx = dataService2.newWriteOnlyTransaction();
		tx.put(LogicalDatastoreType.CONFIGURATION,
OptnodeProvider.OPTNODE_IDD, optnode);
		tx.submit();
		LOG4.info("init Optnode Config: Config Status Populated: {}",
optnode);
	}

	public static void eraseNode(){
		WriteTransaction tx = dataService2.newWriteOnlyTransaction();
		tx.delete(LogicalDatastoreType.OPERATIONAL,
OptnodeProvider.OPTNODE_IDD);
		tx.delete(LogicalDatastoreType.CONFIGURATION,
OptnodeProvider.OPTNODE_IDD);
		tx.submit();
		OptnodeRPCImpl.OPTNODE_FLOWS.clear();
		LOG4.info("NODE REMOVED: Erasing CONFIGURATION & OPERATIONAL
Datastores");

	}
	@Override
	public void close() throws Exception {
		optListener.close();
	}
}
```

Code 6 - OptnodeOFFeaturesHandler.java

```
package org.opendaylight.optnode.impl;

import java.nio.ByteBuffer;
import java.util.ArrayList;
import java.util.List;
import org.opendaylight.openflowjava.util.ByteBufUtils;
import org.opendaylight.optnode.impl.util.ITUGridC;
import
org.opendaylight.yang.gen.v1.urn.ietf.params.xml.ns.yang.ietf.yang.types.re
v100924.MacAddress;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.flow.inventory.rev130819.Flow
CapableNodeConnectorUpdated;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.flow.inventory.rev130819.Flow
CapableNodeUpdated;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.inventory.rev130819.NodeConne
ctorRemoved;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.inventory.rev130819.NodeConne
ctorUpdated;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.inventory.rev130819.NodeRef;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.inventory.rev130819.NodeRemov
ed;
```

```
import
org.opendaylight.yang.gen.v1.urn.opendaylight.inventory.rev130819.NodeUpdat
ed;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.inventory.rev130819.Opendayli
ghtInventoryListener;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.optnode.properties.Interface;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.optnode.properties.InterfaceBuilder;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.optnode.properties._interface.Wavelength;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.optnode.re
v150105.optnode.properties._interface.WavelengthBuilder;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class OptnodeOFFeaturesHandler implements
OpendaylightInventoryListener{

      private static final Logger LOG5 =
LoggerFactory.getLogger(OptnodeOFFeaturesHandler.class);

    public static NodeRef OPTNODE_NODEREF = null;
    public static MacAddress OPTNODE_MAC = null;
      public static String OPTNODE_ID = null;
    public static String OPTNODE_IP = null;
    public static Integer OPTNODE_NUMPORTS = 0;
    public static List<Interface> OPTNODE_INTERFACES = new
ArrayList<Interface>();

      @Override
      public void onNodeConnectorRemoved(NodeConnectorRemoved
notification) {
            LOG5.info("Node Connector Removed!!   " +
notification.toString());

            OPTNODE_NUMPORTS--;
      }
      @Override
      public void onNodeConnectorUpdated(NodeConnectorUpdated
notification) {
            LOG5.info("Node Connector Updated!!   " +
notification.getId().getValue());

            FlowCapableNodeConnectorUpdated flowNode2 =
notification.getAugmentation(FlowCapableNodeConnectorUpdated.class);
            String macStr = flowNode2.getHardwareAddress().getValue();
            OPTNODE_MAC = flowNode2.getHardwareAddress();

            String macSpace = macStr.replace(':', ' ');
            byte[] macByte = ByteBufUtils.hexStringToBytes(macSpace);
            byte[] tmpMac = new byte[8];
            for (int i=0; i<6; i++)
                  tmpMac[i+2] = macByte[i];
```

```
            ByteBuffer b = ByteBuffer.allocate(8).put(tmpMac, 0,
tmpMac.length);
            Long macBitmap = b.getLong(0);
            macBitmap = macBitmap << 10;
            ITUGridC itu = new ITUGridC();
            itu.initialize100G();
            List<Wavelength> OPTNODE_WAVE = new ArrayList<Wavelength>();

            int pos = 0;
            int waveid = 1;
            macBitmap = macBitmap >> 10;
            while (macBitmap != 0)
            {
                  if ((macBitmap & 0x1) == 1)
                  {
                        String strWl =
String.valueOf(itu.getItuGridW()[pos]);
                        LOG5.info("WAVELENGTH: " + strWl + "  ON PORT: "
+ flowNode2.getPortNumber().getUint32());
                        OPTNODE_WAVE.add(new
WavelengthBuilder().setChannel(strWl).setWaveid(waveid).build());
                        waveid++;
                  }
                  macBitmap = macBitmap >> 1;
                  pos++;
            }

            OPTNODE_INTERFACES.add(new
InterfaceBuilder().setPortid(flowNode2.getPortNumber().getUint32()).setNumb
erofwavelengths(waveid-1).setWavelength(OPTNODE_WAVE).build());
            OPTNODE_NUMPORTS++;
      }

      @Override
      public void onNodeRemoved(NodeRemoved notification) {

            LOG5.info("NODE : " + OPTNODE_ID + " HAS BEEN REMOVED");

            OptnodeConsumer.eraseNode();
            OPTNODE_INTERFACES.clear();
            OPTNODE_NUMPORTS = 0;
      }

      @Override
      public void onNodeUpdated(NodeUpdated notification) {
            LOG5.info("Node Updated!!   " +
notification.getId().getValue());

            FlowCapableNodeUpdated flowNode =
notification.getAugmentation(FlowCapableNodeUpdated.class);
            if (flowNode != null && flowNode.getIpAddress() != null){
                  OPTNODE_IP =
flowNode.getIpAddress().getIpv4Address().getValue();
                  OPTNODE_ID = notification.getId().getValue();
                  OptnodeConsumer.initOptnodeOperational();
                  OptnodeConsumer.initOptnodeConfig();
            }
            OPTNODE_NODEREF = notification.getNodeRef();
      }
}
```

VII. OPT State Information File

The following xml code shows the state information “received” from the switch, that will be sent to the controller by means of a OFPT_FEATURES_REPLY message. The code is shown next:

Code 7 - OPT.xml

```
<node id="1">
	<ip>192.168.1.1</ip>
	<numPorts>2</numPorts>
	<port id="1">
		<numWavelengths>2</numWavelengths>
		<wavelength id="1">
			<channel>1524.11</channel>
		</wavelength>
		<wavelength id="2">
			<channel>1561.42</channel>
		</wavelength>
	</port>
	<port id="2">
		<numWavelengths>2</numWavelengths>
		<wavelength id="1">
			<channel>1524.11</channel>
		</wavelength>
		<wavelength id="2">
			<channel>1561.42</channel>
		</wavelength>
	</port>
</node>
```

VIII. Method of Wavelength State-Info Encapsulation

The encapsulation of the supported wavelengths into a Mac Address in order to be sent through OpenFlow messages is done by using a reference ITU Grid utility library that contains all the available wavelengths used for optical communications. The grid code is shown next:

Code 8 - Wavelengths ITU Grid

```
public void initialize100G ()
	{
		double[] tmpWave = {1524.11, 1524.89, 1525.66, 1526.44,
1527.22, 1527.99, 1528.77, 1529.55, 1530.33, 1531.12, 1531.90, 1532.68,
1533.47, 1534.25, 1535.04, 1535.82, 1536.61, 1537.40, 1538.19, 1538.98,
1539.77, 1540.56, 1541.35, 1542.14, 1542.94, 1543.73, 1544.53, 1545.32,
1546.12, 1546.92, 1547.72, 1548.51, 1549.32, 1550.12, 1550.92, 1551.72,
1552.52, 1553.33, 1554.13, 1554.94, 1555.75, 1556.55, 1557.36, 1558.17,
1558.98, 1559.79, 1560.61, 1561.42, 1562.23, 1563.05, 1563.86, 1564.68,
1565.50, 1566.33};

		this.ituGridW = tmpWave;
	}
```

The idea for the encapsulation is based on comparing the wavelength to be encapsulated with the WaveGrid seen above and using an empty 8 byte Bitmap to set a single bit inside the Bitmap that will represent the selected Wavelength. The function that provides this setup is the following:

Code 9 - Set-Wavelength Function

```
public Long setLambdaByWave (Long bitmap, double wl)
	{
		int i = 0;
		while (i<ITUGrid.GRID_SIZE)
		{
			if (ituGridW[i] == wl)
				break;
			i++;
		}
		long mask = (long)1 << (i + 10);

		return bandwidth | mask;
	}
```

As the Mac Address field only contains 6 bytes (48 bits), then only 48 wavelengths of the total of 54 available ones can be encapsulated at once. Depending on the optical node, a fine tuning of the required wavelengths to be supported can be performed.

In order to accommodate the 6 bytes to be used for the wavelengths into the original 8 byte Bitmap that is required by the function, the bits that correspond to the wavelength encoding are accommodated at the last 6 bytes of the Bitmap 8 byte array, so then this value can be mapped to another array that contains only 6 bytes, by means of the next function:

Code 10 - 8-to-6 Byte array conversion

```
byte [] macBytes = new byte[6];
            for (int i=0; i<6; i++)
                  macBytes[i] = bitmac[i+2];
```

Fynally, this 6 macBytes byte array can be converted from Bytes to String so the “ : ” character could be added in order to accommodate the data to the one required for the Mac Address field.

Code 11 - Byte to Mac Address conversion

```
String macSpace = ByteBufUtils.bytesToHexString(macBytes);
            String macStr = macSpace.replace(' ', ':');
            MacAddress macX = MacAddress.getDefaultInstance(macStr);

            eth.setAddress(macX);
```

Once the Mac Address is in the correct format, it can be set as a port Ethernet Address or as a new flow’s Ethernet match in order to be sent inside an OFTP_FEATURES_REPLY or OFTP_FLOW_MOD message.

On the receiver side, the same encapsulation method logic can be used to de-encapsulate the wavelength information. In this case the received Mac Address can be converted to a 6 byte array, which would be afterwards mapped into an 8 byte Bitmap so the utility library can compare the bits with the Wavelength grid and return the correct wavelength values.

IX. OF Agent Implementation Modules

The main classes used for the implementation of the OPTNODE application are the OptAg, OptTCPClientInitializer and the OptScenarioHandler, the code developed for each is the following:

Code 12 - OptAg.java

```
package optagent;

import io.netty.bootstrap.Bootstrap;
import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelOption;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioDatagramChannel;
import io.netty.channel.socket.nio.NioSocketChannel;
import java.io.IOException;
import java.net.InetAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;
import java.util.ArrayList;
import java.util.List;
import java.util.Stack;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.LinkedBlockingQueue;
import optagent.node.Switch;
import optagent.utils.XMLParser;
import org.opendaylight.openflowjava.util.ByteBufUtils;
import org.openflow.example.cli.Options;
import org.openflow.example.cli.ParseException;
import org.openflow.example.cli.SimpleCLI;
import org.openflow.io.OFMessageAsyncStream;
import org.openflow.protocol.OFMessage;
import org.openflow.protocol.OFType;
import org.openflow.protocol.factory.BasicFactory;
import com.google.common.util.concurrent.SettableFuture;

public class OptAg implements OFClient
{
    protected ExecutorService es;
    protected BasicFactory factory;
    protected SocketChannel serverSock;
    protected Integer threadCount;
    protected int port = 6633;
    protected OFMessageAsyncStream stream;
    private OFListener ofListener;
    protected Switch ofSwitch = null;
    private final String host = "172.26.37.209";
    private EventLoopGroup group;
    private SettableFuture<Boolean> isOnlineFuture;
    private SettableFuture<Boolean> scenarioDone;
    private OptTcpClientInitializer clientInitializer;
    private OptScenarioHandler scenarioHandler;
```

```
    public Switch newOPSSwitch ()
    {
       Switch sw = new Switch();
       return sw;
    }

    public OptAg (int port) throws IOException
    {
       XMLParser xmlparser = new XMLParser();
       this.ofSwitch = xmlparser.readXML("conf/OPT.xml");
             if (this.ofSwitch == null)
                   System.out.println("ERROR!");
             else
             {
                   this.ofSwitch.setSwId(1);
                   System.out.println(this.ofSwitch.toString());
             }
    }

    public static void main(String [] args) throws IOException {
        SimpleCLI cmd = parseArgs(args);
        int port = Integer.valueOf(cmd.getOptionValue("p"));
        OptAg sc = new OptAg (port);
        sc.run();
    }

    public static SimpleCLI parseArgs(String[] args) {
        Options options = new Options();
        options.addOption("h", "help", "print help");
        options.addOption("p", "port", 6633, "the port to listen on");
        options.addOption("t", "threads", 1, "the number of threads to
run");
        try {
            SimpleCLI cmd = SimpleCLI.parse(options, args);
            if (cmd.hasOption("h")) {
                printUsage(options);
                System.exit(0);
            }
            return cmd;
        } catch (ParseException e) {
            System.err.println(e);
            printUsage(options);
        }

        System.exit(-1);
        return null;
    }

    public static void printUsage(Options options) {
        SimpleCLI.printHelp("Usage: "
                + OptAg.class.getCanonicalName() + " [options]",
                options);
    }
       @Override
       public void run() {
         group = new NioEventLoopGroup();
        clientInitializer = new OptTcpClientInitializer(isOnlineFuture);
        scenarioHandler = new OptScenarioHandler(new
/*Stack*/LinkedBlockingQueue<ClientEvent>(), this);
```

```
OFMessage ofHello = new OFMessage ();
        ofHello.setType(OFType.HELLO);
        ofHello.setLengthU(8);
        ofHello.setXid(1);
        ofHello.setVersion(OFMessage.OFP_VERSION);
        ByteBuffer data = ByteBuffer.allocate(8);
        ofHello.writeTo(data);
        scenarioHandler.getScenario().add(new SendEvent(data.array()));
        clientInitializer.setScenario(scenarioHandler);
        try {
            Bootstrap b = new Bootstrap();
            b.group(group)
                .channel(NioSocketChannel.class)
                .option(ChannelOption.SO_KEEPALIVE, true)
                .handler(clientInitializer);

            ChannelFuture f = b.connect(host, port).sync();
            System.out.println("Connected...");

            synchronized (scenarioHandler) {
                System.out.println("WAITING FOR SCENARIO");
                scenarioHandler.wait();
            }
            f.channel().closeFuture().sync();

        } catch (Exception ex) {
             System.out.println("Exception: " + ex.toString());
            System.out.println(ex.getMessage());
        } finally {
             System.out.println("shutting down");
            try {
                group.shutdownGracefully().get();
                System.out.println("shutdown succesful");
            } catch (InterruptedException | ExecutionException e) {
             System.out.println(e.getMessage());
            }
        }
        scenarioDone.set(true);
      }
      @Override
      public SettableFuture<Boolean> getIsOnlineFuture() {
            // TODO Auto-generated method stub
            return isOnlineFuture;
      }
      @Override
      public SettableFuture<Boolean> getScenarioDone() {
            // TODO Auto-generated method stub
            return scenarioDone;
      }
      @Override
      public void setScenarioHandler(OptScenarioHandler arg0) {
            // TODO Auto-generated method stub
            this.scenarioHandler = arg0;
      }
      @Override
      public void setSecuredClient(boolean arg0) {
            // TODO Auto-generated method stub

      }
}
```

Code 13 - OptTCPClientInitializer.java

```
package optagent;

import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelPipeline;
import io.netty.channel.socket.nio.NioSocketChannel;

import com.google.common.util.concurrent.SettableFuture;

public class OptTcpClientInitializer extends
ChannelInitializer<NioSocketChannel> {

    private SettableFuture<Boolean> isOnlineFuture;
    private OptScenarioHandler scenarioHandler;

    public OptTcpClientInitializer(SettableFuture<Boolean> isOnlineFuture)
{
        this.isOnlineFuture = isOnlineFuture;
    }

    @Override
    public void initChannel(NioSocketChannel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        OptClientHandler simpleClientHandler = new
OptClientHandler(isOnlineFuture, scenarioHandler);
        simpleClientHandler.setScenario(scenarioHandler);
        pipeline.addLast("framer", new OptClientFramer());
        pipeline.addLast("handler", simpleClientHandler);
        isOnlineFuture = null;
    }

    public void setScenario(OptScenarioHandler scenarioHandler) {
        this.scenarioHandler = scenarioHandler;
    }
}
```

Code 14 - OptScenarioHandler.java

```
package optagent;

import io.netty.channel.ChannelHandlerContext;
import java.nio.ByteBuffer;
import java.util.ArrayList;
import java.util.List;
import java.util.Stack;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.TimeUnit;
import optagent.node.Port;
import optagent.node.Wavelength;
import optagent.utils.ITUGridC;
import org.opendaylight.openflowjava.util.ByteBufUtils;
import org.openflow.protocol.OFEchoReply;
import org.openflow.protocol.OFFeaturesReply;
import org.openflow.protocol.OFFeaturesReply.OFCapabilities;
import org.openflow.protocol.OFFlowMod;
import org.openflow.protocol.OFGetConfigReply;
import org.openflow.protocol.OFMessage;
import org.openflow.protocol.OFPhysicalPort;
```

```
import org.openflow.protocol.OFStatisticsMessageBase;
import org.openflow.protocol.OFStatisticsReply;
import org.openflow.protocol.OFStatisticsRequest;
import org.openflow.protocol.OFType;
import org.openflow.protocol.action.OFActionType;
import org.openflow.protocol.statistics.OFPortStatisticsReply;
import org.openflow.protocol.statistics.OFStatistics;
import org.openflow.protocol.statistics.OFStatisticsType;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class OptScenarioHandler extends Thread {

    private static final Logger LOGGER =
LoggerFactory.getLogger(OptScenarioHandler.class);
    private BlockingQueue<ClientEvent> scenario;
    private BlockingQueue<OFMessage> ofMsg;
    private ChannelHandlerContext ctx;
    private int eventNumber;

    private OptAg agent;
    public OptScenarioHandler(/*Stack*/BlockingQueue<ClientEvent> scenario,
OptAg agent) {
        this.scenario = scenario;
        ofMsg = new LinkedBlockingQueue<>();
        this.agent = agent;
    }

    @Override
    public void run() {
      System.out.println("[run]");
        int freezeCounter = 0;
        boolean exit = false;
        while (!exit)
        {
        if (!scenario.isEmpty())
        {
             System.out.println("Scenario = " + scenario.size());
            System.out.println("Running event #" + eventNumber + " --------
------------------------------");
            ClientEvent peek = scenario.peek();
            OFMessage msg = null;
            if (peek instanceof WaitForMessageEvent)
            {
             WaitForMessageEvent wfme = (WaitForMessageEvent)peek;
             System.out.println("WaitForMessageEvent: " +
ByteBufUtils.bytesToHexString(wfme.headerExpected));
                try
                {
                    WaitForMessageEvent event = (WaitForMessageEvent) peek;
                    msg = ofMsg.poll(2000, TimeUnit.MILLISECONDS);
                    byte[] msgh = new byte[1];
                    msgh[0] = msg.getType().getTypeValue();
                    event.setHeaderReceived(msgh);
                }
                catch (InterruptedException e)
                {
                   System.out.println(e.getMessage() + " - " + e);
                    break;
                }
```

```
            }
            else if (peek instanceof SendEvent)
            {
             SendEvent event = (SendEvent) peek;
             System.out.println("Proceed - sendevent: " +
ByteBufUtils.bytesToHexString(event.msgToSend));
                event.setCtx(ctx);
            }
            if (peek.eventExecuted())
            {
             System.out.println("Pop");
                if (peek instanceof WaitForMessageEvent)
                {
                    WaitForMessageEvent e = (WaitForMessageEvent)peek;
                    parseOfMessage(msg);
                }
                scenario.poll();
                eventNumber++;
                freezeCounter = 0;
            }
            else
            {
                freezeCounter++;
            }
            if (freezeCounter > 2)
            {
             System.out.println("Scenario freezed: " + freezeCounter);
                break;
            }
            try
            {
                sleep(100);
            }
            catch (InterruptedException e)
            {
             System.out.println(e.getMessage() + " - " + e);
            }
        }
        else
        {
             try
            {
                sleep(5000);
            }
            catch (InterruptedException e)
            {
             System.out.println(e.getMessage() + " - " + e);
            }
             if (scenario.isEmpty())
                    exit = true;
        }
        }
        System.out.println("Scenario finished");
        synchronized (this)
        {
            this.notify();
        }
    }
```

```
public boolean isEmpty() {
        return scenario.isEmpty();
    }
    public BlockingQueue<ClientEvent> getScenario() {
        return scenario;
    }
    public void setScenario(BlockingQueue<ClientEvent> scenario) {
        this.scenario = scenario;
    }
    public void setCtx(ChannelHandlerContext ctx) {
        this.ctx = ctx;
    }
    public void addOfMsg (OFMessage msg)
    {
       ofMsg.add(msg);
        this.scenario.add(new WaitForMessageEvent(msg));
    }

    private void sendMsg (OFMessage msg)
    {
      msg.computeLength();
      System.out.println("[sendMsg] " + msg.getLengthU() + " bytes");
      ByteBuffer data = ByteBuffer.allocate(msg.getLengthU());
            msg.writeTo(data);
            this.scenario.add(new SendEvent(data.array()));
    }

    private void sendEchoReply (int xId)
    {
      OFEchoReply ofEchoRep = new OFEchoReply();
      ofEchoRep.setLengthU(8);
      ofEchoRep.setXid(xId);

      this.sendMsg(ofEchoRep);
    }
    private void sendFeaturesReply (int xId)
      {
      System.out.println("[sendFeaturesReply]");
            OFFeaturesReply ofFeatRep = new OFFeaturesReply();
            ofFeatRep.setXid(xId);
            // Switch
            ofFeatRep.setDatapathId((long)33);
            ofFeatRep.setBuffers(256);
            ofFeatRep.setTables((byte)255);
            // Ports
            List<OFPhysicalPort> ports = new ArrayList<OFPhysicalPort>();
List<Port> portList = this.agent.ofSwitch.getPortList();
            for (Port port : portList)
            {
                  OFPhysicalPort p = new OFPhysicalPort();
                  p.setPortNumber((short) port.getPortId());
                  p.setName("in" + port.getSwitch().getSwId() + ":" +
port.getPortId());
                  p.setConfig(0);
                  p.setPeerFeatures(0);

                  ITUGridC grid = new ITUGridC();
                  grid.initialize100G();
                  Long bitmap = (long) 0;
```

```
                    for (Wavelength w : port.getWlList())
                    {
                          bitmap = grid.setLambdaByWave(bitmap,
w.getChannel());
                    }
                    byte[] bitmac = ByteBuffer.allocate(8).putLong(bitmap
>> 10).array();
                    byte[] mac = new byte[6];
                    for (int i = 0; i<6; i++)
                          mac[5 - i] = bitmac[7 - i];
                    p.setHardwareAddress(mac);
                    ///
                    ports.add(p);
              }
              ofFeatRep.setPorts(ports);
              System.out.println("pList = "+ports.size());
              int capabilities = OFCapabilities.OFPC_PORT_STATS.getValue();
              ofFeatRep.setCapabilities(capabilities);
              // Actions
              int actions = (OFActionType.OUTPUT.getTypeValue()) +
(OFActionType.SET_VLAN_VID.getTypeValue()) +
(OFActionType.SET_NW_TOS.getTypeValue());
              ofFeatRep.setActions(actions);
              this.sendMsg(ofFeatRep);
        }
     private void sendGetConfigReply (int xId)
        {
              OFGetConfigReply ofGetConfigRep = new OFGetConfigReply();
              ofGetConfigRep.setXid(xId);
              ofGetConfigRep.setFlags((short)0);
              this.sendMsg(ofGetConfigRep);
        }
     private void sendPortStats (int xId)
        {
              AgentToSw AtS = new AgentToSw();
              OFStatisticsReply ofStatsRep = new OFStatisticsReply();
              ofStatsRep.setStatisticType(OFStatisticsType.PORT);
              ofStatsRep.setXid(xId);
              ofStatsRep.setFlags((short) 0);

              // PortStats
              List<OFStatistics> portStats = new ArrayList<>();
              List<Port> portList = this.agent.ofSwitch.getPortList();
              for (Port port : portList)
              {
                    long rx = 0, rtx = 0, tx = 0;
                    List<Wavelength> wlList = port.getWlList();
                    for (Wavelength wl : wlList)
                    {
                          // Get (Lambda) Counters
                          int counters[] =
AtS.operationGetCountersRet(AgentToSw.GET_PORT_COUNTERS, port.getPortId(),
wl.getId());
                          for (int n = 0; n < counters.length / 3; n++)
                          {
                                rx += counters[3 * n + 1];
                                rtx += counters[3 * n + 2];
                          }
                          tx = rx - rtx;
                    }
```

```
System.out.println("PORT-" + port.getPortId() + "\t" + tx + " / " + rx + "
/ " + rtx + "\n");
                    // Set Stats
                    OFPortStatisticsReply p = new OFPortStatisticsReply();
                    p.setPortNumber((short) port.getPortId());
                    //p.sets
                    p.setTransmitPackets(tx);
                    p.setTransmitBytes((long) (tx * AgentToSw.KBpOP));
                    p.setreceivePackets(rx);
                    p.setReceiveBytes((long) (rx * AgentToSw.KBpOP));
                    p.setCollisions(rtx);
                    portStats.add(p);
             }
             ofStatsRep.setStatistics(portStats);
             System.out.println("PStat " + portStats.size());
             this.sendMsg(ofStatsRep);
      }
    private void parseFlowMod (OFFlowMod ofFlowMod)
    {
      switch (ofFlowMod.getCommand())
      {
             case OFFlowMod.OFPFC_ADD:
                    System.out.println("----------------  NEW FLOW_MOD  ---
             ---------------");

                    System.out.println("MATCH  ------> " +
ofFlowMod.getMatch().toString());

                    List<OFAction> b = ofFlowMod.getActions();

                    OFActionOutput out = (OFActionOutput) b.get(0);

                    System.out.println("ACTION   ----> " + out.toString());
                    break;
             case OFFlowMod.OFPFC_DELETE_STRICT:
                    break;
             case OFFlowMod.OFPFC_DELETE:
                    // Delete all flows
                    break;
      }
    }
    private void parseStatsReq (OFStatisticsRequest ofStatsReq)
      {
             switch (ofStatsReq.getStatisticType())
             {
                    case PORT:
                           sendPortStats (ofStatsReq.getXid() + 1);
                           break;
                    case FLOW:
                           break;
                    default:
                           break;
             }
      }
    public void parseOfMessage (OFMessage ofMsg)
    {
      switch (ofMsg.getType())
      {
                    case HELLO:
                           break;
```

```
                    case ECHO_REQUEST:
                          sendEchoReply (ofMsg.getXid() + 1);
                          break;
                    case GET_CONFIG_REQUEST:
                          sendGetConfigReply (ofMsg.getXid() + 1);
                          break;
                    case FEATURES_REQUEST:
                          sendFeaturesReply (ofMsg.getXid() + 1);
                          break;
                    case FLOW_MOD:
                          parseFlowMod ((OFFlowMod)ofMsg);
                          System.out.println(ofMsg.toString());
                          break;
                    case BARRIER_REQUEST:
                          break;
                    case STATS_REQUEST:
                          sendPortStats(ofMsg.getXid() + 1);
                          break;
                    default:
                          System.out.println("Unsupported message " +
ofMsg.getType());
                          break;
      }
    }
}
```

X. OPTNODE Application Exporting Procedure

In order to export the OPTNODE application to the OpenFlow Plugin ODL distribution or to any available distribution, it is first necessary to download the distribution from git by using the following command from the Ubuntu console:

```
git clone https://git.opendaylight.org/gerrit/p/openflowplugin.git
```

Then, the maven build command can be used over the /openflowplugin folder in order to build the project:

```
mvn clean install
```

After the project is built, the Apache Karaf distribution of this project needs to be launched and then it is possible to install features corresponding to both openflowplugin and openflowjava projects, that will be used to interact with the OPTNODE application and the OF Agent.

```
feature:install odl-openflowplugin-all
feature:install odl-openflowjava-all
```

Once the features are installed, the OPTNODE project should be imported to the Apache Karaf distribution of the OpenFlow Plugin project by adding the correspondent repository as it can be seen next.

```
opendaylight-user@root>repo-add mvn:org.opendaylight.optnode/optnode-features/1.0-SNAPSHOT/xml/features
Adding feature url mvn:org.opendaylight.optnode/optnode-features/1.0-SNAPSHOT/xml/features
```

Figure 3 – Adding OPTNODE repository

The Apache Karaf will add the new feature to its repositories and then optnode-features will be included in the repositories-list as it can be seen next.

```
opendaylight-user@root>repo-list
Repository                                | URL
------------------------------------------------------------------------------------------------------------------------------------------
openflowplugin-li-0.2.0-SNAPSHOT          | mvn:org.opendaylight.openflowplugin/features-openflowplugin-li/0.2.0-SNAPSHOT/xml/features
spring-3.0.3                              | mvn:org.apache.karaf.features/spring/3.0.3/xml/features
odl-openflowjava-0.7.0-SNAPSHOT           | mvn:org.opendaylight.openflowjava/features-openflowjava/0.7.0-SNAPSHOT/xml/features
odl-protocol-framework-0.6.0-SNAPSHOT     | mvn:org.opendaylight.controller/features-protocol-framework/0.6.0-SNAPSHOT/xml/features
odl-mdsal-1.3.0-SNAPSHOT                  | mvn:org.opendaylight.controller/features-mdsal/1.3.0-SNAPSHOT/xml/features
enterprise-3.0.3                          | mvn:org.apache.karaf.features/enterprise/3.0.3/xml/features
odl-netconf-0.3.0-SNAPSHOT                | mvn:org.opendaylight.controller/features-netconf/0.3.0-SNAPSHOT/xml/features
odl-yangtools-0.8.0-SNAPSHOT              | mvn:org.opendaylight.yangtools/features-yangtools/0.8.0-SNAPSHOT/xml/features
odl-controller-1.2.0-SNAPSHOT             | mvn:org.opendaylight.controller/features-restconf/1.2.0-SNAPSHOT/xml/features
openflowplugin-0.2.0-SNAPSHOT             | mvn:org.opendaylight.openflowplugin/features-openflowplugin/0.2.0-SNAPSHOT/xml/features
odl-aaa-0.2.0-SNAPSHOT                    | mvn:org.opendaylight.aaa/features-aaa/0.2.0-SNAPSHOT/xml/features
odl-controller-1.3.0-SNAPSHOT             | mvn:org.opendaylight.controller/features-restconf/1.3.0-SNAPSHOT/xml/features
odl-dlux-0.3.0-SNAPSHOT                   | mvn:org.opendaylight.dlux/features-dlux/0.3.0-SNAPSHOT/xml/features
org.ops4j.pax.web-3.1.4                   | mvn:org.ops4j.pax.web/pax-web-features/3.1.4/xml/features
standard-3.0.3                            | mvn:org.apache.karaf.features/standard/3.0.3/xml/features
odl-optnode-1.0-SNAPSHOT                  | mvn:org.opendaylight.optnode/optnode-features/1.0-SNAPSHOT/xml/features
odl-config-persister-0.3.0-SNAPSHOT       | mvn:org.opendaylight.controller/features-config-persister/0.3.0-SNAPSHOT/xml/features
odl-aaa-0.2.0-SNAPSHOT                    | mvn:org.opendaylight.aaa/features-aaa-api/0.2.0-SNAPSHOT/xml/features
odl-yangtools-0.7.0-SNAPSHOT              | mvn:org.opendaylight.yangtools/features-yangtools/0.7.0-SNAPSHOT/xml/features
odl-config-0.3.0-SNAPSHOT                 | mvn:org.opendaylight.controller/features-config/0.3.0-SNAPSHOT/xml/features
odl-protocol-framework-0.7.0-SNAPSHOT     | mvn:org.opendaylight.controller/features-protocol-framework/0.7.0-SNAPSHOT/xml/features
odl-mdsal-1.2.0-SNAPSHOT                  | mvn:org.opendaylight.controller/features-mdsal/1.2.0-SNAPSHOT/xml/features
odl-netconf-0.4.0-SNAPSHOT                | mvn:org.opendaylight.controller/features-netconf/0.4.0-SNAPSHOT/xml/features
odl-config-0.4.0-SNAPSHOT                 | mvn:org.opendaylight.controller/features-config/0.4.0-SNAPSHOT/xml/features
odl-controller-1.6.0-SNAPSHOT             | mvn:org.opendaylight.controller/features-akka/1.6.0-SNAPSHOT/xml/features
org.ops4j.pax.cdi-0.11.0                  | mvn:org.ops4j.pax.cdi/pax-cdi-features/0.11.0/xml/features
odl-config-persister-0.3.0-SNAPSHOT       | mvn:org.opendaylight.controller/features-config-netty/0.3.0-SNAPSHOT/xml/features
odl-config-persister-0.4.0-SNAPSHOT       | mvn:org.opendaylight.controller/features-config-persister/0.4.0-SNAPSHOT/xml/features
odl-config-persister-0.4.0-SNAPSHOT       | mvn:org.opendaylight.controller/features-config-netty/0.4.0-SNAPSHOT/xml/features
odl-controller-1.5.0-SNAPSHOT             | mvn:org.opendaylight.controller/features-akka/1.5.0-SNAPSHOT/xml/features
openflowplugin-extension-0.2.0-SNAPSHOT   | mvn:org.opendaylight.openflowplugin/features-openflowplugin-extension/0.2.0-SNAPSHOT/xml/features
```

Figure 4 - OpenFlow Plugin repositories-list

Fynally the optnode features will be allowed for installation by using the following commands in the Apache Karaf console:

```
feature:install odl-optnode
feature:install odl-optnode-api
feature:install odl-optnode-rest
feature:install odl-optnode-ui
```

REFERENCES

[1] Krishnaswamy, U, “Innovation in SDN Tools and Platforms”, ON.LAB, 4-7 (2013).

[2] ONF White Paper, “Software-Defined Networking: The New Norm for Networks”, Open Networking Foundation, 3-7 (2012).

[3] Goransson, P. and Black, C, “How SDN Works”, Software Defined Networking - A Comprehensive Approach, 59-61 (2014).

[4] Open Networking Foundation
<www.opennetworking.org>

[5] ONF - SDN 3 Layers
<www.opennetworking.org/sdn-resources/sdn-definition>

[6] ONF, “SDN Architecture Overview”, Version 1.0, Open Networking Foundation, (2013).

[7] Goransson & Black, “Software Defined Networks - A Comprehensive Approach”, 1st Edition, Chapter 4, (2014).

[8] ONF, “OpenFlow Switch Specification”, Version 1.4.0, Open Networking Foundation, (2013).

[9] ONF, “OpenFlow Switch Specification”, Version 1.0.0, Open Networking Foundation, (2009).

[10] The Opendaylight Platform
<www.opendaylight.org>

[11] Floodlight OpenFlow Controller - Project Floodlight
<www.projectfloodlight.org/floodlight/>

[12] OpenContrail
<www.opencontrail.org/>

[13] Ryu SDN Framework
<osrg.github.io/ryu/>

[14] ONOS Project
<onosproject.org/>

[15] SAL Architecture Overview
<https://wiki.opendaylight.org/view/OpenDaylight_Controller:_SAL_Architecture_Overview>

[16] Plugin Design
<https://wiki.opendaylight.org/view/OpenDaylight_OpenFlow_Plugin:Overview_Architecture>

[17] OSGi Alliance - The OSGi Architecture
<www.osgi.org/Technology/WhatIsOSGi>

[18] Karaf 4.0.1 - The Apache Software Foundation
<karaf.apache.org/>

[19] Andy, B, “A Guide to NETCONF for SNMP Developers”, IEEE 802 Plenary, v0.6,17-24 (2014).

ACRONYMS

AD-SAL	API-Driven Abstraction Layer
API	Application Programming Interface
BGP-LS	Border Gateway Protocol Link-State
CLI	Command-Line Interface
DLUX	openDayLight User eXperience
DOM	Document Object Model
DTO	Data Transfer Objects
FPGA	Field Programmable Gate Array
GRE	Generic Routing Encapsulation
IDS	Intrusion Detection System
IT	Information Technology
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LISP	LISt Processing
MD-SAL	Model-Driven Abstraction Layer
MPLS	Multiprotocol Label Switching
NFV	Network Functions Virtualization
ODL	OpenDaylight
OF	OpenFlow
ONF	Open Networking Foundation
ONOS	Open Network Operating System
OSGi	Open Services Gateway Initiative
OVSDB	Open vSwitch Database Management Protocol
PCEP	Path Computation Element Protocol
POM	Project Object Model
QoS	Quality of Service
REST	Representational State Transfer
RPC	Remote Procedure Call
SDN	Software Defined Networking
SNMP	Simple Network Management Protocol

TCP	Transmission Control Protocol
VLAN	Virtual LAN
VNF	Virtualized Network Function
VRRP	Virtual Router Redundancy Protocol
XML	eXtensible Markup Language
XMPP	Extensible Messaging and Presence Protocol