OPENDAYLIGHT AS A CONTROLLER FOR SOFTWARE DEFINED NETWORKING

ARNAU SALAS FIGUEROLA
Supervisor: FRANCO CALLEGATI
Spring 2014/2015
To my parents
that have always supported me
Contents

Abstract vii

1 Introduction 1

2 An introduction to NFV and SDN 5
  2.1 Network Functions Virtualization 6
  2.2 Software Defined Networking 8
  2.3 Relationship between NFV and SDN 11

3 Practical implementation of SDN and NFV 13
  3.1 Southbound APIs 13
  3.1.1 OpenFlow 14
  3.2 Northbound APIs 18
  3.3 SDN Controller 18
  3.4 Mininet 19

4 OpenDaylight 21
  4.1 The project 21
  4.2 History 22
    4.2.1 Releases 22
    4.2.2 Industry support 24
  4.3 Technical overview 25

5 Implementation of applications in OpenDaylight 29
  5.1 AD-SAL 30
  5.2 MD-SAL 34
## CONTENTS

6  NFV implementation in ODL: Virtual Tenant Network  
   6.1  Overview .............................................. 41  
   6.1.1  Integration with OpenDaylight ................. 43  
   6.2  Use case ............................................. 44  

7  Conclusions .............................................. 49  

8  Acknowledgements ........................................ 53  

A  Appendix .................................................. 55  
   A.1  Full requests sent to postman in MD-SAL example 55  
   A.1.1  Putting the flows ................................. 55  
   A.1.2  Getting the list of flows ...................... 60
Abstract

This document intends to be an in depth approach to the new possibilities that telecommunication networks are offering through the new paradigms of Network Functions Virtualization and Software Defined Networking, which are following the development to the cloud and the open source principle.

The main advantage that these innovations provide is a cost reduced, dynamical and highly programmable network without sacrificing too much the performance.

However, to take advantage of these improvements, new protocols and software are needed in order to control it and, in the end, have the best possible performance.

For this reason, this document starts deepening into the NFV and SDN concepts, but then focuses on OpenDaylight, an SDN controller, one of the most industry supported ones.

It attempts to do it in three ways: firstly learning how it works internally, then studying how to create applications to make use of it, and finally how to take advantage of the projects created around it, making focus on the Virtual Tennant Networks.
Chapter 1

Introduction

In the modern telecommunication networks, there is an increasing variety of proprietary hardware, and everyday, companies are launching new services and a wider variety of network configurations is needed. In order to satisfy these demands, the use of hardware-based equipment is becoming less of a solution, considering that it requires additional floor space, power and trained staff, and rapidly reaches its end of life. That being so, a network based on boxes dedicated to single functions is definitely not the best approach to achieve optimal dynamic service offerings.

The network design must be able to dynamically adapt to the changing needs of traffic and services running over it, and it is at this point where Network Functions Virtualization (NFV) and Software Defined Networks (SDN) become relevant. Both these concepts follow the paradigm of Cloud Computing.

This paradigm is leading to a transformation not only of networking but also of storage and computing, in such a way that resources are offered by a provider to a client. As a result of this, data is treated in distributed and virtualized resources, both software and hardware, which means that they can be located anywhere around the world and exploited through broadband connection [1].

In terms of costs, the change from specialized to homogeneous and generic hardware means a great reduction. This can lead to spend more on software innovation which is also cheaper. Additionally, the use of open source software, which is very present in these new paradigms, can also mean a significant drop of costs.
CHAPTER 1. INTRODUCTION

But how has the industry arrived to this point? The origin of this evolution comes with the one from the Internet. The Internet is mainly based on packet forwarding according to their IP address. However, every day this is becoming less simple because packets are being processed in additional intermediate nodes in order to add functions such as firewalls or load balancing.

This kind of functions are commonly known as middle boxes. They typically consist of a physical device with proprietary software and a limited set of functions, thus making the user totally dependent on the provider if they require a certain amount of them.

It is clear that a large amount of companies behaving this way leads to a chaotic network, making it difficult for operators to develop new network functions and provoking a standstill in its evolution.

This situation caused the search for a turn that would redirect the network to a more dynamic vendor-free state. As a result, the Network Functions Virtualization was introduced back in 2012 with the main idea of virtually creating the once physical middle boxes functions, and putting them into generic pieces of hardware.

This concept contributes to reinforce a tendency experienced in the last few years that will most likely expanding, which consists in moving the network resources as close as possible to the user, and leaving the core network as an ensemble of links fast enough to allow communication between the edges [2].

Along with these new approaches and in order to make them work and profitable, it has been introduced a new paradigm: Software Defined Networking (SDN). The key idea in which it is based consists in decoupling the control plane from the forwarding plane. This allows networks administrators to flexibly and efficiently manage the services and applications provided, and abstract them from the underlying infrastructure by directly programming the forwarding plane.

To carry out the managing of the software defined networks, a controller capable of performing the diverse needed tasks is required. There are a number of them, such as Pox, Ryu or Floodlight, but one of the most documented one and in which this document will focus is OpenDaylight. The first version of this open source controller was released in 2014 by the Linux Foundation.

Once the controller is chosen, it is also necessary to have a protocol that will allow the communication between the control plane and the forwarding plane, and make the controller able to determine the path that packets will have to follow in the network. The most extended one is OpenFlow, introduced
in 2008 and now part of the Open Network Foundation.

All in all, it is clear that with all these new paradigms the telecommunication networks are changing, evolving into a more dynamic and efficient state. Because of this reason, this document has the goal of clarifying all these new paradigms and putting all them together in order to explore their possibilities, by setting out some situations and using OpenDaylight to deal with them.

So to clarify all this concepts, in the Chapter 2 we will see in more depth both SDN and NFV, as well as their origins. Instead, on Chapter 3 we will study how are they implemented, and the real tools we will be using to do so.

Later, in the Chapter 4, we will focus on our chosen SDN controller, OpenDaylight, and we will deeply see how it works internally in order to, on Chapter 5 put it in practice and see how to develop applications based on it.

Finally in Chapter 6 we will see a practical implementation of NFV, designed as a project inside OpenDaylight, and how to take advantage from it.
Chapter 2

An introduction to NFV and SDN

As already seen in Chapter 1, the evolution of telecommunication networks has led to the simplification of the core networks and to virtualize network functions and services at the edge networks. All of this following the cloud paradigm, based on using vendor-independent software-defined infrastructures and deploying them into generic hardware.

As already seen in Chapter 1, the evolution of telecommunication networks has led to the simplification of the core networks and to virtualize network functions and services at the edge networks. All of this following the cloud paradigm, based on using vendor-independent software-defined infrastructures and deploying them into generic hardware.

Figure 2.1: Networking evolution tendency

In order to carry out such evolution, some classical views of the network
had to be changed, and thus the concepts of NFV and SDN were presented. As we will see later in this chapter, they do not depend on each other, but combined represent the way in which the industry has approached its new challenges.

2.1 Network Functions Virtualization

On October 22-24, 2012 during the “SDN and OpenFlow World Congress”, it was published a white paper introducing Network Functions Virtualization, with the collaboration of some major companies of the industry such as AT&T, Deutsche Telekom, Telecom Italia or Telefonica, which are part of the ETSI (European Telecommunication Standards Institute) [3].

According to this white paper [4], the purpose of this new paradigm was to face the problems that launching new services in a network with a large and increasing variety of proprietary hardware is causing in the last few years. This is, the need of more space and power, the increasing costs of energy and capital investments, the complexity of hardware-based appliances and their decreasing life cycles leading to little or no revenue.

The way in which issues are approached is making use of standard virtualization technology to virtualize a wide range of network node functions, consisting in one or more virtual machines running different software and processes, into industry standard high volume servers, switches and storage.

The main benefits expected to obtain through virtualizing network functions are the following:

- Cost reduction: virtualizing leads to less investment in equipment and a reduced power consumption through consolidating equipment.

- Business speed up: the innovation cycle of network operators is decreased due to focusing innovation on software-based developments.

- Multi-version and multi-tenancy: networks appliances are now capable of using a single platform for different applications, users and tenants, allowing operators to reuse the same resources for a diversity of purposes depending on their needs.

- Scalability: services can be rapidly scaled as required, based on geography or customer situation.
• Openness: the open nature of NFV allows the incorporation to the industry to smaller or software-only based companies, diminishing investment and risks. This brings new opportunities of revenue and innovation.

Of course there are also some downsides. The main one is the possible loss of performance due to the move form specialized to homogeneous hardware. However this is not expected to be a problem in the long term thanks to the high speed of the innovation in software.

At the same time, to achieve these benefits, there are some technical challenges that have to be faced:

• Making the virtualized network appliances portable between different hardware vendors.

• Being able to manage and orchestrate many virtual network appliances at the same time and ensuring their security.
• Achieving the automation of all the functions to ensure scalability.

• Ensuring adaptability in case of software or hardware failure.

• Integrating multiple virtual appliances from different vendors.

In practice, a service provider who follows the NFV paradigm will implement a number of virtual network functions (VNFs), but as a single VNF will not provide by itself a complete product or service for the customer, some of this services will have to be concatenated. As a result, it is introduced the concept of service chaining, in which multiple VNFs are used in sequence to deliver a service.

For operators, this situation of having the services distributed and virtualized means also a great advantage in terms of efficiency managing the resources which, at the same time, leads to an overall improvement to the Quality of Service (QoS) provided to the customer.

At this point it is easy to come to a conclusion: the advantages of NFV are obvious, and its great adoption by the industry proves it, but it seems that a way to control all this virtualized deployment would be very useful, some sort of orchestration process. And here is where SDN enters the scene.

2.2 Software Defined Networking

The first standard documentation relative to SDN was introduced by the University of Berkeley and Stanford in 2008. Later, in 2011, it was born the Open Networking Foundation (ONF), a user-driven organization dedicated to the promotion and adoption of SDN through open standards development [6].
2.2. SOFTWARE DEFINED NETWORKING

According to the ONF [7], Software-Defined Networking is a new approach to networking, in which the control plane (which decides how to handle traffic) is decoupled from the data plane (which forwards the traffic according to the decisions that the control plane makes) and is directly programmable.

This results in an extremely dynamic, easy to use, cost-effective, and adaptable architecture that gives administrators unprecedented programmability, automation and control. To summarize, the following are the main characteristics of SDN:

- Directly programmable: the decoupling from forwarding functions allows the direct programming of the network control.
- Agile: this separation lets administrators dynamically adapt the traffic to the changing needs.
- Centrally managed: the intelligence of the network is provided by SDN controllers that have a general view of the network, and are seen by higher level applications as a single switch.
- Programmatically configured: network managers are able to configure, manage, secure and optimize networks efficiently through automated SDN programs which, in addition, they can write themselves thanks to not depending on proprietary software.
- Open standards-based and vendor-neutral: being implemented through open standards allows the simplification of network operation and design.
To be able to provide these characteristics, the SDN architecture consists of three distinct layers:

- **Application Layer**: it is composed by the applications that communicate directly to the controller and ask for the resources they need. They can have a very wide range of forms and purposes.

- **Control Layer**: it provides the logically centralized control functionality that supervises the network behaviour through open interfaces. This is usually done by a software entity that is capable of translating the needs of the applications into rules for the data plane and also providing them with information about the network.

- **Infrastructure Layer**: it consists of all the low level network elements and devices that provide packet switching and forwarding.

In order to allow the interaction between these layers two interfaces are used:

- **Northbound Interface**: it provides communication between the applications and the controller. Using open APIs the applications can ask for information about the network state and consequently apply the network control measures they find necessary.

- **Southbound Interface**: it communicates the control layer with the network elements from the data path, allowing programmatic control of forwarding rules. It is expected to be open, vendor neutral and easily interoperable. The most extended protocol used in this interface is OpenFlow.
As a result of all this, it seems like, thanks to its architecture, SDN is capable of providing the network the innovation necessary to move from a static network to a dynamic platform able to efficiently adapt to the growing needs of the industry and the users.

Later, in the next chapter, we will see some of the most extended practical implementations of each one of the above elements.

### 2.3 Relationship between NFV and SDN

Having seen in more detail these two paradigms, we can conclude that despite being independent concepts they combine very well and complement each other. But to make it more clear, we can summarize them as follows.

**SDN**: it is concerned in network functionalities. Decouples the control and data planes and provides centralized controller and network programmability.
NFV: it is the concept of transferring the network functions from dedicated hardware devices to software based applications and decouples the network functions from the proprietary hardware without affecting the functionality.

This way we can conclude that:

- SDN serves NFV by providing the programmable connectivity between VNFs. Connections that will be orchestrated by the SDN controller.

- NFV serves SDN by implementing its network functions in a software manner, allowing the SDN controller to be virtualized and run on a cloud, that can be migrated according to the needs of the moment.
Chapter 3

Practical implementation of SDN and NFV

Now that we have seen in more detail what is NFV and how we can use SDN in order to take advantage of it, in this chapter we will go into detail about which practical implementations that are currently being used to exploit them.

We will be taking a look at how northbound and southbound interfaces are implemented, we will also make a general review of SDN controllers, and lastly, although not being actually used in the practice, we will take a look at mininet, a network simulator, provided that it has been used in the practical experimentation that will be developed later in this document.

Before getting into further detail, it is important to remark how most of the protocols and implementations that we will see follow the open paradigm. It has become clear, with the support of the biggest companies of the industry, this is the way in which it will be developing and innovating.

3.1 Southbound APIs

The southbound application program interfaces are used to communicate between the SDN controller and the switches and routers of the network. They facilitate efficient control over the network and enable the controller to dynamically make changes according to real-time demands and needs [9]. The most extended one, and the one used while developing this document, is OpenFlow.
3.1.1 OpenFlow

The first OpenFlow specification was created in 2008 by the Stanford University. They released the version 1.0 by the end of 2009, but with a clear goal of making it open and owned by the community [10]. This is why, since 2011, the Open Network Foundation has been the organisation responsible for its promotion and adoption. Now, it is supported by many switch and router vendors such as Cisco, IBM, Juniper Networks or Hewlett-Packard.

The first version developed already managed by the ONF, the 1.1, was released on 28 February 2011. One year later it was released the version 1.2, and the most recent one is the version 1.4.

OpenFlow allows the actual moving of network control from the switches to the control layer, actually separating the control plane from the data plane. However, as a southbound interface, it needs to be implemented on both sides it, this is, in the SDN controller and the infrastructure devices. These last ones are called OpenFlow Switches.

![OpenFlow logo](image)

Figure 3.1: OpenFlow logo

**OpenFlow Switch**

It is the representation of the actual underlying switch that the SDN controller is going to manage. It basically consists of one or more OpenFlow tables and a group table, which perform packet analysis and forwarding within the switch.

These switches that support OpenFlow can be both virtual or physical. The physical ones are not only those built with OpenFlow in mind, but also those legacy switches that can be updated to support at least the first version of OpenFlow.

Whichever is the case, an OpenFlow Switch will consist of the following components:
3.1. SOUTHBOUND APIS

- **Ports**: packets will enter the switch and exit it through them. They do not have to be necessarily physical as they may be logical ports defined by the switch.

- **OpenFlow tables**: they perform packet analysing and forwarding, and contain a series of OpenFlow entries, which are used to match and process packets according to their packet headers.

- **Channel**: it is the interface used to communicate the switch with the controller, therefore where the switch receives the configuration from it.

![OpenFlow switch structure](image)

Figure 3.2: OpenFlow switch structure [11]

In OpenFlow, all the traffic without distinguishing between packets and circuits is abstracted as **flows**. This is because OpenFlow takes advantage of the fact that typically, modern switches already make use of flow tables to implement different kinds of functions, and although each vendor has its own flow table, there is a common set of functions that works in most of them [12].
Mode of operation: The Pipeline

When a flow arrives to a switch, it goes through all the linked flow tables, which provide matching, forwarding and packet modification which combined are called the pipeline.

In each flow table, the arriving packet is checked against the entries according to its header. The entries are made up of match fields, for matching packets, a priority field, to decide matching order, a counter for statistic purposes and a set of instructions to be applied if the match happens.

There is a wide range of possible matches, typically from Layer 2 to Layer 4. For OpenFlow 1.3, the following is the full list:

![OpenFlow 1.3 Flow Match Fields](image)

Figure 3.3: List of OpenFlow 1.3 possible matches

Regarding to the instructions that can be applied, there are some required ones and some others which are optional [13]:

- **Apply-Actions action/s (Optional):** applies the specific action/s immediately, without any change to the Action Set.

- **Clear-Actions (Optional):** clears all the actions in the action set immediately.
3.1. **SOUTHBOUND APIS**

- **Write-Actions** action/s *(Required)*: merges the specified action/s into the current action set.

- **Write-Metadata** metadata/mask *(Optional)*: writes the masked metadata value into the metadata field.

- **Goto-Table** next-table-id *(Required)*: indicates the next table in the processing pipeline. The table-id must be greater than the current table-id.

The following figure illustrates the overall work-flow of a packet flow through an OpenFlow switch:

![Figure 3.4: OpenFlow work flow [13]](image-url)
3.2 Northbound APIs

In the SDN architecture, the northbound APIs are used to allow communication between the controller and the services and applications running over the network [14]. They enable the efficiency to orchestrate and automation of the network according to the needs of the application.

Since the value of SDN resides in the ability of northbound applications to take advantage of the resources in the most innovative ways, the northbound APIs have to support the most applications. This leads to these APIs to be the most dispersed ones.

One way to implement such interfaces, used in controllers such as OpenDaylight, is through REST APIs. REST (Representational State Transfer) is a software architecture style for building scalable web services. It consists of a coordinated set of constraints applied to the design of components in a distributed hypermedia system that can lead to a better performing and maintainable architecture [15].

REST APIs typically communicate through HTTP, and use the same commands used to retrieve web pages and send data to remote servers (GET, POST, PUT, DELETE, etc.).

Northbound APIs are also used to integrate the SDN controller with automation and orchestration platforms, such as OpenStack. These allows these platforms to fulfill the needs of the applications without having to completely understand how that translates to the network.

3.3 SDN Controller

In the SDN architecture, the controller provides the intelligence of the network and it is where the control plane resides. Using the northbound and southbound APIs, it allows the communication between the application layer and the infrastructure layer.

Typically, the controllers are formed by modules that do a variety of functions. Some of the most common ones are tasks related to commanding and inventorying the network or gathering information and statistics about it.

There are multiple SDN controllers that have been developed, both open-source and proprietary, some of them for particular purposes and others for general purposes. Among the open-source controllers, the following are some of the most relevant ones [16]:

...
3.4. MININET

- **NOX**: based on C++ and Python, it was the first OpenFlow controller.

- **POX**: general SDN controller based on Python, that supports OpenFlow. It has a high-level SDN API including a queriable topology graph and support for virtualization.

- **RYU**: it is an open-source Network Operating System (NOS) that supports OpenFlow. It is based in Python.

- **OpenDaylight**: it is a Java-based open-source SDN controller hosted by The Linux Foundation that has received great support from the industry. We will widely talk about it in the next chapter, as it has been the main topic of investigation of this thesis.

3.4 Mininet

For the purpose of this thesis, it was necessary a tool that allowed an experimentation environment, because testing relatively new tools that are currently being developed and might not be fully reliable could be a cause of problems in real environments. This is why mininet was used in order to simulate the infrastructure layer.

Mininet is a network emulator which creates a network of virtual hosts, switches, controllers, and links. The hosts run standard Linux network software, and its switches support OpenFlow for highly flexible custom routing [17].

Some of its main characteristics are:

- It provides a simple and inexpensive network testbed for developing OpenFlow applications.

- It enables complex topology testing, without the need to wire up a physical network.

- It includes a CLI (Command Line Interface) that is topology-aware and OpenFlow-aware, for debugging or running network-wide tests.

- It supports arbitrary custom topologies, and includes a basic set of parametrized topologies.
• It can be used out of the box without programming, as well as using the provided Python API for network creation and experimentation.

Figure 3.5: The basic mininet instruction [18]

In addition, mininet is completely compatible with our chosen controller, OpenDaylight, and no further actions need to be done in order for them to work together.
4.1 The project

OpenDaylight is a collaborative open-source project hosted by The Linux Foundation. The goal of the project is to accelerate the adoption of SDN and create a solid foundation for NFV [19].
With OpenDaylight, a community has come together to accomplish this goal through the combination of open community developers and open-source code and project governance that guarantees an open, community decision making process on business and technical issues.

It can be a core component within any SDN architecture. The open-source nature of the controller enables users to reduce operational complexity, extend the life of their existing infrastructure hardware and enable new services and capabilities only available with SDN.

For these reasons, such open-source controller framework can be of great utility for different kind of enterprises such as enterprise IT providers, network service providers or cloud services provider.

4.2 History

On February 8, 2013, a new coalition forming around SDN was announced by the SDN Central [19]. One month later, on April 8, The Linux Foundation announced the founding of the OpenDaylight Project as a community-led and industry-supported framework to promote the adoption of the new network paradigms of SDN and NFV.

The original founders of the project were: Arista Networks, Big Switch Networks, Brocade, Cisco, Citrix, Ericsson, HP, IBM, Juniper Networks, Microsoft, NEC, Nuage Networks, PLUMgrid, Red Hat and VMware. They committed to providing economical and engineering resources to help in the development of the platform.

4.2.1 Releases

Since the creation of the project there have been three major releases of the OpenDaylight controller: Hydrogen, Helium and Lithium [20].
4.2. HISTORY

Hydrogen

It was released on February 4, 2014. It was delivered in three different editions, each one oriented to a different kind of user:

- **Base Edition**: oriented to those exploring SDN and OpenFlow for proof-of-concepts or academic initiatives in physical or virtual environments.

- **Virtualization Edition**: designed for data centers and included the basics plus functionality for creating Virtual Tenant Networks and virtual overlays, as well as applications for security and network management.

- **Service Provider Edition**: oriented to providers and carriers managing existing networks that wanted to start using SDN and NFV. Included protocol support as well as security and network management applications.

Due to the release dates and the amount of documentation, some of the experimentation in this thesis was still done using this release.

Helium

Firstly released on September 29, 2014. However, small revisions of it were being released until March 2015. It abandoned the idea of different editions and was released as a unique version.

This was possible thanks to the introduction of karaf as the tool to manage the controller. It allows the dynamic managing of the available modules, so that, starting from the base controller, the user can install the features that satisfy its particular needs.

This version has been the main subject of study during the development of this document.

Lithium

It was released on June 29, 2015. It followed the idea introduced by Helium. It introduced some new features in many of the functionalities included in the controller, making particular focus on broadening the programmability of intelligent networks.
4.2.2 Industry support

As we have seen, from the beginning, OpenDaylight has received wide support from the industry. This has been an important reason of its success. At this moment, there are almost 50 companies supporting the project, divided into Platinum, Gold, and Silver members, depending on the amount of resources they provide.

![OpenDaylight Supporting Members](image)

Figure 4.3: OpenDaylight supporting members [21]
4.3 Technical overview

OpenDaylight is an open-source project with a modular and flexible controller platform on its core. This controller is implemented completely in software, and it is contained in its own Java Virtual Machine. For this reason, it can be deployed on any hardware or operating system that supports Java [22].

As we can see in the image above, OpenDaylight has the structure of a SDN environment, as we have seen them in chapter 3. The controller exposes open northbound APIs which are used by applications. It supports the OSGi framework and bidirectional REST for the northbound API.

OSGi is a modular system and service platform for the Java programming language that implements a completely dynamic component model, something that does not exist in standalone JVM environments.

While REST is used by applications running outside the controller itself, and even in different machines, the inner applications use OSGi. All in all,
the applications are the ones providing the logic, and using the controller to gather network intelligence or run algorithms, and then to orchestrate the new rules throughout the network.

In OpenDaylight, there are some dynamically pluggable modules, responsible for performing network tasks, which are contained in the controller itself. But it is also possible to insert other services and extensions for enhanced SDN functionality.

On the other hand, the southbound layer supports multiple protocols, for instance OpenFlow in its versions 1.0 and 1.3, and BGP-LS. These modules are dynamically linked to a Service Abstraction Layer (SAL). The SAL exposes the infrastructure layer to the applications north of it, and determines how to fulfill the requested services independently of the underlying protocol used and the network devices.

There are two different approaches to the SAL that can be taken into account when programming applications for OpenDaylight: the API-Driven SAL (AD-SAL) and the Model-Driven SAL (MD-SAL).

**AD-SAL**

The AD-SAL approach has the following main characteristics:

- It can be used with both southbound and northbound pluggins.

- It is stateless.

- It is limited to flow-capable devices and services only.

- The applications are programmed into the controller as OSGi bundles.

- The flow programming is reactive, by receiving events from the network.
4.3. TECHNICAL OVERVIEW

MD-SAL

This approach has the following features:

- It has a common REST API for all the modules.
- It can store data for models in permanent or volatile APIs.
- It is model agnostic. It supports any device or service models.
- The applications are programmed outside the controller.
- The flow programming is proactive, without the possibility to receive events from the network.
CHAPTER 4. OPENDAYLIGHT

Figure 4.6: MD-SAL mode of operation

Model-Driven SAL (MD-SAL)
Chapter 5

Implementation of applications in OpenDaylight

We have already seen the framework that OpenDaylight offers in order to develop applications. Now, in this chapter we are going to see some actual implementations of applications to be able to see how the controller really works [24].

Before getting into further detail, it is important to mention that, although the edition of the code has been made through the Eclipse IDE, in OpenDaylight the projects are built via Maven.

Maven is a build automation tool hosted by the Apache Software Foundation used primarily for Java projects. Maven projects are configured using a Project Object Model, which is stored in a pom.xml file.

When used for a Java project, Maven auto-generates the following directory structure:
5.1 AD-SAL

The developed application for this case is a learning switch. This consists in the dynamically installation of flow rules into the switches in order to accelerate future traffic.

As the switches we will use are OpenFlow-enabled, the procedure will be as we have seen in chapter 3: when a flow arrives to the switch, it will check through the flow-entries in the tables for rules matching its header. If none is found, the packet will be sent to the controller. It is at this point where our application goes into action.

The application has the following structure:
To build the application with maven, there are two important files:

- **pom.xml**: it is used to indicate the inter-bundle dependencies. It specifies the details of the project and its parent, and provides a list of the used pluggins and bundle dependencies.

- **Activator.java**: it handles the configuration of the dependencies during runtime. It has methods to set the list of java interfaces implemented by the bundle and registers other bundles used.

Once the packet arrives to the controller, a *packet-in* notification is sent. Because of this, all the methods implementing the corresponding listener interface (in this case `IListenDataPacket`) will be able to react to this event by implementing the method `recvieveDataPacket`.

For this reason, it is in this method where the intelligence of our application is. The logic applied in this method is:

- When the packet arrives, decode the information in its header and save it, as well as the port where it comes from.

- If the packet is not an Ethernet packet, it is ignored.
CHAPTER 5. IMPLEMENTATION OF APPLICATIONS IN OPENDAYLIGHT

- Then we associate the source MAC address of the arriving packet to the port it comes from, and save it in a table.

- After that, we check in the same table if there is an associated port for the destination MAC address from the packet.

- If there is not, the packet is flooded to all the ports.

- Otherwise, the flow entry is created with the match of the destination address, and the action to set the output to the port found in the table.

This way, when a similar packet arrives with the same destination, it will find the flow entry that matches it and will be directly forwarded without having to go through the controller again.

Once we have the program written, we will see which are the steps to run it on the controller and see its behavior.

Firstly, we run the controller, that will start all the default bundles. Once that is done, we install our project from its root, with the maven command 

\$mvn clean install

If everything went fine we will see the result:

```
INFO BUILD SUCCESS
INFO Total time: 8.479s
INFO Finished at: Thu Jun 18 02:13:24 PDT 2015
INFO Final Memory: 240/331M
```

Figure 5.3: Build success result

After that, the our project is installed as a bundle in our controller, and we are ready to test its behaviour.

To do so, we will run mininet. For this example we will run a topology with two switches and two hosts:
5.1. AD-SAL

The command to create it is: `>sudo mn --mac --topo linear,2 --switch ovsk --controller remote`. In our case, mininet is running on the same machine as the controller. Otherwise, we would need to specify the IP address in which it were running.

When the topology is running and correctly connected to the controller, we can run a ping test between the two hosts.

As we can see, the first two pings take significantly longer than the rest of them. This is because during these, the packets were being sent to the
CHAPTER 5. IMPLEMENTATION OF APPLICATIONS IN OPENDAYLIGHT

controller in order to create the flow entries. After that, when all the possible entries are installed, flows are directly forwarded and thus the pings take less than one millisecond.

Now we can check the flows installed in the switches with the following command: $sudo ovs-ofcl dump-flows s1 and $sudo ovs-ofcl dump-flows s2.

![Flow dump result](image)

Figure 5.6: Flow dump result

We can see the two flows installed by our application in each switch and, in addition, a default flow that was already installed, that made the packets without match go to the controller.

5.2 MD-SAL

As we said in the previous chapter, the flow installation through MD-SAL is done proactively instead of reactively. For these reason, in this example we will see how to interact with the topology in this case, making use of the REST API provided by the controller.

In this case, we will be using the karaf distribution of the controller.

![Karaf used to run ODL](image)

Figure 5.7: Karaf used to run ODL
5.2. MD-SAL

That being so, we need to check if the required features, that are those related to the `restconf`.

```
openaylight-user@root$ feature:list -i | grep restconf
odl-restconf-all | 1.1-Helium | x | odl-controller
odl-restconf   | OpenDaylight :: Restconf :: All | 1.1-Helium | x | odl-controller
-1.1-Helium     | OpenDaylight :: Restconf | 1.1-Helium | x | odl-controller
-1.1-Helium     | OpenDaylight :: restconf | 1.1-Helium | x | odl-controller
```

Figure 5.8: Restconf features installed

It is also remarkable that since the Helium distribution, there is available a graphic user interface in the form of a web, that can be accessed after installing the feature `odl-mdsal-apidocs`, and entering the address


supposing that we are accessing it from the same machine in which the controller is running.

Figure 5.9: REST API explorer seen in a web browser

Each element of the list is a different set of instructions of the REST API which can be explored and used from the same explorer.
However, we will be using Postman. Postman is REST client provided as an extension of the Google Chrome web browser. It offers a very intuitive GUI and has some interesting features such as the possibility to save groups of commands into collections.

In order to interact with any REST API, there are three elements that we may have to provide:

- **Headers**: they can specify options such as the language in which we have written the body or we want to receive the response. We will also use it to provide the required authentication.

- **Address**: it will specify which is the command we are trying to send. It goes along with the type of instruction we want to send (PUT, GET,
POST, DELETE, etc.). To know which is the address we just need to look up the documentation on the wiki.

- **Body**: for certain kind of instructions such as PUT, we need to provide the data.

In addition, after every REST request, we will receive an HTTP response. A 200 response will mean success, while 400 and 500 responses will mean different problems that will be specified.

In this case we are going to be using the same topology used in the AD-SAL case. So we are going to run mininet the same way we did, and it will automatically connect to the controller.

Now in Postman, we are going to do a PUT call using the following headers:

<table>
<thead>
<tr>
<th>Content-Type</th>
<th>application/xml</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authorization</td>
<td>Basic YWRtaW46YWRtaW4=</td>
</tr>
<tr>
<td>Accept</td>
<td>application/json</td>
</tr>
</tbody>
</table>

![Figure 5.12: Postman headers](image)

Where the authorisation code has been obtained when introducing the default credentials on the corresponding section of Postman.

The URLs we are going to use are the following form:

http://192.168.56.142:8181/restconf/config/opendaylight-inventory:nodes/node/openflow:1/table/0/flow/1

Being 192.168.56.142 our controller’s IP address, and just changing the number of the node and/or the flow for each specific case.

Finally, the body of the request will be:
Again, changing the values for each particular case.

Apart from the response code received, there is also the possibility to check the flows installed in a switch through a GET request. In this case, the headers remain the same, and the address is:

http://192.168.56.142:8181/restconf/config/opendaylight-inventory:nodes/node/openflow:1/table/0

Being it a GET request, now we do not need to provide a body but instead, we will receive a response with all the data.
5.2. MD-SAL

Figure 5.14: Response to the GET request

We can now also do a ping test from mininet, to check the proper function of the flows.

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.160 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.050 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.060 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.052 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.054 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.054 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.053 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.054 ms
```

Figure 5.15: Ping test result

This time all the pings take less than one millisecond because the flows are already installed in the switches.
Chapter 6

NFV implementation in ODL: Virtual Tenant Network

So far, we have seen how to create applications that benefit from the capabilities of OpenDaylight. In this chapter instead, we will see how to take advantage of one of the already existing projects in the controller. Particularly, we will focus on Virtual Tenant Network (VTN), a project that provides tools to implement Network Virtualization.

6.1 Overview

OpenDaylight Virtual Tenant Network is an application that provides multi-tenant virtual network on an SDN controller.

One of the problems that companies have nowadays when deploying their networks is that, these, are usually composed by a certain number of individual isolated networks, one for each department or system. This fact leads to some disadvantages: each tenant needs its own infrastructure and the appliances cannot be shared among them, there is a higher complexity when designing, implementing and operating the network, and a much bigger investment is needed to deploy and operate the network.

The key point of VTN the ability to isolate the physical plane from the logical plane. This is achieved by creating a new logical level of abstraction. By doing so, it gives the chance to create any network needed without having to be aware of the physical network lying underneath or the available bandwidth.

VTN makes it possible to create L2 or L3 networks that, to the eyes of
the user, look like conventional networks. Firstly, the user designs the desired network or networks using the VTN tools, then, these networks are mapped to the physical one by properly configuring the switches using the SDN controller. The logical plane in which the virtual network has been created allows to hide the complexity beneath it and makes it easier to manage the network resources reducing the configuration complexity and the possible errors[25].

![Figure 6.1: Utility of VTN](image)

The overall process to achieve this is:

- The user defines a virtual network.
- Then it maps it to a physical network through pairing their interfaces.
- To make that possible, the VTN intelligence will send the proper instructions to the underlying topology.

In order to create the virtual network, VTN provides different elements that can be created: virtual nodes (bridges, routers...), virtual interfaces and virtual links. The way to interact with it is through its REST API, that we will make use of again with Postman. The set of instructions can be found in the wiki of the project [26].
6.1. OVERVIEW

6.1.1 Integration with OpenDaylight

To make everything possible, the VTN project is divided into different modules:

![Diagram of VTN modules in ODL](image)

Figure 6.2: VTN modules in ODL

**VTN Coordinator**

In the OpenDaylight architecture, the VTN Coordinator is part of the network application, orchestration and services layer. Its main function is to create the virtual network through the OpenDaylight REST APIs. It also provides its own REST APIs to external northbound applications while supporting the use of multiple controllers, and their coordination, at the same time[28].

**VTN Manager**

The manager module provides the intelligence of the project. It is a plugin inside the controller that, interacting with other plugins, is able to implement the components of the created VTN model.
6.2 Use case

In this experiment, we will use the capabilities of VTN to create two virtual networks in order to isolate two pairs of hosts from a topology that we will create with mininet.

Once the controller is running with the necessary features installed, we run the VTN Coordinator that has been installed externally. Then we start mininet with the command:

```
> sudo mn --mac --topo tree,2 --switch ovsk,protocols=OpenFlow13 --controller remote,ip=192.168.56.142
```

to obtain the following topology:

![Topology used for the experiment](image-url)
6.2. USE CASE

Now in Postman, we follow these steps to achieve our goal:

- The first thing to do is to specify which controller we are connecting to, by setting its IP address. After that, all the requests will be sent twice, one for each virtual network.

![Connection to the controller](http://192.168.56.142:8083/vtn-webapi/controllers.json)

![Creation of a virtual network](http://192.168.56.142:8083/vtn-webapi/vtns.json)

![Creation of a virtual bridge](http://192.168.56.142:8083/vtn-webapi/vtn/vbrides.json)

- Now we need to add two interfaces to the virtual bridge, one for each host.
At this point comes the mapping, achieved by associating each created interface to the mininet switch interface to which the desired host is connected.

Once everything is set up, we can perform a pingall test in mininet to check that it is working properly. The result is the following:

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> X h3 X
h2 -> X X h4
h3 -> h1 X X
h4 -> X h2 X
*** Results: 66% dropped (4/12 received)
```

As we can see, the pings only succeed with the pairs of hosts h1-h3 and h2-h4.

We can also send a command to see how the necessary flows have been automatically installed into the switches:
6.2. USE CASE

```
$ mininet> sh ovs-ofctl dump-flows ovn-pod s2

OFPST_FLOW reply (OK=10) {oid=623}:
  cookie=0x0, duration=00:00:00.63, table=0, n_packets=1, n_bytes=102, priority=10, in_port=3, vlan_tci=0x0000/0x1ff, dl_src=00:00:00:00:00:00, dl_dst=00:00:00:00:00:02 actions=output2
  cookie=0x0, duration=00:00:00.63, table=0, n_packets=1, n_bytes=102, priority=10, in_port=3, vlan_tci=0x0000/0x1ff, dl_src=00:00:00:00:00:00, dl_dst=00:00:00:00:00:01 actions=output1
```

Figure 6.11: Flow dump of switch 2
Chapter 7

Conclusions

In this document we have seen the possibilities that OpenDaylight offers as an SDN controller. Particularly, we have seen the two different ways of programming applications, with their advantages and disadvantages.

Even though all the experiments have been carried out in simulated networks, we have been able to test see that the controller is not difficult to manage, because it makes use of already existing tools such as Maven or OSGI in order to work.

We have seen how different are the two approaches to program applications, AD-SAL and MD-SAL. This helps to choose between them depending on whether we want a reactive approach or a proactive one.

On the other hand, we have seen VTN. It is one project being developed inside OpenDaylight, and by itself, it is a paradigmatic implementation of both NFV and SDN. It is able to both create virtual networks and give absolute control of it, allowing at the same time the managing of a real network.

Because of its capabilities and intelligence, VTN could be appropriately used by operators to implement service function chaining, another issue that can be improved thanks to the introduction of the new network paradigms.

OpenDaylight is supposed to be one of the most documented controllers, however, one of the obstacles found while developing this thesis has been precisely this. Many times, while investigating, documentation was found incomplete or outdated. This could be understandable due to the fast evolution of these topics, but should be improved in the future if it wants to be used massively.

All in all, with a massive industry support and a constant updating allow-
ing compatibility with more protocols and standards, OpenDaylight is very likely going to become an essential part of the developing of telecommunication networks.
Bibliography


[21] https://www.opendaylight.org/membership

[22] http://www.opendaylight.org/project/technical-overview

[23] https://www.opendaylight.org


Chapter 8

Acknowledgements

I want to start giving thanks to Franco Callegati for introducing me to the subject of this thesis and proposing me this project, as well as Davide Careglio who made possible our first contact.

I would also like to thank Walter Cerroni, Chiara Contoli, Francesco Foresta and Giuliano Santandrea, for all the help they gave me during the development of this project.

Finally, I want to thank my family for the support they have always given me, that have made this experience possible.
Appendix A

Appendix

A.1 Full requests sent to postman in MD-SAL example

A.1.1 Putting the flows

Switch 1, Flow 1

URL:

http://192.168.56.142:8181/restconf/config/opendaylight-inventory:nodes/node/openflow:1/table/0/flow/1

Body:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<flow xmlns="urn:opendaylight:flow:inventory">
    <strict>false</strict>
    <flow-name>flow1</flow-name>
    <id>1</id>
    <cookie_mask>255</cookie_mask>
    <cookie>102</cookie>
    <table_id>0</table_id>
    <priority>20</priority>
    <hard-timeout>1200</hard-timeout>
</flow>
```
<idle-timeout>3400</idle-timeout>
<installHw>false</installHw>

<instructions>
  <instruction>
    <order>0</order>
    <apply-actions>
      <action>
        <order>0</order>
        <output-action>
          <output-node-connector>1</output-node-connector>
          <max-length>60</max-length>
        </output-action>
      </action>
    </apply-actions>
  </instruction>
</instructions>

<match>
  <ethernet-match>
    <ethernet-destination>
      <address>00:00:00:00:00:01</address>
    </ethernet-destination>
  </ethernet-match>
</match>
</flow>

Switch 1, Flow 2

URL:
http://192.168.56.142:8181/restconf/config/opendaylight-inventory:nodes/node/openflow:1/table/0/flow/2

Body:

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<flow xmlns="urn:opendaylight:flow:inventory">
  <strict>false</strict>
A.1. FULL REQUESTS SENT TO POSTMAN IN MD-SAL EXAMPLE 57

Switch 2, Flow 1

URL:

http://192.168.56.142:8181/restconf/config/opendaylight-inventor
y:node/openflow:2/table/0/flow/1

Body:

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<flow xmlns="urn:opendaylight:flow:inventory">
  <strict>false</strict>
  <flow-name>flow1</flow-name>
  <id>1</id>
  <cookie_mask>255</cookie_mask>
  <cookie>102</cookie>
  <table_id>0</table_id>
  <priority>20</priority>
  <hard-timeout>1200</hard-timeout>
  <idle-timeout>3400</idle-timeout>
  <installHw>false</installHw>
  <instructions>
    <instruction>
      <order>0</order>
      <apply-actions>
        <action>
          <order>0</order>
          <output-action>
            <output-node-connector>1</output-node-connector>
            <max-length>60</max-length>
          </output-action>
        </action>
      </apply-actions>
    </instruction>
  </instructions>
  <match>
    <ethernet-match>
      <ethernet-destination>
        <address>00:00:00:00:00:02</address>
      </ethernet-destination>
    </ethernet-match>
  </match>
</flow>
A.1. **FULL REQUESTS SENT TO POSTMAN IN MD-SAL EXAMPLE 59**

</flow>

**Switch 2, Flow 2**

**URL:**

http://192.168.56.142:8181/restconf/config/opendaylight-inventory:nodes/node/openflow:2/table/0/flow/2

**Body:**

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<flow xmlns="urn:opendaylight:flow:inventory">
  <strict>false</strict>
  <flow-name>flow2</flow-name>
  <id>2</id>
  <cookie_mask>255</cookie_mask>
  <cookie>102</cookie>
  <table_id>0</table_id>
  <priority>20</priority>
  <hard-timeout>1200</hard-timeout>
  <idle-timeout>3400</idle-timeout>
  <installHw>false</installHw>
  <instructions>
    <instruction>
      <order>0</order>
      <apply-actions>
        <action>
          <order>0</order>
          <output-action>
            <output-node-connector>2</output-node-connector>
            <max-length>60</max-length>
          </output-action>
        </action>
        </apply-actions>
      </instruction>
    </instructions>
```
A.1.2 Getting the list of flows

Switch 1

URL:

http://192.168.56.142:8181/restconf/config/opendaylight-inventory
y:nodes/node/openflow:1/table/0/

Response:

<table
   xmlns="urn:opendaylight:flow:inventory">
   <id>0</id>
   <flow>
      <id>1</id>
      <instructions>
         <instruction>
            <order>0</order>
            <apply-actions>
               <action>
                  <order>0</order>
                  <output-action>
                     <max-length>60</max-length>
                     <output-node-connector>1</output-node-connector>
                  </output-action>
               </action>
            </apply-actions>
         </instruction>
      </instructions>
   </flow>
</table>
A.1. FULL REQUESTS SENT TO POSTMAN IN MD-SAL EXAMPLE

```xml
</instructions>
<hard-timeout>1200</hard-timeout>
<flow-name>flow1</flow-name>
<match>
  <ethernet-match>
    <ethernet-destination>
      <address>00:00:00:00:00:01</address>
    </ethernet-destination>
  </ethernet-match>
</match>
<idle-timeout>3400</idle-timeout>
(strict>false</strict>
<table_id>0</table_id>
<installHw>false</installHw>
<priority>20</priority>
<cookie_mask>255</cookie_mask>
<cookie>102</cookie>
</flow>
<flow>
  <id>2</id>
  <instructions>
    <instruction>
      <order>0</order>
      <apply-actions>
        <action>
          <order>0</order>
          <output-action>
            <max-length>60</max-length>
            <output-node-connector>2</output-node-connector>
          </output-action>
        </action>
      </apply-actions>
    </instruction>
  </instructions>
<hard-timeout>1200</hard-timeout>
<flow-name>flow2</flow-name>
<match>
```
Switch 2

URL:
http://192.168.56.142:8181/restconf/config/opendaylight-inventory:nodes/node/openflow:2/table/0/

Response:
<table
xmlns="urn:opendaylight:flow:inventory">
    <id>0</id>
    <flow>
        <id>1</id>
        <instructions>
            <instruction>
                <order>0</order>
                <apply-actions>
                    <action>
                        <order>0</order>
                        <output-action>
                            <max-length>60</max-length>
                            <output-node-connector>1</output-node-connector>
                        </output-action>
                    </action>
                </apply-actions>
            </instruction>
        </instructions>
    </flow>
</table>
A.1. FULL REQUESTS SENT TO POSTMAN IN MD-SAL EXAMPLE

```xml
<output-action>
</action>
</apply-actions>
</instruction>
</instructions>
<hard-timeout>1200</hard-timeout>
<flow-name>flow1</flow-name>
<match>
  <ethernet-match>
    <ethernet-destination>
      <address>00:00:00:00:00:02</address>
    </ethernet-destination>
  </ethernet-match>
</match>
<idle-timeout>3400</idle-timeout>
<strict>false</strict>
<table_id>0</table_id>
<installHw>false</installHw>
<priority>20</priority>
<cookie_mask>255</cookie_mask>
<cookie>102</cookie>
</flow>
<flow>
  <id>2</id>
  <instructions>
    <instruction>
      <order>0</order>
      <apply-actions>
        <action>
          <order>0</order>
          <output-action>
            <max-length>60</max-length>
            <output-node-connector>2</output-node-connector>
          </output-action>
        </action>
      </apply-actions>
    </instruction>
  </instructions>
</flow>
```
<table>
<tr><td></td></tr>
</table>

</instructions>
<hard-timeout>1200</hard-timeout>
<flow-name>flow2</flow-name>
<match>
  <ethernet-match>
    <ethernet-destination>
      <address>00:00:00:00:00:01</address>
    </ethernet-destination>
  </ethernet-match>
</match>
<idle-timeout>3400</idle-timeout>
<strict>false</strict>
<table_id>0</table_id>
<installHw>false</installHw>
<priority>20</priority>
<cookie_mask>255</cookie_mask>
<cookie>102</cookie>
</flow>