Jorge Filipe Henriques Correia
Licenciado em Ciências da Engenharia Eletrotécnica e de Computadores

Application development for Software-Defined networks in state of the art controllers

Dissertação para obtenção do Grau de Mestre em
Engenharia Eletrotécnica e de Computadores

Orientador: Pedro Amaral, Prof. Dr, FCT-UNL

Júri:

Presidente: Prof. Doutor Tiago Cardoso
Arguente: Prof. Doutor Paulo Pinto
Vogal: Prof. Doutor Pedro Amaral

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Março, 2016

Application development for Software-Defined networks in state of the art controllers

Aos meus avós

Aos meus pais e irmão

À minha namorada

Agradecimentos

Em primeiro lugar gostaria de agradecer esta dissertação ao meu orientador, o professor doutor Pedro Amaral, por sempre me ter apoiado durante a sua realização, mesmo depois de algum desleixo da minha parte. Demonstrou-se sempre disponível para ajudar, respondendo a todas as minhas questões, por vezes fora de horas, para me permitir terminar esta etapa da minha vida.

Um grande obrigado à Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa por ter assegurado todos os meios e condições ao longo do meu percurso académico.

Agradeço também à minha família que sempre me apoiou em todas as minhas opções e que fez com que nunca me faltasse nada na vida, fazendo-me feliz, acima de tudo. Aos meus avós, pela ajuda necessária para completar este curso, sendo na calma e confiança transmitida pela minha avó, ou uma palavra sábia transmitida no momento certo pelo meu avô. À minha tia pelas toneladas de conselhos e relatos de experiências vividas que me serviram de lição para os estudos e para a vida. Ao meu irmão, pelas conversas infindáveis até altas horas da madrugada, com tantos desabafos sérios como gargalhadas. E ainda um agradecimento especial aos meus Pais, por todo o seu esforço e dedicação, por quererem sempre o melhor para mim, por estarem sempre presentes quando mais preciso. É graças a eles que consigo concluir mais uma etapa na minha vida e é a eles que dedico todo o meu trabalho. Nunca uma página será grande o suficiente para transmitir o quão agradecido estou, mas fica um profundo Obrigado por tudo o que têm feito por mim.

Finalmente, mas não menos importante, à minha namorada. Minha fiel companheira que nos últimos mais de quatro anos me surpreendeu todos os dias dando-me a conhecer uma das melhores pessoas que já conheci. O seu apoio incondicional e o seu altruísmo, que tiveram o ponto alto na ajuda prestada na elaboração desta dissertação, nunca serão devidamente pagos, mas todos os dias o tentarei fazer de qualquer maneira que me seja possível.

A todos, o meu tremendo obrigado.

Abstract

In the last few years, the importance of the internet in our lives increased considerably.

Networks have become a big part of our lives and there will be a setup almost everywhere we go: in our homes, in the workplace, in stores, in universities, in the subway. Each and every one of these places has a network, a router, Wi-Fi, etc. Due to its high importance, service providers must guarantee a fully operational network, 24 hours a day, leaving no room for mistakes.

However, that creates a problem: how can developers test new protocols? In no way is a service provider willing to risk ruining its network because a developer tested a non-working protocol.

Researchers who dedicate themselves to the study of these frameworks believe that the main problems of a fully operational network lie essentially in its architecture, as network devices incorporate different and quite complex functions. Major networks, such as service providers, are built upon robust architectures with the ability to support large traffic volumes, with different characteristics. The service provider is able to process large amounts of data simultaneously, as well as route and forward traffic. As they have built-in control

functions that work in a distributed manner and considering they are made by a limited number of manufacturers, these networks present several limitations. Besides its complexity and configuration, it must be taken into account that every network should be prepared to deal with potential failures that might occur, as well as any security-related problems. A network - regardless of its level of use - must allow its users to use it as safely as possible.

Networks today have poor flexibility and their development, growth and innovation are far from simple. Thus, the provision of more diversified services to satisfy the users presents a challenge to service providers, since the system and the administration functions are separated.

The answer to these problems lies within the Software-Defined Networks (SDN), given that they seem to be very promising as far as innovation is concerned, allowing the development of new strategies and management control networks.

These networks use programmable switches and routers that can process packets of data for several isolated experimental networks simultaneously, through virtualization. These networks run in the Control Plane, in servers operating separately from the network devices. This gives the network administrator a greater control over the network, as it allows to manage different resources by directing them to different traffic flows.

A SDN using OpenFlow is capable of supporting a high-response network to each and every controller failures that might occur, without slowing the network's response, as it offers great flexibility and helps with fighting the limitations of any existing network.

The main goal of this thesis is to explain how to use this new approach (SDN) and its capacities. This work will serve as a basis to all who wish to obtain new knowledge about this topic. One of the main focuses of this thesis is to

pinpoint the advantages and disadvantages of SDN with an OpenFlow architecture.

Keywords: Software-Defined Networks, OpenFlow

Resumo

Nos últimos anos, a importância da internet nas nossas vidas aumentou consideravelmente.

As redes tornaram-se uma grande parte das nossas vidas, e nos sítios que frequentamos no nosso quotidiano: nas nossas casas, no trabalho, nas lojas, nas universidades, no metro. Aqui, é quase sempre possível encontrar uma rede, um router, uma rede Wi-Fi, etc. Devido à sua elevada importância, os prestadores de serviços devem garantir uma rede totalmente operacional, 24 horas por dia, sem haver margem para erros.

No entanto, isso cria um problema: Como é que se podem testar novos protocolos? Os prestadores de serviços não estão dispostos a arriscar arruinar a sua rede porque um programador testou um protocolo que pode comprometer a rede.

Os investigadores que se dedicam ao estudo destas redes acreditam que os principais problemas de uma rede totalmente operacional estão na sua arquitetura, pois os dispositivos de rede disponibilizam funcionalidades diferentes e bastante complexas. Grandes redes, como as dos prestadores de serviços, são construídas sobre arquiteturas robustas e são capazes de suportar

grandes quantidades tráfego e com diferentes caraterísticas. O prestador de serviço necessita de processar uma grande quantidade de dados simultaneamente, enquanto encaminha tráfego. Como estas redes contêm funções internas de controlo que funcionam de forma distribuída e considerando que são desenvolvidas por um número limitado de fabricantes, estas redes apresentam várias limitações. Além da sua complexidade e configuração, estas redes também têm de estar preparadas para lidar com potenciais falhas que possam ocorrer, assim como com problemas de segurança. Uma rede – independentemente do nível de uso- deve ser o mais segura possível.

As redes atuais têm pouca flexibilidade e o seu desenvolvimento, crescimento e inovação são tarefas complexas. Portanto, a prestação de serviços mais diversificada para satisfazer os utilizadores representa um desafio para os prestadores de serviços.

A resposta para estes problemas está no paradigma das redes definidas por software (SDN), dado que as SDNs são muito promissoras, quanto à inovação e à possibilidade de automação e mais independência do hardware.

Estas redes utilizam switches programáveis que podem processar pacotes de dados de acordo com as regras instaladas por um plano de controlo separado, gerido em software. Isto dá ao administrador de rede um maior controlo sobre a rede, já que permite gerir os recursos, direcionando-os para diferentes fluxos de tráfego.

Uma rede definida por software, que use o protocolo OpenFlow, oferece uma resposta rápida para cada falha que possa ocorrer no controlador, sem atrasar o funcionamento da rede, Oferece uma grande flexibilidade e ajuda na luta contra as limitações das redes tradicionais.

O objetivo principal desta tese é explicar como usar esta nova abordagem (SDN) e as suas capacidades. Este trabalho servirá como base para quem deseja

saber mais sobre este tópico. Um dos focos é mostrar as vantagens e desvantagens das SDN com o protocolo OpenFlow.

Palavras- Chave: Software-Defined Network, OpenFlow

Contents

Figures

Tables

1

1. Introduction

1.1. Motivation

Today's computer networks and infrastructures constitute an important service to society, by serving as infrastructure for several other services. However, the inflexibility of the traditional networks' architecture is becoming a problem.

Network operations can be divided into two levels designated by data plane and control plane [1], as shown in Figure 1.1. In traditional networks these two levels run within each switch.

The data plane is responsible for forwarding packets from one device to another. The control plane consists in the protocols used to produce the decisions on how to forward the packets. These decisions are usually enforced by the forwarding tables of the switches, routers and all data plane devices.

There are also software services which are used to remotely monitor and configure the control functionality.

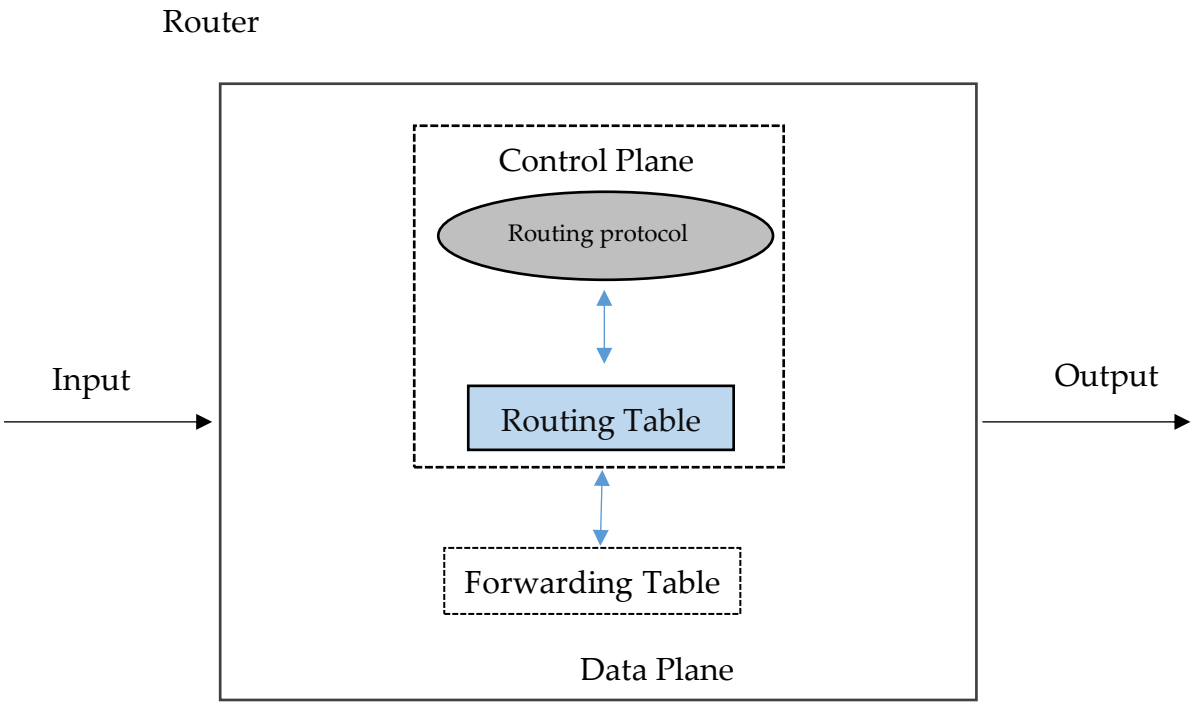

Figure 1.1: View of traditional networking

Traditional networks are composed by several kinds of equipment, from routers and switches to middleboxes, such as network address translators, firewalls, server load balancers, and intrusion-detection systems [1].

In traditional networks the architecture itself and its configuration are responsible for a large percentage of errors. The device's software differs from manufacturer to manufacturer, and sometimes even from product to product, within the same manufacturer. In addition, network administrators need to configure every device individually and are prone to committing various errors. In fact, configuration errors still account for a large percentage of data center failures and are the number one security threat. However paramount, reconfiguration and response mechanisms are still virtually non-existent in current IP networks [1].

Moreover, with the addition of thousands of network devices that must be individually configured and managed, networks became vastly more complex.

When a network device is added, the control plane on every existing network's elements needs to be updated, which leads to an ossification of the network. The deployment of new services requires individual configuration of every network equipment, which is very time consuming and sometimes entails the installation of new devices. This is blocking innovation and has been hindering networks' management. So, the network designers have to implement sophisticated policies and tasks, by using a limited and restrictive set of low-level device configuration commands, in order to meet systems requirements and guarantee their good performance.

Software-Defined Networks offer a new paradigm that tries to overcome the mentioned obstacles. SDNs have four basic design characteristics: [1]

1. The control and data plane are decoupled;
2. Forwarding decisions are flow-based;
3. Control logic is moved to an external entity, the SDN controller or Network Operating System (NOS);
4. The network is programmable through software applications running on top of the NOS that interacts with the underlying data plane devices. [1].

The following image presents the structure of a Software-defined Network:

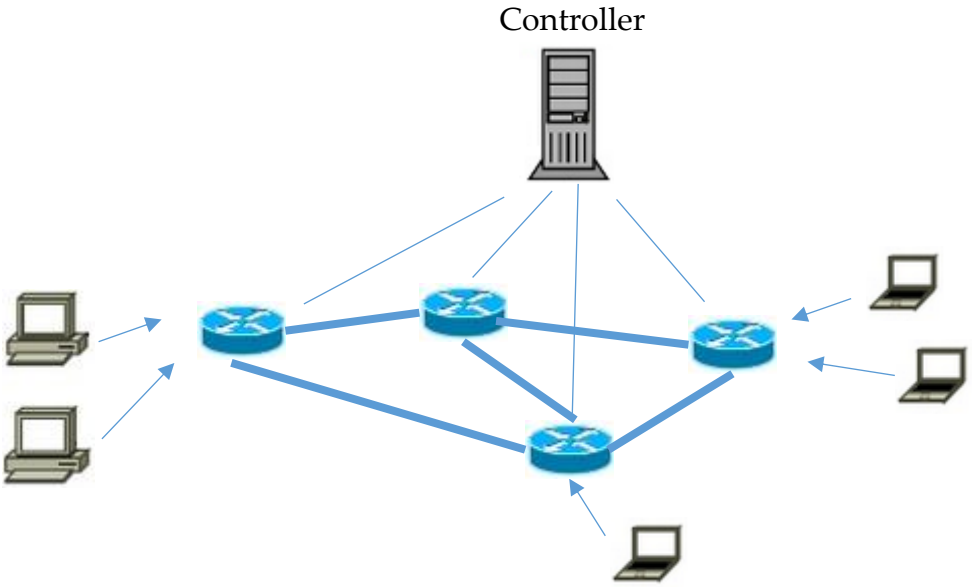

Figure 1.2: Architecture and structure of a SDN

These aspects are key factors to simplifying network management and enabling network evolution and innovation [1]. In spite of the benefits of this new paradigm, the security and dependability of the SDN itself are still an open issue [2]. However, SDNs are already a reality with several application examples. In data center networks they are already replacing traditional networks, and several examples of their use in service provider networks and campus networks are also already in production. Standards like OpenFlow (a southbound API between the control plane and the data plane) are already implemented in modern equipment, and several equipment vendors and network operators are committed to the development of controllers and Network Operating Systems ecosystems.

1.2. Objectives and contributions

The main objective of this thesis is to study the state of the art in the development of software applications for software defined networks. In this process we will:

- Present the Software-Defined Networks paradigm and identify its characteristics;
- Present the OpenFlow protocol and its characteristics;
- Explain and document how to program SDN applications that use OpenFlow.

The main contribution will be the presentation of a systematized view of how to implement applications for SDNs in a current state of the art controller.

1.3. Thesis Layout

This thesis is structured into three chapters: the present Chapter 1 is a brief introduction of the proposed work, its motivation and objectives. The second is the state of the art, that addresses the currently available solutions in SDNs and especially in OpenFlow, which is intended to explore its evolution, extension and how it can be used, and also its advantages and disadvantages. In Chapter 3, a guide of how to develop an application for a SDN is documented.

2

2. State of the Art

2.1. Typical Network Architecture

Network devices are responsible for receiving the packets, checking the packets headers, and deciding where to route them. There are network devices operating in Layers 1, 2 and 3. In Layer 1 we have a HUB, which only retransmits what it receives, without checking anything, thus working as a signal repeater. In Layer 2 there are Switches, which interconnect computers from the same network. Finally, in Layer 3, we have a Router which interconnects computers and other devices from different networks.

On these networks, the Switch has two levels: the Control Plane, that checks the packets' headers and decides where to route them; and Data Plane, that forwards traffic to the next hop, according to the Control Plane's decision. So, when a packet arrives at a Switch, the Control Plane examines it, decides where to send it and communicates the decision to the Data Plane, which then proceeds to sending the packet to where it is supposed to go.

This communication between the Control and Data Planes happens inside the switch, which has vendors' closed software and hardware. This limits the ability to engineer and manage traffic across equipment from several vendors. On top of that, the algorithms of the Control Plane have to be configured

before the switch is installed, and if the network administrator wants to add another rule he has to access the switch and change it manually.

The architecture of current networks is based essentially on the use of layered networks which divides the information in different units of varying sizes called packages. These packages are generally lower than the size of the original message and are sent by alternative routes [3]. This characteristic makes for greater effectiveness in the communication network, because if there is a network failure the data flow is not interrupted [4].

The current architecture of networks meets the following specifications [4]:

- Connectivity;
- Generality;
- Heterogeneity;
- Robustness;
- Accessibility

2.2. The Road to Software-Defined Networks

One of the first examples of control centralization in a network came in the 1990s, when AT&T introduced Network Control Point (NCP) [5], with the separation of the voice and the signaling channels in the telephone network.

In this system, when someone wanted to make a call, the signaling of the call went straight to a NCP, and then it would make a query to a back-end Database containing all the routing information that would reply with the asked information. This offered the possibility of services on demand, the easy and quick input of new information (since it was only necessary to update the

Database), and shorter holding time due to the fact that the NPC could know the status of a line, so it would route the call through the quickest line available. This type of control was called Central Control and is still used today by AT&T.

Computer Networks nowadays involve many kinds of components like Switches, Routers, Firewalls, Network Address Translators, Server Load Balancers, among others. That conjunction of equipment makes the networks complex and difficult to manage.

The architecture of a network Switch has two levels: data plane, which deals with packet switching; and control plane, which does the routing decisions for the packets. The decision process of switching or routing and the actual data plane are, typically, on the same device. This creates a tight relation between the control and data planes, which makes tasks like debugging configuration problems very difficult.

There are several kinds of Switches in different layers of the OSI model. In layer 1 the Switch is used as signal repeater and regenerator, in layer 2 it interconnects devices that belong to the same network, and in layer 3 it interconnects devices from different networks.

The networks' configuration process consists on having the network's administrator configure each device individually, using different interfaces that diverge from vendor to vendor, and sometimes even from device to device, both from the same vendor.

The exponential growth of networks led to scalability problems. Adding many devices means the network will become much more complex, resulting in much more work for the administrators to configure the devices, trying to guess the traffic patterns (which are static), and readjust all the network. So, adding millions of new devices becomes impossible due to the work involved.

Therefore, this mode of operation has blocked innovation, caused poor performance due to redundant operations at different protocol layers, increased complexity, created major scaling problems, and raised the costs of running a network.

The first attempt to solve some of these issues was Active Networks, of which there were two kinds: Integrated and Discrete. In the integrated approach, each message contains a code which will be evaluated by every programmable middle box (or active node) and then ran in an execution environment of those middle boxes. On the discrete kind, the code is installed in the active nodes, and the packets are dispatched to the appropriate code block based on the values on the packet headers.

These networks were a great step towards Software-Defined Networks, but came too soon. At the time, there was no application for these Networks since data centers and cloud computing did not exist. Other major issues that prevented the Active Networks' success was the idea of having code passing through the network, which raised many security concerns and the need of hardware and firmware upgrades due to lack of operability with the existing networks.

2.3. Network Virtualization

With Network Virtualization [15] it is possible to create several logical network partitions on a physical network infrastructure, isolated from each other. These logical networks can provide the same services, similarly to an ordinary network, and can be different from each other, in spite of sharing the same resources and coexisting in the same physical network infrastructure, as show on the **Figure 2.1.**

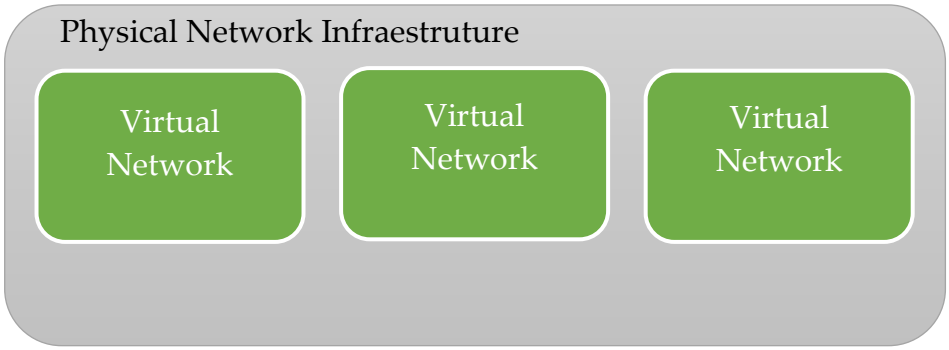

Figure 2.1: Virtual Network integration in physical infrastructure

For the user, there is no difference in using a logical partition or a physical network, but for network managers it is a big improvement in terms of managing networks. There are several resources available to a network such as switches, routers and virtual machines. These resources can be used by several logical partitions at the same time, without interfering with each other, and can be reallocated in real time in order to obtain better efficiency. If one instance of the network has a lot of resources allocated, but is not using all of them, they can be reallocated to a struggling partition with a lot of traffic at that moment. So with virtualization, the use of a physical network infrastructure is maximized, because instead of one network, it is possible to have multiple other network instances, so the costs are decreased as well, because the network resources are shared by several networks as needed at any given time.

2.4. Common problems of Traditional Networks

The growth of networks, as well as their importance, is undeniable. With this growth the amount of hardware needed is larger, therefore some scalability problems occur. Networks are configured using commands or network management systems [27], so it is a very time consuming work to set up a new network, or to change a certain parameter in every hardware of the network.

The rise of the cloud, mobile computing, data centers and other trends like Internet of Things demands good performance from the networks, and because they require timely adjustments to change their configurations, they can be considered essentially static and inefficient at deploying dynamic services such as ensuring resources to applications when they need it the most. This means the adaptation of these networks to new services may take years, since they need feature upgrades, architecture adjustments or introduction of new devices to meet new service requirements. For example, the traditional Layer 2 VLAN mechanism of a cloud data center with virtual machines and virtual networks is required to run new protocols on switches to meet scalability requirements; however, the physical devices involved cannot adapt to these requirements quickly enough.

Software-defined networks are a viable possibility to solve the problems presented above and will be described ahead. These networks are able to provide better visibility into the network, making troubleshooting the network easier. Anomalies from the network can also trigger actions to identify where the problem is and possibly solve the issue.

In this case of the cloud data center, software-defined virtual switches combined with overlay networking can bypass the limitations of physical switches and still satisfy the scalability requirements of the network.

2.5. Software-Defined Networks

Traditionally, networking relied and evolved over a non-transparent distributed model for the deployment of protocols and configuration of devices that resulted in complex protocols and intrinsically difficult configuration of network devices [5].

Software-Defined Networks present a new way of thinking in networking, shifting the complexity of protocols and management functions from the network devices to a general purpose logically centralized service. The motivation behind this decoupling is to simplify the operation of the network for users.

Software-Defined Networks (SDN) are based on virtualization and allow software to run separately from the hardware. The routing decisions (Control Plane) are separated from packet switching (Data Plane), leaving the decisions regarding where traffic is sent to be handled by a centralized Control Plane, which knows the state of the entire Network at any given time. This separation of Control and Data Planes gives the Switch more process capability and can virtualize the network environment, offering at the same time a much more programmable network due to the fact that only one element has to be configured: the centralized controller. This also allows network developers to develop new protocols that can control the Data Plane, and to test it without configuring every switch of a given network.

Understanding the biggest contributions in the SDN research field allows for a better picture of its composition, benefits, and drawbacks, thus historical changes in networks that have molded the current SDN architecture [5].

SDN contributions can be decomposed in three phases: the introduction of programmable network hardware; the control and data plane separation; and, finally, standardization of the data plane interface [5].

As previously seen, the history of SDN takes us back to when the incredible success of the Internet exacerbated the challenges of management and evolution of the network infrastructure, in which the focus was on innovation in the networking community. However, these innovations were in some cases catalyzed by progress in other areas, including distributed systems, operating systems, and programming languages. Efforts to create a programmable network infrastructure also clearly refer to the long list of discussion about the programmable packet processing at high speeds. Based on the above, the history of SDN can be divided into three main stages: (1) The first stage concerns the ideas of active networks (mid-2000) in which functions that were programmable were introduced, and originated more creativity and evolution; (2) The second step concerns the separation of data and control plane (mid 2001). Finally, (3) the third step relates to the emergence of OpenFlow, considered an open interface to make the separation of control plane and data plane [6].

In addition to that, an important aspect, as seen previously, is that the Software-Defined networks constitute a much larger universe than the one defined by the OpenFlow.

OpenFlow provides a simple solution for creating multiple virtual networks on a physical infrastructure, where each network consists of switches and routers. But the paradigm of Software-Defined Networks makes it possible to develop new network applications, something that was thought in the past, in traditional networks [13].

In SDN there is only one communication protocol between different network elements, and the controller manipulates the flows in the network. The OpenFlow protocol was chosen in this work because it has a more advanced stage of development and provides a pattern of open communication. [7]

2.6. SDN Architecture

Like in a traditional network, a SDN is composed of network devices. The main difference between them, is that in SDN the network devices are simple processing elements without embedded control or software to take autonomous decisions. This means the network intelligence is removed from the data plane devices to a centralized logic [1].

To ensure compatibility of configuration and interoperability between different control and data plane devices, these new networks are built on open and standard interfaces such as OpenFlow [1]. In the following figure is displayed a view of an SDN architecture.

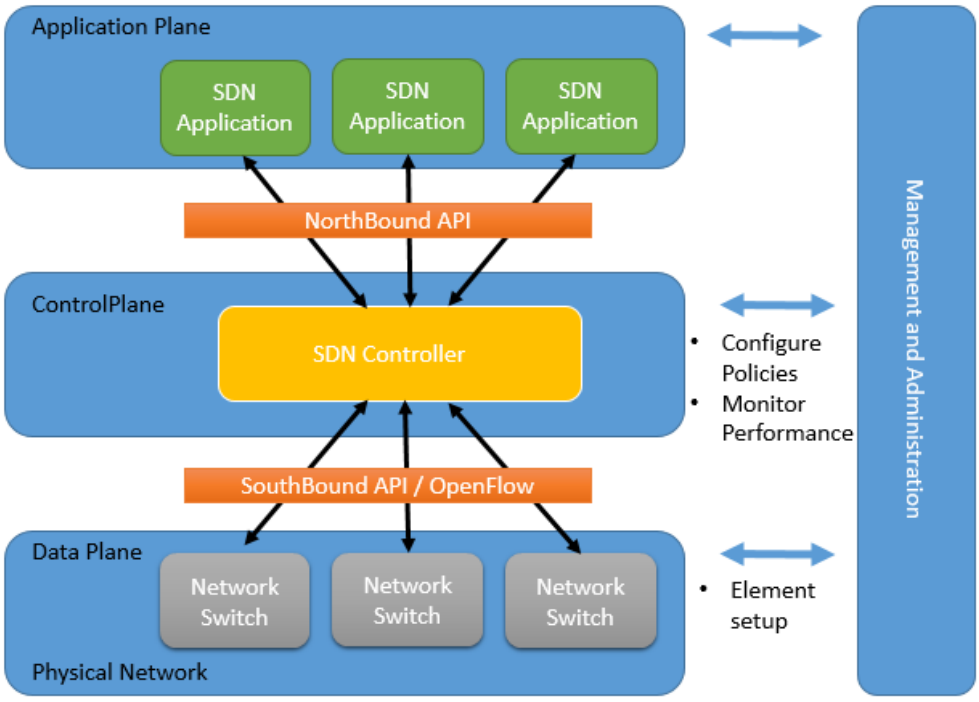

Figure 2.2: SDN architecture

SDNs have three layers, as shown in Figure 2.2. At the top is the Application Layer, where are the SDN applications, which are programs for

network virtualization, network monitoring, intrusion detection (IDS) and flow balancing that communicate with the SDN Controller. This layer is completely isolated from the physical network.

Following, comes the Control Plane to support data management applications and the set-up containing the "brain", the SDN controller, making the routing decisions. It provides abstractions, essential services and common application programming interfaces for developers [2]. Between Application Plane and Control Plane, are the northbound application program interfaces (API), which provide a channel that allows the SDN Controller to send instructions to the applications running over the network, and help network administrators to shape traffic and deploy services.

The last layer is Data Plane, the physical network containing the switches and other network elements, used to forward packets [8], respecting the SDN controller rules and policies. Between the SDN controller and the physical network are the southbound application program interfaces (API) that provide a channel by which instructions are sent to the devices to program them. These APIs use a routing protocol and enable the SDN Controller to make instant changes to face with real-time demands.

In this architecture there are two main elements, routing/forwarding devices such as switches and the controller. An OpenFlow forwarding enabled device is based on a set of flow tables. Each entry of a flow table has three parts, one matching rule, actions to be performed in matching packets and counters to keep the corresponding packet statistics. When a packet arrives, the lookup process begins in the first table and ends with a match in one of the frames (or a rule is not found) [1].

This separation of Control and Data Planes gives the Switch more process capability and can virtualize the network environment, offering at the same time

a much more programmable network due to the fact that it is only necessary to configure one element: the centralized controller. This also allows network developers to test new protocols without configuring every switch of a network.

With this information and based on an appropriate communication protocol, in this case OpenFlow, the controller sets the specific flow in each element involved, enabling the routing package. However, it is important to emphasize that the packets in the same flow are pre-configured and therefore will not need to perform a new query to the controller.

The SDN and its architecture objectives are facilitating network management and ease the burden of solving network problems through a logically centralized control offered by a network operating system (NOS).

At an architecture level, one of the most important aspects is having the controller as a central controller for the whole network, so there is only a single entity that ultimately manages all network devices. However, if this entity fails or if a malicious person takes control, there can be serious problems because the one who controls the SDN controller, can control the entire network [2].

Another important aspect that needs to be mentioned relates to the controller. In this architecture, as there is a centralized element, all information is concentrated in one place. Each reading operation, after a writing operation, returns an updated value [1].

With distributed elements, it is important to define strategies to ensure the consistency of data updates.

Some controllers have a low semantic consistency, which means that the data updates in distinct nodes can be updated in all nodes of the controller. On the other hand, a strong consistency ensures that all nodes of the controller will read a discounted value after a writing operation [9].

In spite of its impact on the performance of the system, strong consistency offers a simpler interface for application users when compared with low consistency [9].

Another important aspect to be mentioned is the fault tolerance. When a node fails, the other node must assure the operation of the failed node. Until now, despite some controllers having trouble of collision, there are no arbitrary failures, i.e. any node with an abnormal behavior will not be replaced by a more appropriate behavior [1].

However, it is important to take into account, to assure the safety of SDN, the following architecture aspects: availability, performance, integrity and confidentiality [10].

OpenFlow is considered the first standard in SDN, it was the original southbound API and remains as one of the most common protocols.

2.7. Security issues of SDN

The countless promises of a simplified control and a real-time programming offered by SDN constitute incentives for operators to keep the evolution at an accelerated pace. However, these questions raise barriers to safety, and the aim is that this will serve as a complement to secure architecture so that networks are protected against attacks by malicious users [10].

Looking at SDN characteristics, it is possible to verify that the main security problems lie on SDN's greatest benefit, i.e. the programmability and logical network centralization.

Software-Defined Networks threats can be divided into two groups: (1) the threats that are specific to SDN and are not present in traditional networks, and (2) the threats that are not specific to SDN.

First the non-SDN-specific threats. Compared with traditional networks, the separation of the control and data planes enables multi-tenancy and programmability, and introduces centralized management into the network architecture. From a security perspective, the ability to share and dynamically operate the same physical network is one of the key security-related differences between SDN and traditional architectures. As such, SDN security issues relate to the new control plane model, and more specifically to securing inter-component communication, and controlling the scope of applications and tenants through specific APIs and access policies [10].

While it may sound like there are a number of obstacles to overcome, the programmability and centralized management brought by SDN enables a much greater level of autonomy to mitigate any security breaches [10].

Another problem is the falsification of a flow. The traffic flows can be forged or faked by a faulty (non-malicious) device or by a malicious user that can be used to attack switches and controllers.

A traditional network is managed individually, but a SDN allows a coordinated management, resulting in a more flexible distribution process. While there is a risk of the SDN control plane becoming a bottleneck. An attacker can use network elements to launch an attack against the switches and controller resources. The solution to the problem presented undergoes the use of intrusion detection systems with support for runtime root- cause analysis to help identify abnormal flows. Another security threat is the vulnerabilities explored in switches [2, 10]. An attacker could theoretically gain unauthorized physical or virtual access to the network or compromise a host that is already connected to the SDN and then try to perform attacks to destabilize the network elements.

For this threat, the solution is to use software attestation mechanisms or mechanisms to monitor and detect abnormal behavior of network devices using switches. Recovery mechanisms are very important in a network, i.e., when there is a network problem it is important, of course, to understand their cause and to recover by resetting. For that purpose, reliable information is needed from all components and domains of the network [2, 10]. Finally, related to non-specific threats it is important to refer that the created logs should be stored in a remote and secure environment [2].

Concerning SDN specific threats, it is important to refer to the form of communication between the three planes, namely the existing communication between both controller and applications. Attacks performed here can be used for data theft [2].

As it was said above, the communication between the three planes represent the most crucial link between controllers and forwarding devices [2]. However, despite this being an important element in an SDN, it also represents a threat for being a weak link in safety, compromising the communication. In fact, compromised OpenFlow-enabled forwarding devices can be used for man-in-the-middle attacks that are nearly impossible to detect [11]. However, there are efforts to fight security threats, including security of SDN controllers, such as creating security domains to isolate applications, [12], the security of SDN is still reduced to the optional use of TLS. It is important to mention that the use of TLS is a good start for the construction of safer SDN architectures because it is one way of making it easier to identify and fix security issues by reducing the complexity of architecture SDN [12].

2.8. OpenFlow

2.8.1. OpenFlow Protocol

The OpenFlow is a standard protocol which is managed by the Open Network Foundation (ONF), a user-driven organization who wants to promote SDN, leading to a worldwide adoption of this kind of Networking. ONF is responsible for managing and publishing OpenFlow specifications.

In a conventional network model, the decisions regarding where an incoming packet should be sent and fast packet forwarding occur in the Switch. An OpenFlow Switch allows the intelligence to be made on a centralized separate controller. The OpenFlow Switch and Controller communicate via the OpenFlow protocol, which defines messages, such as packet-received, send-packet-out, modify-forwarding-table, and get-stats. This means the policies and rules can be changed as network and application requirements change, and can be done immediately, being automatically propagated throughout the network. This makes network management much easier.

OpenFlow [16] protocol allows a network administrator to quickly change the several flow-tables in the different switches and routers, which allows to define different traffic flows with almost on-the-go. It also enables the Service Providers to maintain their Production Flow running, while having the capability to run another flow working completely isolated, without the possibility of compromise the Production Flow. This second flow can be used, for example, to create the opportunity to new researchers of testing new security models, addressing schemes or even new routing protocols, since the researchers control the routes of their packets and process received on their own flows.

Other benefits of OpenFlow are:

- More flexibility and control of software and simulation.
- More speed and scalability of vendor hardware.
- Vendors do not need to expose their closed implementation.

OpenFlow had several versions. A presentation of the evolution of the protocol will be presented.

2.8.2. OpenFlow characteristics

To better understand how OpenFlow [26] protocol works, several concepts should be described before:

Packet - an Ethernet frame, including header and payload;

Flow Entry - An element in the flow table used to match and process packets. It has match fields and a list of instructions with one or more actions;

Flow Tables - Contains at least one flow entry. Since OpenFlow version 1.1 a switch can have more than one flow table enabling the pipeline search, which will try to match to the first flow table, then the second, and so on. A flow table can have a flow entry with an action instructing for the packet to be immediately sent. In this case, the switch will not try to match the packet with the remaining flow tables;

Match or Match Field - A field to which a packet is compared including packet headers, the ingress port, and the metadata value. It is possible to match every packets by wildcarding this field. Usually when a packet is matched, an action is assigned to that packet.

Action - The element that can forward the packet to an outport of the switch. It can also change the content of packet, like decrementing the TTL field. Multiple actions can be applied to one packet, and if two actions change the same

field, when the packet leaves this field will have the value from the last action. A list of actions can be found at [26];

Action Set - a set of actions associated with the packet that are accumulated while the packet is processed by each table and that are executed when the instruction set instructs the packet to exit the processing pipeline;

Instruction - instructions describe the OpenFlow processing necessary when a packet matches the flow entry. An instruction either directs the packet to another flow table, or contains a set of actions to add to the action set, or contains a list of actions to apply immediately to the packet;

Action Bucket – used in groups. It is a set of actions and associated parameters that will be applied when a packet is sent to a group;

Group - a list of action buckets that will be applied to the packets. It has the means to choose which buckets will be applied to each packet

2.8.3. OpenFlow Switch

At first, in OpenFlow 1.0, an OpenFlow Switch contained two main components:

- A Secure Channel necessary to connect to the SDN controller through the OpenFlow protocol. With this protocol, the controller can add, update, and delete flow entries.
- A Flow Table that contains several flow entries, and does packet lookups and forwarding.

In OpenFlow 1.1 it was introduced the possibility to have more Flow Tables, and a Group Table, which allowed the Switch to receive a packet from a sender, and forward it to several destinations.

A flow table consists of flow entries, and each flow entry has the components shown in Figure 2.3.

There is a search made to all packets arriving at a switch, to see if there is a match at the flow table. The flow table contains a set of flow entries, and each of them has three components: *Header Fields* (to match packet headers), *counters* which are updated when a packet is matched with this flow entry, and a set of *actions* to apply to matching packets (ex.: forward the packet to a specified port, or forward the same packet to more than one port).

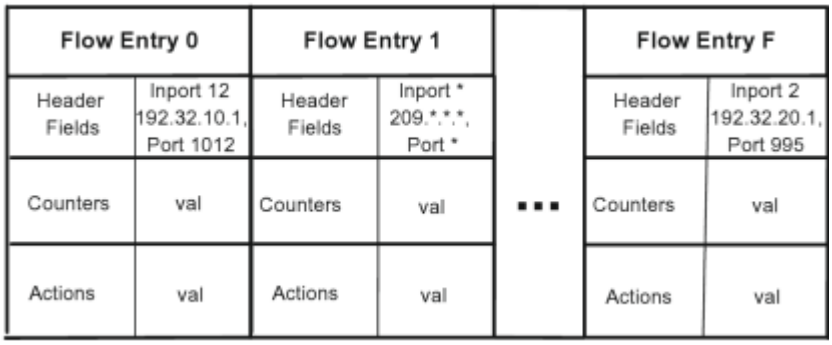

Figure 2.3: Set of flow entries

The SDN Controller determines how to handle packets that did not find any matching flow entry. These packets are sent by the switch through the secure channel, and then the controller adds a flow entry on the switch for that kind of packets, deletes or updates a flow entry or drop the packet.

In OpenFlow 1.1, the Switch was given the possibility of having two or more flow tables and a group table, as is shown in the Figure 2.4.

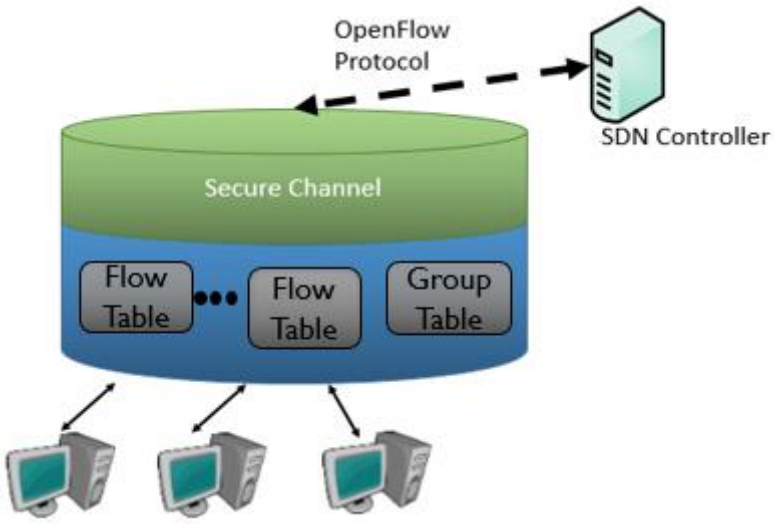

Figure 2.4: Architecture OpenFlow

The process of matching a packet to a flow entry starts at the first flow table, proceeding to the next tables. When a match is made, an action on the packet may be performed. If a packet does not match with any flow entries, depending on the configuration, the packet could be sent to the controller over OpenFlow channel, or dropped.

The group table contains group entries, and each group entry has a list of Action Buckets (which are a list of actions), therefore, a group entry has a list of list of actions.

The key message types traded between the controller and a switch is show on **Table 2.1.**

Message Type	Sent by	Description
Request features	Controller	Asks a switch for its configuration (i.e. port information)
Request stats	Controller	Asks a switch port or flow statistics (i.e. byte and packet counters)
Packet Out	Controller	Send a specific packet through a specific out port
Flow modification	Controller	Insert, update or delete a flow entry in the switch's flow table
Features reply	Switch	Switch description of its features (i.e. ports)
Stats reply	Switch	Report of port or flow statistics from the switch
Packet_IN	Switch	Sent when a packet arrives and has no matching rule, or has a rule whose action directs the packet to the controller
Flow removed	Switch	Notification that a flow entry was removed due to a request from the controller or due to timer expiration

Table 2.1: Summary of the key OpenFlow messages

2.9. Building SDNs for simulations using virtual software switches

2.9.1. Mininet

Mininet [17] can create a network of virtual switches, controllers, links and hosts. Hosts run Linux and switches support OpenFlow.

With Mininet it is possible to have a simple and inexpensive way of having a network running in a single computer and can be used for research and

development. It is possible to implement simple or complex topologies and find the best performance possible for a network, with a given hardware. It has a Command- line Interface that is topology-aware and OpenFlow-aware which is perfect for debugging.

Mininet's networks run code used on real networks that is why it can be moved to a physical network with minimal changes in order to evaluate the real performance of the implementation, which may be similar to networks' performance on Mininet.

Instead of virtualizing computing resources Mininet uses process-based virtualization to run many hosts and switches on a single OS kernel. So it is faster, offers the possibility of having more hosts and switches, provides more bandwidth and is easily installed than others emulators that use full system virtualization. Compared to hardware testbeds, Mininet does not need any money to test a network and it is quickly reconfigurable and restorable.

Mininet has some limitations. It has resource limits, the server resources have to be shared and balanced between the virtual hosts and switches. Using only Linux kernel for all hosts means it is impossible to run software that depends on other system kernels.

2.9.2. Floodlight Controller

As explained before, every Software-Defined Network needs a controller. A controller is responsible for managing flows on the switches' flow tables, and can do it remotely and on real time. SDN controllers are the "brains" of the networks because they can tell any switch / router where to send packets.

The chosen controller was Floodlight [18]. Other options were ONOS or OpenDaylight but these two are more complex and difficult to set up than needed for the purpose of this thesis.

Floodlight is an open-source Controller. It is easy to set up, having minimal dependencies and is user friendly and developer friendly because it uses a module system (written in Java) that makes it easy to extend, adapt software and develop applications. It has Representational State Transfer Application Program Interfaces (REST APIs), that can be written in any language and exchange information with an external entity at runtime. This controller can work with OpenFlow and non-OpenFlow networks.

The Controller performs the typical network operations, monitoring the network and updating flow tables in the switches while having applications, built as Java modules or over the Floodlight REST API that can be written in any language, realizing other features according the user needs over the network.

When Floodlight starts [19], the controller and the module Java applications start running. These Java applications are loaded in floodlight's properties file. The REST APIs are available via the REST port and can retrieve information or send http REST commands to the controller in order to invoke a variety of services.

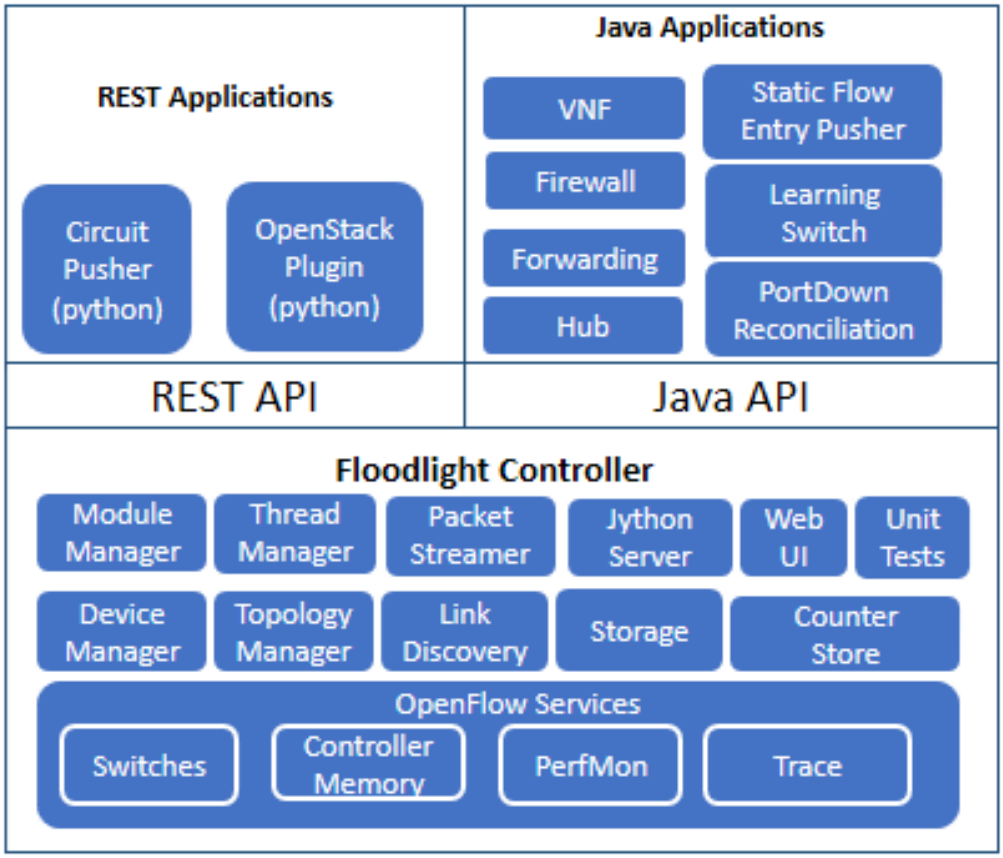

Figure 2.5: Set modules in Floodlight controller

The Module Applications implemented in floodlight are represented in Figure 2.5.

- **Virtual Network Filter (VNF).** A MAC-based network isolation application which is not enabled by default. Exposes a variety of REST API that allow add, remove and inquire virtual networks.
- **Firewall.** Application that can apply Access Control List (ACL) rules to allow or deny traffic based on a specified match. Exposes a variety of REST API that allow to enable or disable the firewall and add, remove or list rules.
- **Forwarding.** A default reactive packet forwarding application which will forward packets between two devices. Since Floodlight is designed to work in networks that contain both OpenFlow and non-OpenFlow

switches Forwarding has to take this into account. The algorithm will find all OpenFlow islands that have device attachment points for both the source and destination devices. FlowMods will then be installed along the shortest path for the flow. If a PacketIn is received on an island and there is no attachment point for the device on that island the packet will be flooded.

- **Hub.** An application that flood any incoming packet to all active ports, except the port from which the packet arrived.
- **Static Flow Entry Pusher.** An application that can install a specific flow entry in a specific switch. Exposes a variety of REST API that allow to add, remove and inquire flow entries.
- **Learning Switch.** A L2 learning switch. Exposes a REST API that allow to list the current switch table.
- **Port Down reconciliation.** An application to reconcile flows in case a port or link goes down.

Some examples of REST applications that use floodlight REST APIs are:

- **Circuit Pusher,** that can create a permanent flow entry on all switches in route between two devices based on IP addresses with a given priority.
- **OpenStack,** that allows Floodlight to run as the network backend for OpenStack using a Neutron plugin, exposing the network-as-a-service model.

2.10. OpenFlow Concepts in Floodlight Controller

The controller will be able to interact with network switches writing messages to the switch and processing the messages from the switch.

The OpenFlowJ-Loxigen is a single, common, version-agnostic API that support OpenFlow versions from 1.0 to 1.4. It provides builders for several OpenFlow Concepts. Before explaining how to develop a network, it makes sense to explain these concepts, and to show a simple example about them.

2.10.1. Factories

Each OpenFlow version has a factory that can build all types and messages for that version. This library provides an easy method to create OpenFlow Messages, Matches, Actions, FlowMods, etc. It uses builders that handles the low-level details such as Message lengths. The switches that connect to Floodlight contain an OpenFlow factory compatible with the OpenFlow version running on the switch. It is possible to have several switches, running different versions of OpenFlow, communicating with the controller. In this case OpenFlowJ-Loxigen handles the low-level protocol differences in the background, and the user can write functions without concerning the version of each switch. In order for this to happen, the switch is exposed as an IOFSwitch, which has the function getOFFactory() to return the appropriate factory for the OpenFlow version running on the switch. In the initial handshake between the switch and the controller, a reference to an OpenFlow factory of the correct OpenFlow version is given to the switch. To create a message, the user only needs to reference the factory from the switch, create the builder and send it to the switch. Reference the OFFactory needed is mandatory before starting to compose an OFMessage to send or answer to a switch. In a case where the network

administrator knows there is only one version used in the network (i.e. OpenFlow 1.3), it is possible to refer that particular factory by doing the following:

```
OFFactory my13Factory = OFFactories.getOFFactory(OFVersion.OF_13); /* Get an OpenFlow 1.3 factory. */
```

We can get the correct factory from a switch by creating a new IOFSwitch object with the switch's MAC address, and then use the method getOFFactory().

```
IOFSwitch mySwitch = switchService.getSwitch(DatapathId.of("00:00:00:00:00:00:00:01") );
OFFactory myFactory = mySwitch.getOFFactory(); /* Use the factory version appropriate for the switch in question. */
```

It is also possible to get the proper factory from an existing object generated by an OFFactory itself. OFVersion and OFFactory classes provide functions to do this as shown below.

```
OFVersion flowModVersion = myFlowMod.getVersion(); /* We assume myFlowMod has been already constructed */
OFFactory myFactory = OFFactories.getFactory(flowModVersion); /* Get the OFFactory version we need based on the existing object's version. */
```

OFFactories expose the builders for almost all of the OpenFlow concepts. That is why the first step on composing messages is to refer the proper factory, which will provide the correct builders for the OpenFlow version running in the switch.

2.10.2. Matches

Matches are related with the characteristics of packet header fields. One use of Matches happens when the controller wants to insert / update a flow in a switch. This modifications requests are sent via a specific type of message: FlowMod. Using OpenFlowJ-Loxigen's builder, to construct Matches is a simple and direct process.

```
Match myMatch = myFactory.buildMatch()
        .setExact(MatchField.IN_PORT, OFPort.of(1))
        .setExact(MatchField.ETH_TYPE, EthType.IPv4)
        .setMasked(MatchField.IPV4_SRC,
IPv4AddressWithMask.of("192.168.0.1/24"))
        .setExact(MatchField.IP_PROTO, IpProtocol.TCP)
        .setExact(MatchField.TCP_DST, TransportPort.of(80))
        .build();
```

2.10.3. Actions

Actions differ from one OpenFlow version to another, therefore is mandatory first to get the correct version, which can be provided by the OFFactory. An action set is intended to apply to a packet. It can discard, modify, queue or forward an incoming packet.

2.10.4. Instructions

Starting in OpenFlow 1.1, each flow entry of a flow table has a set of instructions to apply to all matching packets. These instructions can be used to modify the packets state, forward the packet to a particular port or forward the packet to another table or group. Each packet maintains an action set, which contains a set

of actions to apply to the packet when no further table processing can be accomplished. There are several Instructions:

- Write and clear instructions, which provide ways of manipulating the action set.
- Apply instruction that performs actions immediately.
- Goto instruction provides a mechanism to choose the next flow table for processing.
- Meter instruction allows the application of a rate limiter to the flow.
- Experimenter instructions provides a structure for custom extensions to instructions.

2.10.5. **FlowMods**

Like the concepts before, FlowMod also refers to one OpenFlow version only, therefore it is necessary first to know it. FlowMod will insert, update or remove a rule for a specific type of packets in the switch's flow table.

2.10.6. **Groups**

OFGroups allows to make more complex operations in an OpenFlow switch such as duplicating packets or applying different sets of OFActions to a single packet. To allow this, the structure of an OFGroup is a list of lists of OFActions. Those lists are called buckets, therefore an OFBucket contains a set of OFActions. There are four types of OFBucket:

- OFGroupType.ALL: Provide each OFBucket with a copy of the packet, and apply the list of OFAction's within each OFBucket to the OFBucket's copy of the packet.

- OFGroupType.SELECT: Use a switch-determined (typically round-robin) approach to load-balance the packet between all OFBuckets. Weights can be assigned for a weighted round-robin distribution of packets.
- OFGroupType.INDIRECT: Only a single OFBucket is allowed, and all OFAction's are applied. This allows for more efficient forwarding when many flows contain the same action set. Identical to ALL with a single OFBucket.
- OFGroupType.FF: Fast-Failover. Use a single OFBucket and change automatically to the next OFBucket in the OFGroup if a specific link or a link in the specified OFGroup fails for the active OFBucket.

The user has to configure the switch with the amount, and type, of OFGroups wanted. OFGroups can be added, modified or deleted through the OFMessages: OFGroupAdd, OFGroupModify and OFGroupDelete which can be composed and then written to a switch.

2.10.7. **Packet-Ins**

OFPacketIn is an OpenFlow object which also can be done by getting an OFFactory. When a switch receives a packet which does not have its destination on the switch's flow table, it sends a message to the controller as a packet-in. The controller then can process the OFPacketIn and can get useful information such as the Match corresponding to the packet within.

2.10.8. **Packet-Outs**

OFPacketOut is an OFMessage that allows the controller to send a packet to the switch with instructions to be injected into its data plane. After the switch received an OFPacketOut, it should take the payload from the packet and send it

out through whichever port(s) the OFPacketOut specifies. This kind of packets should also contain some data validating the packet itself.

2.10.9. Meters

Meters allow to monitor the ingress rate of traffic from one flow, before they leave the switch. Using the instruction goto-meter, the packets are sent to a meter which can perform some operations based on the rate it receives packets. Meters can be compared to flows since they can be managed (installed, modified, and removed) at runtime using OpenFlow. Also, OpenFlow defines an abstraction called a meter table, which simply contains rows of meters. These meters receive packets as input and (optionally) send packets as output.

2.10.10. Collect switch statistics

There are several statistics messages available in OpenFlow that allow the controller to query the switch for information about its flow stats, meter stats, queue stats, aggregate stats, table stats, and port stats. It is very useful to know about this information, however these statistics cannot be shown in real-time since the query from the controller has to go to the switch and then the switch will answer the query. When the response arrives at the controller, the statistics are probably already outdated because they were verified when the stats reply message was being written. For many applications, this inaccuracy is tolerable but the use of reactive algorithms that rely on these statistics need to be careful with this delay.

Statistics can also be used to compute bandwidth. To determine bandwidth consumption, we can use byte counters returned at two points in time. The difference between these two counters divided by the time elapsed

between the "snapshot" point of each counter value tells us the bandwidth. But this is not as easy as it seems because there is no timestamp of when the statistics were taken. To compute the time elapsed between two reads, the controller can only rely on when the stats message was sent or when the reply was received. This could lead to an inaccurate time interval because there are delays in the network and they can vary. There are two ways to work around this problem:

- Issue lots of stats requests and compute the bandwidth frequently to attempt to keep up with almost real time bandwidth consumption. The problem is the delay, which can vary and alter the bandwidth values;
- Issue less frequent stats requests and compute and update the bandwidth less frequently. This solution will delete the delay problem, but will not allow to monitor the bandwidth consumption in real-time.

It is a network administrator's decision whether to take the first or the second approach.

2.11. Conclusion

Software-Defined Networks (SDN) constitute a new paradigm for the development of research computer networks. These networks gained importance in recent years on the researchers' part, including OpenFlow, which made a new approach possible. However, Software-Defined Networks go far beyond OpenFlow, opening new perspectives in terms of abstractions, control environments and network applications that can be developed simply and free of limitations of current network technologies.

3

3. Developing Applications for Software-Defined Networks

The goal of this thesis is to present a practical guide to develop an application capable of managing a Software-Defined Network (SDN). As said before, the Floodlight controller will be used, using OpenFlow to communicate with the switches, and Mininet to set a virtual network.

In this Chapter we will document the steps necessary for the development of an application that runs in the floodlight controller which could control a software-defined network. The communication between the application in the controller and switches is performed using the Openflow API. The controller needs to be able to process information received from networks' switches, and also need to send messages so that they can operate as the user wants. It will be described how to configure the controller using Floodlight's libraries.

To show some of the capabilities of a SDN, two examples of applications will be used. For each example it will be described how to develop, how to simulate and how the results can be verified.

3.1. Example 1

The application in the first example implements the behaviour of a learning switch. A learning switch learns paths such that they do not flood (broadcast) excessively. They do this by learning the paths to the foreign MACs, and, upon receiving a packet destined to the foreign switch, they will only send out through the correct path.

To better explain, let us use the topology presented in Figure 3.1. Before any connection between the hosts, the two switches do not know any paths to any hosts. When Host 1 wants to send a packet to Host 4, it sends it to Switch A. Switch A will record which port Host 1 came in from. Since Switch A does not know where the MAC address for Host 4 is, it will flood and send a copy of the packet to both remaining ports. It will reach Host 2, but it will also reach Switch B. Switch B will save off how to get to Host 1 (via Switch A), and flood to hosts 3 and 4.

If, afterwards, Host 3 tries to send a packet to Host 1, Switch B will learn how to get to Host 3, will not flood and will send it directly to Switch A. Switch A will also learn how to get to Host 3 (via Switch B), and forward the packet directly to Host 3.

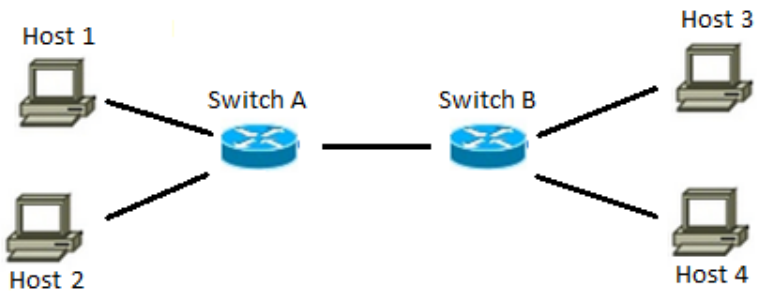

Figure 3.1: Network Topology

We will implement the behavior of a learning switch, and we will use three steps to complete it. The switches in our network will have only one rule: to send a Packet_IN message to the controller when a packet with an unknown destination is received. The three steps are three possible answers that the controller will send to the switches through a Packet_OUT message. We will start with the simplest, and least efficient until the most efficient. The steps represent the response of the controller after the reception of the Packet_IN message.

Step A: The controller sends a Packet_OUT message to the switch with the action set to flood the original packet. Upon the reception of the Packet_OUT the switch floods the packet through all ports except the in port of the original packet. No flow entries are installed in the switch in this step. This is the behavior of an HUB.

Step B: First the controller will check its tables to see if it knows the mapping for the desired destination. If the mapping was already learned by the controller app, the action is a forward action with the output port where the switch should send the packet. If the mapping is still not known the action is to flood the packet, as explained in Step A.

Step C: The controller makes the same verification as in Step B. If the mapping is available, the controller app sends a Flow Modification message instructing the switch to add a Flow Entry rule that matches incoming packets with the learned incoming port / source MAC pair and an action to forward them to the learned output port. If the path was not learned yet, the controller will instruct the switch to flood the packet, as in the previous examples.

Next, the implementation details for each of the steps are described. It will also be explained the setup of a test simulation were the implications of each step in terms of controller-switch communication will be shown by measuring the effective transmission throughput that we can achieve in each case.

3.1.1. Developing the floodlight module

The first step is to create a new class in floodlight. The class should be created in the "src/main/java" directory and it should be on its own Java package, the module can implement a number of different floodlight interfaces. In this example the module implements the *IOFMessageListener* and the *IFloodlightModule* interfaces. *IOFMessageListener* is a message listener that gets notifications when the controller receives a packet. *IFloodlightModule* defines an interface for loadable floodlight modules, it is responsible to provide a template file when a new class is added, with the needed methods (which are empty) for a module to work. This interface is mandatory to develop a module for floodlight. After defining the implemented interfaces a new class is generated with the skeleton of a floodlight module. An example of the template file generated when a new class is created can be found in [21].

This automatic file generation happens if we use Eclipse, with a floodlight project compiled with *ant*. The virtual machine provided in floodlight's site already has the floodlight project compiled. If the user does not want to use this virtual machine, the project can also be imported to Eclipse using the following commands on a terminal:

```
sudo apt-get install build-essential default-jdk ant
python-dev eclipse
git clone git://github.com/floodlight/floodlight.git
cd floodlight
ant eclipse
```

Next, the interface's methods need to be implemented since they are empty. These methods will determine the behavior of the controller.

Finally, the module must be registered in floodlight for it to be loaded. This is done by adding the module name in two distinct files and it will be explained later.

3.1.2. Emulating the network on Mininet

Mininet offers the possibility to initialize the topology via a Python script where the network elements such as switches, controllers and hosts are created and links between the elements are also defined. Some commands, such as *ping*, *pingAll*, *iperf*, etc, can be launched automatically after the network starts, in the python script. Another option is to start Mininet in the CLI (Command-Line Interface) and write the commands as wanted.

For the first example we used the topology represented in Figure 3.2.

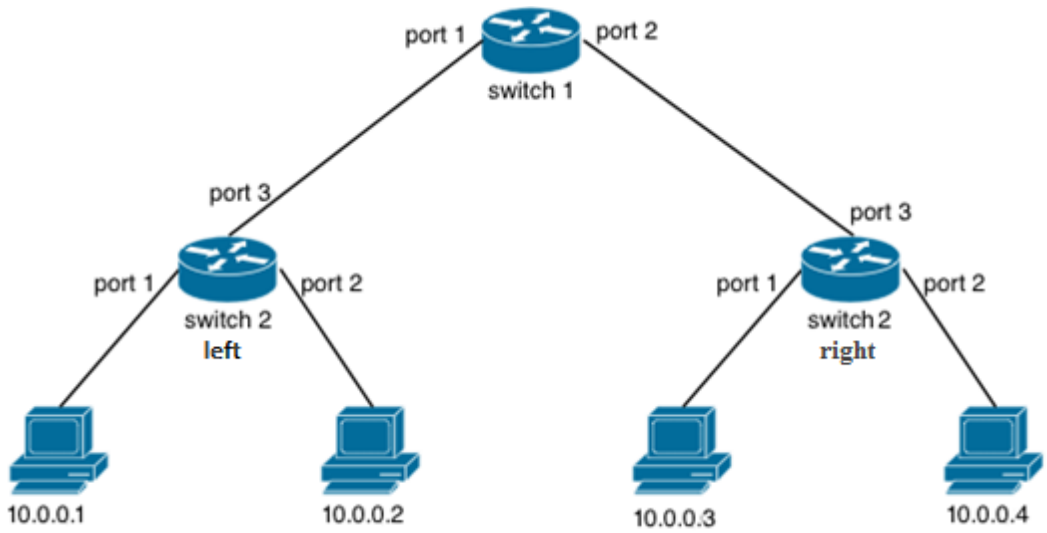

Figure 3.2: Topology used in Example 1

The python script to create this network can be found in [20]. In the file it is possible to see how to initialize the topology by adding the elements and then

the links, how to start the network and two examples of how to run automatically a ping from Host 1 to Host 4 and a *pingAll* instruction. The command examples are only present to illustrate how to do it.

3.1.3. Implementing the behavior step by step

We will start with Step A, where the controller sends a Packet_OUT message to the switch with the action set to flood the original packet. Upon the reception of the Packet_OUT the switch floods the packet through all ports except the in port of the original packet. No flow entries are installed in the switch in this step.

To implement the module we start with the class module skeleton code. The first step is to include the needed dependencies for the code to work. Next, the *IFloodlightProvider* object needs to be declared for registering with the Floodlight main module and a logger object is declared to output the events. This is achieved by adding the following lines in the module:

```
protected IFloodlightProviderService floodlightProvider;
protected static Logger log;
```

Next we need to tell the module loading system that we depend on it, which is done in the *getModuleDependencies* method:

```
public Collection<Class<? extends IFloodlightService>> getModuleDependencies() {
        Collection<Class<? extends IFloodlightService>> l =
            new ArrayList<Class<? extends IFloodlightService>>();
        l.add(IFloodlightProviderService.class);
        return l;
    }
```

The *init* method loads the dependencies and initialize datastructures. It is called early in the controller startup process and it must be modified in order to get the controller instance and to create the logger, adding the following lines:

```
floodlightProvider = context.getServiceImpl(IFloodlightProviderService.class);
//get controller instance
    log = LoggerFactory.getLogger(Example1a.class); //create logger class
```

When the controller receives a Packet-In it should be notified, therefore the listener must be implemented. The method *startUp* is where the message listener is registered by invoking the method *addOFMessageListener* and telling it we want to listen for events of type Packet_IN. The *startUp* method should be edited adding the following line:

```
floodlightProvider.addOFMessageListener(OFType.PACKET_IN, this);
```

It is also necessary to put an ID for the OFMessage listener. This is done in the *getName* call.

```
public String getName() {
        return Example1a.class.getSimpleName();
    }
```

The *received* method is the callback method called when the OpenFlow messages that the module is registered to listen arrive at the controller. In that method we have to add code to handle the reception of the Packet_IN messages. In this case we create a separate method called *processPacketInMessage*, which will

get the information from the Packet_IN and sends a Packet_OUT message to the switch with the Actions set for flooding the packet.
An example of the module is available at [22].

Finally, the module should be registered in floodlight. This is done by adding the following line to the file in the following subdirectory of the floodlight project parent directory *src/main/resources/META-INF/services/ net.floodlightcontroller.core.module.IFloodlightModule:*

net.floodlightcontroller.exe1.Example1a

being *net.floodlightcontroller.exe1* the Java package and *Example1a* the module's name. It is also necessary to add the referred line into the properties file which can be accessed in *src/main/resources/floodlightdefault.properties.*

In Step B, the controller has a hash table where it collects the in port / source MAC address pair. After the reception of the Packet_IN message, the controller searches in the hash table for the destination MAC address. If the destination port for a MAC address is not in the table, it has to be stored and the controller sends a Packet_OUT message to the switch with the action set to flood. Upon the reception of the Packet_OUT the switch floods the packet through all ports except the in port of the original packet. In other hand, if the destination port for a MAC address is found, the controller sends a Packet_OUT message to the switch with the action set to send the packet through a specific port. No flow entries are installed in the switch in this step.

To implement this step's behavior, we start with the class module skeleton code. The methods described in Step A have the same code in this step, but in *init* method it should be added an instance for the hash table. Since we are

working with a hash table, two methods need to be created in order to save new information and to search information from the table. These methods are *addToPortMap* and *getFromPortMap*. Comparing to step A's code, the *processPacketInMessage* method needs to be modified. It needs to get the information from the Packet_IN and search the hash table for the in port and MAC address in order to see if it already has this information or if it needs to save it. Then it will build the Packet_OUT message. If the destination port for the MAC address asked in the Packet_IN is not in the hash table, the Actions set in the Packet_OUT message sent to the switch will have a flooding action. If the destination port is found, the Actions set will have an action with the output port where to send the packet.

An example of this module can be found at [23].

The Step C is similar to Step B with the difference that a flow will be added in case the destination MAC is known by the controller.

The controller has the hash table where it collects the in port / source MAC address pair. After the reception of the Packet_IN message, the controller searches in the hash table for the destination MAC address. If the destination port for a MAC address is not found, it has to be stored and the controller sends a Packet_OUT message to the switch with the action set to flood. Upon the reception of the Packet_OUT the switch floods the packet through all ports except the in port of the original packet. The difference from step B is in case the destination port for a MAC address is found. The Packet_OUT message will still have an action with the specific output port, but now it will also have a FlowMod message which will add a flow entry in the switch, instructing it to send every packet with that destination to a specific output port. To assure the switch always has the updated information of the network, the flow entry in the switch has an

idle timeout. When this timeout expires a message is sent to the controller, which has to analyze the outdated flow entry and send a FlowMod message instructing the switch to delete that flow entry.

To implement this behavior, the modifications will happen in *processPacketInMessage* method which now needs to have the same functionalities as before, but now it also needs to write a FlowMod message. To achieve this, a new method must be developed, called *writeFlowMod*. This method will be called in the *processPacketInMessage* method. As it happens in step B if the destination port for the MAC address asked in the Packet_IN is not in the hash table, the Actions set in the Packet_OUT message sent to the switch will have a flooding action. If the destination port is found, the Packet_OUT message will contain a FlowMod adding a flow entry to that switch. When the flow entry expires the switch will send a message with that information, therefore in the *startup* method, we need to add a listener to this type of messages adding the following line:

```
floodlightProvider.addOFMessageListener
(OFType.FLOW_REMOVED, this)
```

In the *receive* method we need to contemplate this message type and call the *processFlowRemovedMessage* method. This method will send a FlowMod message instructing the switch to delete the expired flow entry:

case FLOW_REMOVED:

```
return this.processFlowRemovedMessage(sw,
(OFFlowRemoved) msg);
```

An example of this module can be found at [24].

3.1.4. Connect Mininet topology to Floodlight controller

After implementing the module on floodlight and creating the python file containing the topology, the controller should be started. Next, opening a terminal and navigating to the folder containing the topology file, the network can be started with the command:

*sudo python *filename.py**

And, immediately, it is possible to see in the controller's log the switches being added. To try the step A, a ping from Host 1 to Host 4 is made in order to force the switch to ask the path to the controller. Opening the python file it is possible to see the following lines:

h1= net.get('h1')

result = h1.cmd('ping -c4 10.0.0.4')

print result

This will make the desired ping automatically, just by running the script mentioned above. If preferred, these lines can be removed and the ping can be done manually in the CLI. The *pingAll* Example next should be removed in order to be clearer to see what is happening.

Opening an Xterm window for any switch ("*xterm s1*" on the terminal for Switch 1) is possible to see the flow entries in the switch. On the Xterm window we can write:

sudo ovs-ofctl dump-flows s1 -O OpenFlow13

Resulting in the following output:

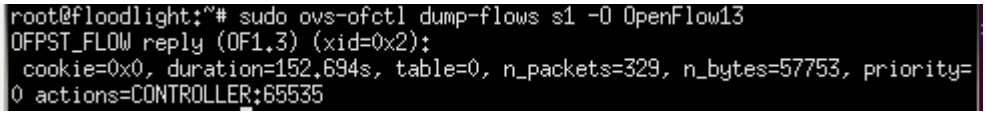

Figure 3.3: Flow configuration of switch S1

It is possible to see a flow with the action: *CONTROLLER.* This flow will direct all packets with an unknown destination to the controller as a Packet_IN message. The controller will process the packet and send a response to the switch instructing it to flood the packet in order to find the correct path.

To test the controller, four packets are sent from Host 1 to Host 4. When the ping is made, it is possible to see that all packets have similar times. To ping Host 4, Host 1 sends a packet to Switch 2 left. The problem is the switch does not know where to send it and sends a message to the controller, the controller processes this message and sends back a message ordering the switch to flood all ports in order to find the path for Host 4. When Switch 2 left floods the packet, it goes to Switch 1, and this switch will now follow the same method as Switch 2 left regarding the controller. At some point, all switches are flooding the network with copies of the first packet sent. The following three packets sent will also be flooded by every switches. For this example, in which was used the floodlight's virtual machine, the ping statistics obtained are shown in the figure below.

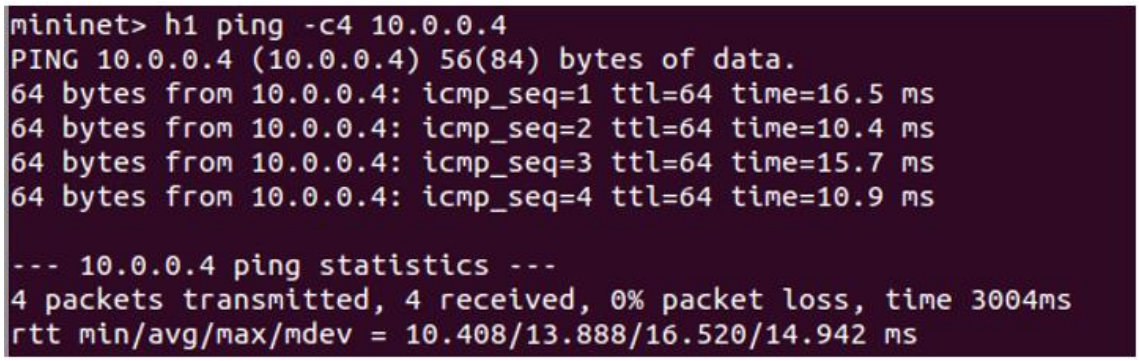

Figure 3.4: Ping statistics for Step A

The packets sent from Host 1 to Host 4 had an average time of 13.888ms. In a trip containing only three switches (S2left-S1-S2right) this solution is not ideal in terms of speed. It is easy to imagine a trip with many hops to be very slow.

Using *iperf*, it runs an *iperf* TCP server on one virtual host, then runs an *iperf* client on a second virtual host. Then they connect to each via a TCP tunnel. Once connected, it is possible to see the speed of the TCP connection between two hosts, in this case Host 1 and Host 4.

```
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h4
*** Results: ['779 Kbits/sec', '1.16 Mbits/sec']
```

Figure 3.5: Bandwidth between Host 1 and Host 4

These results show that a data transfer between Host 1 and Host 4 would be made at a rate in an interval between 779 Kbits per second and 1.16 Mbits per second. This rate will be our reference to compare with the following steps.

Next will be tested the Step B code. Again, first we start the controller and then the topology. Issuing the ping between Host 1 and Host 4 we have the following results.

```
mininet> h1 ping -c4 10.0.0.4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=126 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=0.160 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.142 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=64 time=0.145 ms

--- 10.0.0.4 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3000ms
rtt min/avg/max/mdev = 0.142/31.728/126.667/54.813 ms
```

Figure 3.6: Ping statistics for Step B

Here it is possible to see that the first packet is the slowest. This happens because the controller does not know the network until it receives the first packet. Upon the reception of this first Packet_IN message, asking where to send a packet with an unknown destination, the controller has to learn the network and then

can send to the switch the out port where to send the packet. After the first packet, the controller responds quicker so the packets arrive faster.

Issuing the *iperf*, it is possible to show a big improvement in terms of bandwidth.

Figure 3.7: Bandwidth between Host 1 and Host 4

Finally, it is Step C. Having its code running, we can see the flows inserted in the switch in the Xterm window of s2left:

sudo ovs-ofctl dump-flows s2left -O OpenFlow13

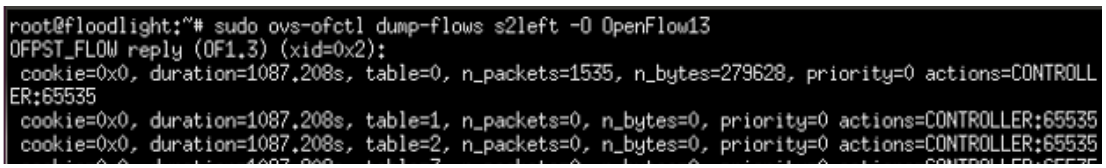

Figure 3.8: Flows in switch s2left

And we can see the flows showing that any packet with an unknown destination will be sent to the controller. When we try to ping Host 4 from Host 1, the first packet takes 131 ms to arrive. As explained in Step B this is the time the controller takes to learn the network. When the controller learns the paths across the network, it sends two FlowAdd messages to all the switches in order to insert the flows containing the destinations ports to be used when sending packets from Host 1 to Host 4 and from Host 4 to Host 1. These messages can be seen in the controller's log with several lines (for several switches) like this:

OFSwitchBase DPID[00:00:00:00:00:00:00:22] adding flow mod OFFlowAddVer13

After the flow entry is added, the packets are forwarded by the switches very quickly without any communication with the controller since the switch

now has a flow entry which matches the packets therefore it knows which port to forward the packets.

```
mininet> h1 ping -c4 10.0.0.4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=131 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=0.184 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.032 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=64 time=0.040 ms

--- 10.0.0.4 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2998ms
rtt min/avg/max/mdev = 0.032/32.833/131.079/56.722 ms
```

Figure 3.9: Ping statistics for Step C

The ping statistics confirm what was said above. The first packet takes longer because it is the time needed by the controller to learn the paths, and then the packets go from Host 1 to Host 4 with great speed, since the switches already know where to send them. If we immediately issue another ping, since the flow is already installed in the switches, we have the results shown in Figure 3.10 confirming that the delivery of the four packets was much faster.

```
mininet> h1 ping -c4 10.0.0.4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=0.042 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=0.046 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.050 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=64 time=0.047 ms

--- 10.0.0.4 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3000ms
rtt min/avg/max/mdev = 0.042/0.046/0.050/0.005 ms
```

Figure 3.10: Ping statistics for Step C with flows instaled

Running the *Iperf* command, it is clear that the bandwidth is higher than step B, which was proved by the times spent by the packets to reach their

destinations. The differences are not too accentuated since the path has only 3 hops.

```
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h4
*** Results: ['21.5 Gbits/sec', '21.6 Gbits/sec']
```

Figure 3.11: *Iperf* results for Step C

Issuing the dump-flows command again on switch s2left, it is now possible to see the two flows added, one used when sending packets to Host 1 and another to send packets to Host 4.

```
root@floodlight:~# sudo ovs-ofctl dump-flows s2left -O OpenFlow13
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x10000000000000, duration=6.937s, table=0, n_packets=7, n_bytes=630, id
le_timeout=15, send_flow_rem priority=100,in_port=1,dl_src=10:00:00:00:00:04,dl_
dst=10:00:00:00:00:01 actions=output:2
 cookie=0x10000000000000, duration=6.935s, table=0, n_packets=7, n_bytes=630, id
le_timeout=15, send_flow_rem priority=100,in_port=2,dl_src=10:00:00:00:00:01,dl_
dst=10:00:00:00:00:04 actions=output:1
 cookie=0x0, duration=122.743s, table=0, n_packets=287, n_bytes=47589, priority=
0 actions=CONTROLLER:65535
 cookie=0x0, duration=122.743s, table=1, n_packets=0, n_bytes=0, priority=0 acti
ons=CONTROLLER:65535
```

Figure 3.12: Flows in switch s2left

3.2. Example 2

With this example it is intended to show the capabilities of group tables to overcome sudden port failures in the network. These groups allow link failure detection and recovery to be done entirely in the data plane, without consulting the controller. This illustration will be made with an application that uses group tables with a Fast-Failover (FF) group to define alternative ports in case of failure. This port definition occurs at a data plane level, without the need to communicate

with the controller, meaning it will be faster than having to wait for the controller's answer.

As mentioned before FF groups are designed to detect and overcome port failure. It has a list of buckets like the other group types but in each bucket it also has another parameter: a watch port or watch group. This parameter defines the port or group that is monitored and analyses the status of the port used by the bucket. Only one bucket can be used at a time and will be used for every packet, unless the liveness status of the port or group transitions from up to down. If this event occurs that bucket will not be used anymore and the FF group will quickly select the next bucket in the bucket list with a watch port/group that is up.

Although the search for the next bucket with a live watch port / group could take some time, it is almost guaranteed it will be faster than the alternative of sending a message for the controller and wait for the reply with a new forwarding rule.

Figure 3.13 shows the topology used in this example. There are four OpenFlow switches between two hosts. Within these four switches is possible to outline two possible paths connecting the two hosts: A and B. The FF groups will be used on Switch 1 and Switch 3. The Switch 1 and Switch 3's FF groups will have two buckets. The first bucket will have a watch port, evaluating the liveness of the port connected to Switch 2a and the second bucket will have a watch port evaluating the liveness of the port connected to Switch 2b. Depending on the link status of the links connecting Switch 2a and Switch 2b to Switch 1 and Switch 3, one path should choose between the Host 1 and Host 2.

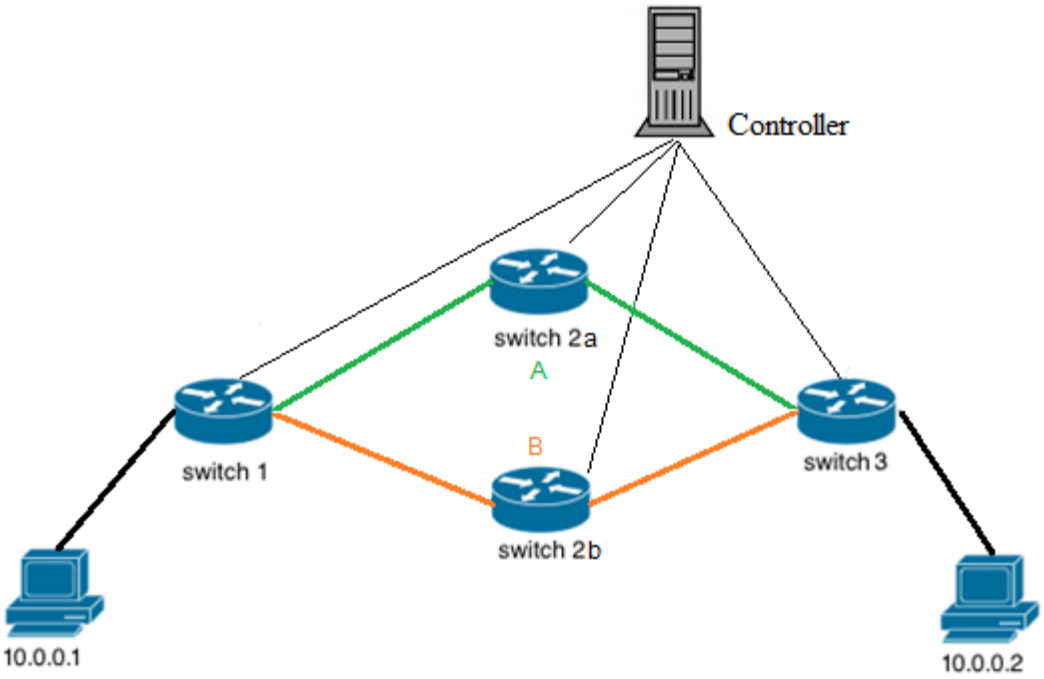

Figure 3.13: Topology used in Example 1

The python script to create this network can be found in [25].

In order to test this example, we will execute the application on the network shown in Figure 3.13 and set a link liveness to down, forcing Switch 1 and Switch 3 to choose another path. The recovering time and efficiency will be studied by measuring the effective transmission throughput and the number of packet loss.

3.2.1. Application Development

In this step we will use Fast Failover groups in Switch 1 and Switch 3. The application is responsible to send the flows and group modification to the switches in our topology.

It will be used a REST API to emulate port up/down events to cause the installed groups to change to path A or path B. Therefore, a handler for the REST API is included to listen for commands.

We will develop an interface which exposes our module as a service to other Floodlight modules. When a REST command is received, our module's handler can be invoked as a service by the REST API module. The code for this interface is very simple, it simply calls the methods implemented in our application. The first method is *handleToggleRequest*, which will change the path used by setting ports along path A up and path B down or the opposite, depending which path is being used. The second method is *handleResetRequest* that will set all ports up.

For this example groups will be installed on Switch 1 and Switch 3. These groups will contain two buckets, representing the two possible paths. The bucket for path A has the switch port number that reaches Switch 2a as its output and will check the liveness of this port. The bucket for path B has the switch port number that reaches Switch 2b and will check the liveness of this port.

When a switch establishes a connection to a controller, the controller sends a feature request message to the switch and waits for a reply. When the reply arrives, the controller gets informed about the features provided by the switch, for instance, the datapath ID (i.e., DPID), list of ports, etc. After the connection is made, the controller periodically sends a Packet_OUT to the switches to flood LLDP packets to its neighbors. When these packets are received, every switch sends a Packet_IN message to the controller with the LLDP Ethernet frame structure received. If two switches send the same packet, it means they have a link between them and the controller learns that link. Having the source switch, source port, destination switch and destination port, a Link object is created. The Link class can provide us the port numbers to use in group's buckets as watch ports. These ports can be extracted from the Link variables *link_dpid1_to_dpid2a* and *link_dpid1_to_dpid2b* using the methods exposed by Link class:

OFPort theSrcPort = someLink.getSrcPort(); // get source port of link

OFPort theDstPort = someLink.getDstPort(); // get destination port of link

We can use the object *link_dpid1_to_dpid2a* to obtain the port on Switch 1, by asking for the source port, and the in port on Switch 2a, by asking for the destination port. The two switches ports leaving Switch 1 to Switch 2a and Switch 2b will be the output action of the two buckets defined in our group and, because its liveness will determine the path chosen, the watch ports' parameter.

Next, we will compose the required buckets for our group used in Switch 1.

```
      ArrayList<OFBucket>
buckets= new ArrayList<OFBucket>(2);
      buckets.add(sw1.getOFFactory().buildBucket()
          .setWatchPort(link_dpid1_to_dpid2a.getSrcPort())
          .setWatchGroup(OFGroup.ZERO)
          .setActions(Collections.singletonList((OFAction)
sw1.getOFFactory().actions().buildOutput()
              .setMaxLen(0xffFFffFF)
              .setPort(link_dpid1_to_dpid2a.getSrcPort())
              .build()))
          .build());
      buckets.add(sw1.getOFFactory().buildBucket()
          .setWatchPort(link_dpid1_to_dpid2b.getSrcPort())
          .setWatchGroup(OFGroup.ZERO)
          .setActions(Collections.singletonList((OFAction)
sw1.getOFFactory().actions().buildOutput()
              .setMaxLen(0xffFFffFF)
              .setPort(link_dpid1_to_dpid2b.getSrcPort())
              .build()))
          .build());
```

This bucket's array is used when we create the group and send it to Switch 1.

```
OFGroupAdd groupAdd = sw1.getOFFactory().buildGroupAdd()
  .setGroup(OFGroup.of(1))
  .setGroupType(OFGroupType.FF)
  .setBuckets(buckets)
  .build();
sw1.write(groupAdd);
```

The final step is to compose the FlowMod messages that will instruct the switch to send the packets coming from Host 1 to our group. In order to permit ARP and IPv4 packets, we need to add two new flows to Switch 1 and Switch 3, one for each packet type. These two switches have two ports, one leading to Switch 2a and to Switch 2b and another leading to the hosts. The Match needed in the *OFFlowAdd* needs to contemplate the packets which the incoming port is the one connected to the hosts, and we can determine that port using the *getHostPort* function, that takes an OFSwitch as an argument and returns an *OFPort* leading to a host. The flow for ARP packets is shown below:

```
OFFlowAdd flowAdd = sw1.getOFFactory().buildFlowAdd()
  .setCookie(cookie)
  .setHardTimeout(0)
  .setIdleTimeout(0)
  .setPriority(FlowModUtils.PRIORITY_MAX)
  .setMatch(sw1.getOFFactory().buildMatch()
  .setExact(MatchField.ETH_TYPE, EthType.ARP)
  .setExact(MatchField.IN_PORT, getHostPort(sw1))
  .build())
```

```
    .setActions(Collections.singletonList((OFAction)
  sw1.getOFFactory().actions().buildGroup()
    .setGroup(OFGroup.of(1))
    .build()))
    .build();
  sw1.write(flowAdd);
```

The flow for IPv4 packets is similar because only the *Eth Type* changes, therefore the ARP *OFFlowAdd* flowAdd code can be reused as follows:

```
  flowAdd = flowAdd.createBuilder()
    .setMatch(sw1.getOFFactory().buildMatch()
      .setExact(MatchField.ETH_TYPE, EthType.IPv4)
      .setExact(MatchField.IN_PORT, getHostPort(sw1))
      .build())
    .build();
  sw1.write(flowAdd);
```

3.2.2. Connect Mininet topology to Floodlight controller

With the controller started we can start the topology for example 2. Since the learning switch is disabled in the controller, the switches in the network do not have flows saved and the paths are not known yet, making it impossible to ping from one host to another. We need to trigger our module to create and insert the desired flows and groups, so we have to use the REST API mentioned in 3.2.1 to invoke the method *handleToggleRequest*, issuing the following command in our controller:

```
curl http://localhost:8080/wm/fast-failover-demo/toggle-path -X POST -d '' | python -
m json.tool
```

Now is possible to check the switch's configuration to see if the configurations are correct. Opening an Xterm for Switch 1 or Switch 3 (“xterm s1” on the terminal) it is possible to see the flows and groups inserted in the switch by our controller. On the Xterm window we can write:

sudo ovs-ofctl dump-flows s1 -O OpenFlow13

And we see the two flows inserted which direct all IPv4 and ARP packets to group 1:

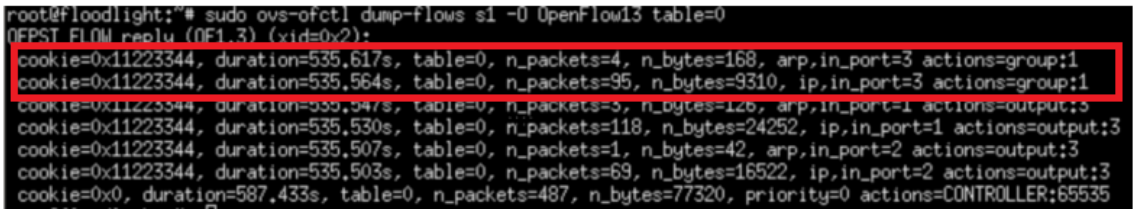

Figure 3.14: Flows in Switch S1

This group 1 can be examined issuing the following command:

sudo ovs-ofctl dump-groups s1 -O OpenFlow13

Figure 3.15: Groups in Switch S1

And it is clear that it is an FF group, that have two buckets whose actions are output to port 1 or output via port 2 (links for switch 2a and switch 2b). The decision concerning where to send the packets depends on the watch ports values.

This is the setup of both Switch 1 and Switch 3 and only one port will be active at a time. To watch the FF group in action, a link failure must be simulated. To do it, the REST API should be called, the link between Switch 1 and Switch 2a is set to down and the path between Host 1 and Host 2 is damaged, so another path must be found. After sending some ARP packets finding the new path, Switch 2b tells Switch 1 that there is a path through it to Host 2 and the packets start being sent through Switch 2b. The reaction time can be seen in the image below, where it is possible to see two packets that took longer to arrive at their destination.

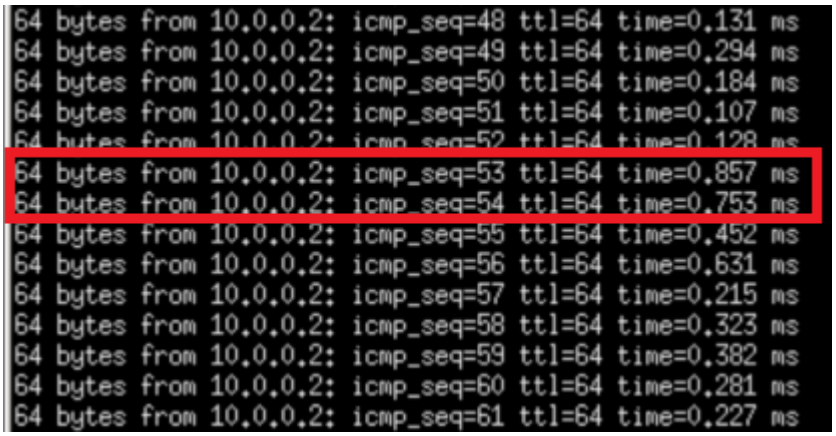

Figure 3.16: Demonstration of a link failure reaction

This example shows how fast a network can react to a link failure just by using groups, without having to communicate with the switch, making it a good solution in preventing this events.

4

4. Conclusion

4.1. Conclusions

This thesis has demonstrated how to use SDN, when using the floodlight controller and Mininet. Other softwares can provide the same result however require more complex setups. With Mininet, the programmer can easily implement a virtual network with a custom topology. The floodlight's modules developed in this thesis were used as southbound APIs to guarantee the communication between the controller and the switches. This enables the possibility of having dynamic behaviors on the same network. A module in floodlight determines the behavior of the controller by having a packet processing method in a familiar, general-purpose programming language. This packet processing method will be called when a switch sends a Packet_IN message to the controller, after the switch itself receives a packet with an unknown destination. This method is conceptually applied to every packet with an unknown destination entering a switch. The ability to add more modules opens the door to a number of extensions and refinements. However, the development of these modules is not an easy task since it is a complex and extensive code and requires the developer to understand how a switch works at a low level because the instructions to the switch contemplate, for example, the switch ports. Even for simple modules, several methods need to be developed

for the module to work, making this approach time-saving for big networks, but complicated to simple exercises.

The cost of the communication between the switches and the controller was evaluated with three different steps. The premise of the test was a switch receiving a packet whose destination was unknown to the switch, and it would send a message to the controller asking for what to do. The three options were telling the switch to flood the packet in order to find the path, telling the switch which out port to send the packet and telling the switch to send every packet with that destination to a specific out port. The last option was proven the fastest because the communication between the switch and the controller happens only once.

The second example showed how to use groups to enhance link failure recovery. With groups, no communication between switch and controller is needed, making it the fastest option.

4.2. Future work

Several aspects were not mentioned or detailed in this thesis and would be interesting to develop in future work.

- Pipeline processing

  With OpenFlow 1.1 came the possibility for a switch to have several flow tables. Pipeline processing always starts at the first flow table and the packet is first matched against its flow entries. Other flow tables may be used depending on the outcome of the match in the first table. If a flow entry is found, the instruction set included in that flow entry is executed, those instructions may explicitly direct the packet to another flow table (using the Goto

Instruction), where the same process is repeated again.

- Queues

Queues are designed to provide a guarantee on the rate of flow of packets placed in the queue. As such, different queues at different rates can be used to prioritize "special" traffic over "ordinary" traffic.

- Flow Meters

With Flow Meters we can monitor the ingress rate of traffic as defined by a flow. Flows can direct packets to a meter using the goto-meter OpenFlow instruction, where the meter can then perform some operation based on the rate it receives packets. Each meter may have one or more meter bands. Each band specifies the rate at which the band applies and the way packets should be processed. Packets are processed by a single meter band based on the current measured meter rate, the meter applies the meter band with the highest configured rate that is lower than the current measured rate. If the current rate is lower than any specified meter band rate, no meter band is applied.

- Collect Switch Statistics

OpenFlow has many statistics messages to allow the controller to query the switch for information about its running state. Examples include flow stats, meter stats, queue stats, aggregate stats, table stats, and port stats.

References

[1] D. Kreutz et al. “Software-defined networking: A comprehensive survey”. InPro-ceedings of the IEEE, volume 103, no. 1, Jan 2015.

[2] D. Kreutz, F. Ramos, and P. Verissimo. “Towards Secure and Dependable Software-defined Networks”. InProceedings of the Second ACM SIGCOMM Workshop onHot Topics in Software-Defined Networking. HotSDN’13. Hong Kong,China: ACM, 2013.

[3] S. Tanenbaum and D. J. Wetherall. Computer Networks. Pearson, Boston, 2011.

[4] J. KuroseandK.Ross. ComputerNetworking,- ATop-DownAproach. Addison-Wesley, 2010.

[5] Nick Feamster, Jennifer Rexford, and Ellen Zegura.The Road to SDN: An Intellectual History of Programmable Networks. 2013.

[6] N. Feamster, J. Rexford, and E. Zegura. “The road to SDN”. Queue, 11 (12):20:20– 20:40, 2013.

[7] MCeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). Op enFlow: enabling innovation in campusnetworks.ACM SIGCOMM Computer Communication Review, 38 (2):69- 70.

[8] Roy T. Fielding and Richard N. Taylor. “Principled Design of the Modern Web Architecture”. Transactions on Internet Technology, 2 (2): 115-150.

[9] F. A. Botelho, F. M. V. Ramos, D. Kreutz, and A. N. Bessani. “On the Feasibility of a Consistent and Fault-Tolerant Data Store for sdns,”. In Proceedings of the 2013 Second European Workshop on Software-Defined Network. Washington, DC, USA: IEEE Computer Society, 2013.

[10] Kristian Slavov, Makan Pourzandi and Daniel Migault. “Identifying and addressing the vulnerabilities and security issues of SDN”.

[11] M. Antikainen, T. Aura, and M. S “Spook in Your Network: Attacking an SDN with a Compromised OpenFlow Switch”. In K. Bernsmed and S. Fischer-Hubner, Springer International Publishing, 2014.

[12] S. Shin, P. Porras, V. Yegneswaran, M Fong, G. Gu, and M. Tyson. “Fresco:Modular Composable Security Services for Software-Defined Networks”. In Internet Society NDSS, 2013.

[13] Dorgival G; Luiz, V, Marcos, V, Henrique, R, and Rogério, N. “Redes Definidas por Software: uma abordagem sistêmica para o desenvolvimento das pesquisas em Redes de Computadores” Minicursos do Simpósio Brasileiro de Redes de Computadores-SBRC, 2012.

[14] Voellmy, A. Programmable and Scalable Software-Defined Networking Controllers. Degree of Doctor of Philosophy. Faculty of the Graduate School of Yale University, 2012

[15] Muntaner, G. (2012). Evaluation of OpenFlow Controllers.

[16] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker e J. Turner. “OpenFlow: enabling innovation in campus networks”. ACM SIGCOMM Computer Communication Review, 2008

[17] Mininet Overview, http://Mininet.org/overview/

[18] Floodlight, http://www.projectfloodlight.org/floodlight/

[19] The Controller, https://floodlight.atlassian.net/wiki/display/floodlightcontroller/the+controller

[20] Example 1. https://github.com/jorgecorreia26/Floodlight-Module-Skeleton/blob/master/Example1%20Python.

[21] New Class. https://github.com/jorgecorreia26/Floodlight-Module-Skeleton/blob/master/Skeleton%20of%20a%20New%20Class

[22] Example 1 - Step A. https://github.com/jorgecorreia26/Floodlight-Module-Tutorial/blob/master/Example%201%20Scenario%20A

[23] Example 1 - Step B. https://github.com/jorgecorreia26/Floodlight-Module-Tutorial/blob/master/Example%201%20Scenario%20B

[24] Example 1 - Step C. https://github.com/jorgecorreia26/Floodlight-Module-Tutorial/blob/master/Example%201%20Scenario%20C

[25] Example 2. https://github.com/jorgecorreia26/Floodlight-Module-Tutorial/blob/master/Example%202%20Python

[27] Yun, Ma. "Bottlenecks in Traditional Networking", URL: http://www1.huawei.com/enapp/2679/hw-314358.htm