MASTER THESIS

UNIVERSITARY MASTER IN TELECOMMUNICATIONS ENGINEERING

Design of a flow monitoring solution for OpenFlow Software-Defined Networks

Author
José Rafael Suárez-Varela Maciá

Directors
Jorge Navarro Ortiz - Universidad de Granada
Pere Barlet Ros - Universitat Politècnica de Catalunya

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, September 2017

Design of a flow monitoring solution for OpenFlow Software-Defined Networks

Author
José Rafael Suárez-Varela Maciá

Directors
Jorge Navarro Ortiz - Universidad de Granada
Pere Barlet Ros - Universitat Politècnica de Catalunya

Design of a flow monitoring system for OpenFlow Software-Defined networks

José Rafael Suárez-Varela Maciá

Keywords: Software-Defined Networks, OpenFlow, Network flow-level monitoring

Abstract

Obtaining flow-level measurements, similar to those provided by Netflow/IPFIX, with OpenFlow is challenging as it requires the installation of an entry per flow in the flow tables. This approach does not scale well with the number of concurrent flows in the traffic as the number of entries in the flow tables is limited and small. Flow monitoring rules may also interfere with forwarding or other rules already present in the switches, which are often defined at different granularities than the flow level. In this project, we present a transparent and scalable flow-based monitoring solution that is fully compatible with current off-the-shelf OpenFlow switches. As in NetFlow/IPFIX, we aggregate packets into flows directly in the switches and asynchronously send traffic reports to an external collector. For the sake of scalability, we propose two different traffic sampling methods depending on the OpenFlow features available in the switch. We developed our complete flow monitoring solution within OpenDaylight controller and evaluated its accuracy in a testbed with Open vSwitch. Our experimental results using real-world traffic traces show that the proposed sampling methods are accurate and can effectively reduce the resource requirements of flow monitoring in OpenFlow environments.

Diseño de un sistema de monitorización de flujos para redes definidas por software con OpenFlow

José Rafael Suárez-Varela Maciá

Keywords: Redes definidas por software, OpenFlow, Monitorización de redes a nivel de flujo

Resumen

Obtener medidas en SDN a nivel de flujo, similares a las que se obtienen con NetFlow/IPFIX, utilizando OpenFlow no es una tarea trivial en tanto que sería necesario instalar una entrada por cada flujo en las tablas de los switches. Debido a que el número de entradas en los switches es bastante limitado, esta propuesta no es escalable dado el elevado número de flujos concurrentes que a menudo se procesan en un switch. Las reglas de monitorización de flujos pueden además interferir con otro tipo de reglas (ej. forwarding) que típicamente se definen a otros niveles de granularidad diferentes al nivel de flujo. En este proyecto se presenta un sistema de monitorización de flujos de tráfico transparente y escalable, totalmente compatible con los switches OpenFlow que se pueden encontrar actualmente en el mercado. Como ocurre en NetFlow/IPFIX, este sistema agrega las estadísticas de los paquetes en registros de flujo y envia asíncronamente a un colector informes con resúmenes de estas estadísticas. Para sobrevenir los posibles problemas de escalabilidad, proponemos dos métodos de muestreo de flujos dependiendo de las características opcionales de OpenFlow disponibles en el switch sobre el que se desea implementar. Hemos implementado nuestra solución de monittorización en el controlador OpenDaylight y se ha evaluado su precisión en un entorno de red con Open vSwitch. Nuestros resultados experimentales muestran que los métodos de muestreo propuestos son precisos y permiten reducir de forma efectiva los recursos necesarios para llevar a cabo monitorización a nivel de flujo en entornos OpenFlow.

Yo, **José Rafael Suárez-Varela Maciá**, alumno de la titulación TITULACIÓN de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI XXX, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Master en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: José Rafael Suárez-Varela Maciá

Granada a 3 de Septiembre de 2017.

D. **Jorge Navarro Ortiz**, Profesor del Área de Ingeniería Telemática del Departamento Teoría de la Señal, Telemática y Comunicaciones (TSTC) de la Universidad de Granada.

D. **Pere Barlet Ros**, Profesor del Departamento de Arquitectura de Computadores (DAC) de la Universitat Politècnica de Catalunya (UPC BarcelonaTech)

Informan:

Que el presente trabajo, titulado ***Design of a flow monitoring solution for OpenFlow Software-Defined networks***, ha sido realizado bajo su supervisión por **José Rafael Suárez-Varela Maciá**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 3 de Septiembre de 2017.

Los directores:

Jorge Navarro Ortiz **Pere Barlet Ros**

Acknowledgement

Ha sido un largo camino hasta llegar a este punto.

Quiero aprovechar estas líneas para agradecer a todas las personas que me han acompañado, en los buenos y en los malos momentos, a lo largo de mi andadura universitaria y cuyo desenlace tiene lugar con este Trabajo de Fin de Máster.

A mis padres, a mi hermana y a Joaquina por su apoyo incondicional, por sus consejos, por su cariño, por sus valores y por ser todos ellos grandes ejemplos de perseverancia y animarme a seguir hacia delante.

A mis amigos, a los de toda la vida y a todos aquellos que he tenido el placer de conocer durante la carrera y el máster. Porque sin vosotros estos años no tendrían el significado que han tenido para mí. Gracias por vuestro apoyo y por los buenos momentos que hemos compartido y que seguiremos compartiendo.

Y, por supuesto, a mis tutores Jorge y Pere. Porque sin ellos no habría sido posible la realización de este trabajo. Me gustaría destacar su apoyo, porque siempre han estado ahí cuando lo he necesitado. Además siempre me han proporcionado buenos consejos y se han preocupado por que se hicieran las cosas de la mejor forma posible.

Contents

1 Introduction **1**
- 1.1 The Software-Defined Networking paradigm . . . 1
- 1.2 Motivations . . . 4
  - 1.2.1 Monitoring and measurements in SDN . . . 5
  - 1.2.2 OpenFlow . . . 6
  - 1.2.3 The OpenDaylight controller . . . 7
- 1.3 Main contributions . . . 8
- 1.4 Organization of this report . . . 10

2 State of the Art **13**
- 2.1 Software-Defined Networking . . . 13
- 2.2 Traffic monitoring for SDN . . . 14
  - 2.2.1 OpenFlow-based solutions . . . 14
  - 2.2.2 Alternative solutions to OpenFlow . . . 15
- 2.3 Concluding remarks about the state of the art . . . 16

3 Analysis of objectives and specification of requirements **17**
- 3.1 Objectives . . . 17
- 3.2 Specification of requirements . . . 18
- 3.3 Assessment of the achievement of the proposed objectives . . . 20

4 Planning and estimated costs **23**
- 4.1 Planning . . . 23
- 4.2 Resources used . . . 26
  - 4.2.1 Human resources . . . 26
  - 4.2.2 Hardware resources . . . 26
  - 4.2.3 Software resources . . . 27
- 4.3 Estimated costs . . . 28
- 4.4 Final budget . . . 29

5 Design and evaluation of the solution **31**
- 5.1 OpenFlow background . . . 31
  - 5.1.1 Multiple flow tables and groups . . . 32
  - 5.1.2 Adding new flow entries and groups . . . 33

5.1.3 Statistics collection 33
5.2 Monitoring system 34
5.2.1 Proposed sampling methods 35
5.2.2 Modularization of the system 38
5.2.3 Statistics retrieval 38
5.3 Experimental evaluation 38
5.3.1 Accuracy of the proposed sampling methods 39
5.3.2 Evaluation of the overhead 43
5.3.3 Validation of the sampling method based on IP suffixes 49

6 Conclusions and future work 51
6.1 Conclusions 51
6.2 Future work 52

Appendices 55

A Installation manual 57

B User manual 61

C Conference paper 65

References 80

List of Figures

1.1 Software-Defined Networking architecture. 2
1.2 Components of a flow entry in OpenFlow. 6
1.3 Architecture of the OpenDaylight beryllium release. 8

4.1 Gantt diagram planned for the project. 26
4.2 Graphic with the estimated costs for the human resources. . . 29

5.1 Scheme of OpenFlow tables and entries of the monitoring system. 35
5.2 Sampling based on hash function 37
5.3 Evaluation of sampling rate for methods based on source IP suffixes. 40
5.4 Evaluation of sampling rate for methods based on pairs of IP suffixes. 40
5.5 Evaluation of sampling rate for the hash-based method. . . . 41
5.6 Weighted Mean Relative Difference (WMRD) between FSDs. 42
5.7 Average number of redundant packets per flow. 44
5.8 Percentage of redundant bytes. 45
5.9 Evaluation of the method based on source IP suffixes with randomized traces 49
5.10 Evaluation of the method based on pairs of IP suffixes with randomized traces 49
5.11 Weighted Mean Relative Difference (WMRD) between FSDs with randomized traces 50

A.1 Output when the application is successfully compiled. 59
A.2 Output of Open vSwitch with a small scenario with a SDN controller and a host. 60

B.1 Execution of the OpenDaylight controller. 61
B.2 Example of the initial flow entries in our scenario. 62
B.3 Example of a group table in our scenario. 63

List of Tables

4.1 Temporal estimation of the project. 25
4.2 Estimated costs for the human resources. 28
4.3 Estimated costs for hardware resources. 29
4.4 Final budget estimated. 30

5.1 Summary of the real-world traffic traces used. 39
5.2 Estimation of the average flow entries used in the switch. . . 48

Chapter 1

Introduction

This chapter provides an introduction including the main topics and technologies related to the project developed for this master thesis. Firstly, we provide a section describing the scope of the topic. There, we briefly discuss the origin of the emerging *Software-Defined Networking* (SDN) paradigm and describe some details to better understand how these new networks operate. In particular, we focus on traffic monitoring and the new challenges to address in SDN.

Then, we provide a section motivating the main issues around the project and some technologies we used in this master thesis. This section is followed by another section that describes the main contributions of the proposal presented in this thesis.

Finally, we outline the structure of this report including a description of the chapters and appendices included in this thesis.

1.1 The Software-Defined Networking paradigm

Current IP networks are increasingly more complex and hard to manage, and this is a tendency that it is going to be more accused with new emerging paradigms of services such as virtualized cloud computing, big data applications, data center services or multimedia content delivery. With the aim of reverse this situation, the Software-Defined Networking (SDN) paradigm was proposed as a solution to build a more flexible network infrastructure with programmable devices, where new protocols and policies can be implemented via software without needing any hardware modification.

The SDN paradigm proposes to separate the control and data planes of "legacy" networks for the sake of flexibility. In this way, the data plane is located in the SDN-enabled forwarding devices (i.e., SDN switches), while

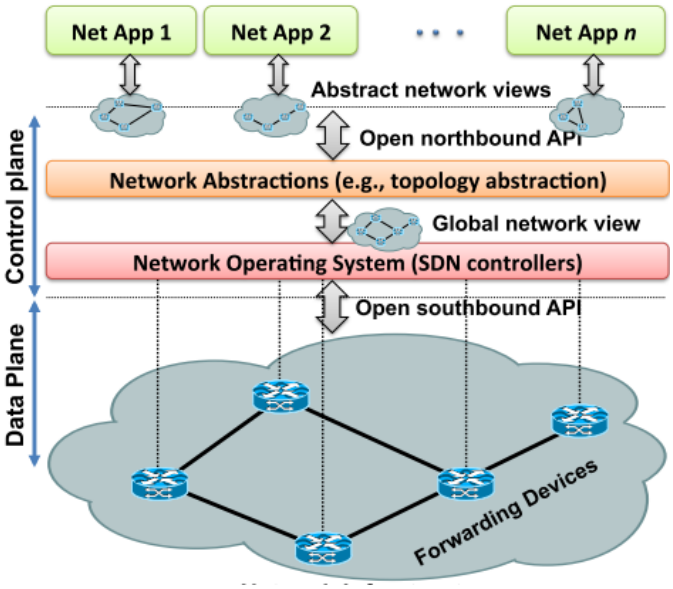


Figure 1.1: Software-Defined Networking architecture.

Source: Kreutz, D et al. "Software-defined networking: A comprehensive survey", Proceedings of the IEEE.

the control plane is logically centralized in new entities called SDN controllers. Thus, the different planes in SDN allow to create different layers of abstraction and, thereby, providing an unprecedented level of flexibility in network management. Fig. 1.1 (taken from [1]) depicts the architecture of an SDN-based network. There, we can see the different entities, planes and layers of abstraction in SDN. We briefly describe the main elements below.

Planes in Software-Defined Networks

- **Control plane:** This plane is in charge of calculating the local state of the forwarding devices in the network and enforcing the proper policies for the correct operation of the network. This includes all the network management tasks such as routing, traffic engineering or security policy enforcement. Unlike traditional networks, in SDN forwarding decisions are flow-based, instead of destination-based. All the packets matching a specific criterion are applied the same actions. This flow abstraction enables to unify the behavior of different types of network devices (e.g., switches, routers, firewalls). In a nutshell, the control plane makes decisions about how and where the traffic is sent and manages all the signalling of the network to properly configure the network devices.

- **Data plane:** This plane performs traffic forwarding in the network devices (i.e., the switches) according to the rules defined by the control plane. This also includes traffic filtering and the different actions that can be typically executed for new incoming packets in a switch. It is worth noting that switches in SDN are of general purpose, i.e., they execute the flow-level rules installed by the control plane and can combine actions that used to correspond to different types of network devices (e.g., routers, switches, middleboxes) in traditional networks. This plane is also known as the "forwarding plane".

Abstractions in Software-Defined Networks

Following the definition in [1], we distinguish the three abstractions described below:

- **Forwarding abstraction:** This abstraction allows the data plane to ideally perform any forwarding action determined by the control plane while hiding the underlying hardware in the network infrastructure. Currently, OpenFlow [2] is the dominant protocol of SDN that implements this abstraction. This protocol provides a standard API (Southbound API) for the communication between the control and the data planes, i.e., between the SDN controllers and the OpenFlow-enabled switches.

- **Distribution abstraction:** This abstraction enables SDN applications to operate without the necessity of being aware of the distributed state of the network. In contrast to traditional networks, the status information in SDN is logically centralized and it enables to make much better decisions with a global view of the network in the controllers. This is possible thanks to the *"Network Operating Systems"* (NOS) of SDN, which collect the status information of the network and it is in charge of installing the desired rules in the switches through a standard communication protocol (e.g., OpenFlow).

- **Specification abstraction:** This abstraction allows to describe the desired operation of the network by means of high-level policies defined by network management applications. Thus, these policies are then mapped into sets of physical configurations for the global network view exposed by the SDN controller. This abstraction provides an interface (Northbound API) which ideally allows network applications to operate over simplified abstract models of the network which are oblivious of the underlying network topology.

The design of the SDN paradigm enables to perform a fine-grained management of the network, taking advantage of the decision making from the global perspective of the network in the controller. However, to be successful in current dynamic environments, traffic monitoring becomes a cornerstone in SDN given that management applications often need to make use of accurate and timely traffic measurements. Additionally, an inherent issue of SDN is its scalability. For a proper design of a monitoring system, it is necessary to consider the network and processing overhead to store and gather the flow statistics. On the one hand, note that controllers are critical points in the infrastructure since all the management decisions are made and communicated from there to each switch under its control. On the other hand, the most straightforward way of implementing per flow monitoring is by maintaining an entry for each flow in a table of the switch. Each of these entries has some counters which are updated every time a packet matches them. Thus, obtaining accurate measurements of all flows results in a great constraint, since nowadays OpenFlow commodity switches do not support a large number of flow entries due to their limited hardware resources available [3].

This master thesis covers a study of traffic monitoring in SDN-based networks. More specifically, it aims to identify all the issues around traffic measurement derived from the SDN paradigm and proposes a scalable and OpenFlow compliant monitoring system more suitable for those networks of the future. The approach of this solution is to obtain flow-level measurement reports equivalent to those of NetFlow/IPFIX [4] in traditional networks. Likewise, we evaluated our system to analyze its feasibility and to quantify its accuracy, resource requirements and network overhead.

1.2 Motivations

The SDN paradigm was the outcome of some works of the research world that suggested the necessity of building programmable networks as a practical way to experiment with new protocols in production networks. From its inception, it has gained lots of attention from academia and industry. It is supported by giants of the Internet world like Google, Cisco, HP, Juniper or NEC and by standardization organizations like the Open Network Foundation (ONF) or the Internet Engineering Task Force (IETF), so one can state that this network paradigm has a lot of potential to succeed. The proposal of the OpenFlow protocol [2] in 2008 was its major driver. In that text, they talk about the commonly held belief that the network infrastructure was “ossified”. Thus, they proposed OpenFlow as a protocol for the communication between the forwarding and control planes in order to decouple logically and physically these two planes.

SDN introduces the benefits of a centralized approach to network configuration. That way, each time network administrators want to make a policy change do not have to configure individually by-hand each network device using its own vendor-specific code. Unlike traditional networks with distributed management, in SDN the administrator can apply some new high-level policies from the controller and this is the responsible for translating the policies into rules and install them in the forwarding devices involved. This allows software developers to be oblivious of the underlying devices and develop their networking logic the same way they do in computer software.

Furthermore, SDN-based networks offer a level of flexibility never seen before. It allows to perform a fine-grained flow-level management ideal for services with strict QoS requirements. It successfully adapts to current environments with high fluctuate traffic to make an efficient use of the network resources. In SDN, the forwarding devices are of general purpose, i.e., they are not designed for a specific network function (router, firewall). This results in the possibility of re-configuring via software the topology and change the role of the different devices at run-time, taking advantage of the global knowledge of the network state in the controller. This is an arduous task in traditional networks, since forwarding devices are inflexible due to the underlying hardwired implementation of routing rules. Concerning the measurements and monitoring tasks, SDN allows to collect some traffic statistics that in traditional large networks in some cases it is unfeasible.

In [3], they envision that, for the moment, the majority of research efforts are focused on providing solutions and services over SDN-based networks, while network management is not given much attention. Likewise, they remark the importance to consider the network management before the technology is widely deployed and, therefore, network management arises as a real need. They state that "addressing SDN management is imperative to avoid patching SDN later".

1.2.1 Monitoring and measurements in SDN

The huge scale and diversity of today's Internet traffic make it difficult for the operators to measure and maintain the status and dynamics of the network in short timescales. This, in turn, has become a great challenge, since there are more and more services and applications with guaranteed QoS requirements to be maintained along end-to-end network paths. It motivates the necessity of ubiquitous accurate traffic measurement and monitoring mechanisms.

The SDN paradigm makes it easier to perform QoS measurements and enables to perform a fine-grained management as well as making an efficient use of the network resources. However, although the SDN paradigm solves

some classical problems of the traditional networks, it brings new challenges. The decoupling of the control and forwarding planes has some new implications that need to be identified and considered in order to devise new smart solutions. Among these issues, the introduction of a centralized control plane makes necessary to consider now a latency between the forwarding and control planes that did not exist in legacy networks. This latency depends on the delay due to the network connection as well as the availability of the SDN controller. Thus, the controller becomes a critical point in the infrastructure and it is prone to become a network bottleneck. In this way, it is of vital importance to find a tradeoff between the tasks where the controller is involved and those that may be devolved to the forwarding devices.

As a conclusion, we see that nowadays there are a number of proposals around measurement and monitoring in SDN with their respective advantages and drawbacks, but there is still much work to be done.

1.2.2 OpenFlow

Since its inception in 2008, OpenFlow [2] has become the *de facto* protocol for the Southbound interface (communication between control and data planes) in SDN. This makes this protocol the main enabler of the SDN paradigm, since it was the first proposal that allowed to completely decouple the control and data planes of a network, which is the basis of the SDN paradigm. This proposal also introduced the idea of performing flow-level management with the aim of creating a standard where the control plane could be oblivious of the underlying hardware of the forwarding devices in the network.

OpenFlow allows to dynamically define the forwarding state of the network from the SDN controller by installing in the switches sets of flow entries. These flow entries are stored in flow tables in the switches and determine their behavior. Fig. 1.2 illustrates the main components of a flow entry.

Figure 1.2: Components of a flow entry in OpenFlow.

We describe below these components:

- **Match fields:** This field defines a filter (packet headers, ingress port, metadata, and others) to specify the packets that will be processed by this flow entry.
- **Priority:** Defines a priority to determine the flow entry to be applied to a packet when some flow entries have an overlap.

- **Counters:** These are some records that maintain the number of packets and bytes processed by the flow entry. It also store the life time of the flow entry since it was installed in the switch.
- **Instructions:** Set of actions to be applied to the packets matching the flow entry.
- **Timeouts:** Defines the amount of time before the flow entry is expired by the switch. There are two types of timeouts:

  1. The hard timeout defines the maximum amount of time since the flow entry was installed by the SDN controller in the switch.

  2. The idle timeout defines the maximum time interval between two consecutive packets matching the flow entry.

  Both timeouts can be installed simultaneously to decide when the flow entry will be evicted from the switch.
- **Cookie:** Unique opaque value selected by the SDN controller to identify the flow entry. This allows the controller to filter specific flow entries when modifying or deleting flow entries.
- **Flags:** These fields define how the flow entries are managed in the switch. For instance, it is possible to define if the switch sends a flow removed message to the controller including the data of the counters when the flow entry expires.

1.2.3 The OpenDaylight controller

The OpenDaylight controller [5] is the result of an open-source project leaded by the Linux foundation which was announced in 2008. This project is strongly supported by both academia and industry and it was created with the aim of accelerating as much as possible the adoption of *Software-Defined Networking* (SDN) and *Network functions virtualization* (NFV) in future networks. This project has a vast support from big players of the current Internet world including companies such as Cisco, Ericsson, Red Hat, ZTE, NEC, AT&T, DELL, Fujitsu, Huawei, Intel, Juniper and many others [6].

OpenDaylight is a vendor-independent platform, which makes it much easier to be adopted in any SDN-based network using switches from different vendors with support for different protocols. Thus, it offers support for a wide range of protocols for the Southbound interface in SDN. It includes support for protocols such as OpenFlow, OF-Config, NETCONF, LISP, OVSDB, BGP or SNMP. This allows also to operate in network environments combining pure SDN devices (e.g., OpenFlow-based switches) with other network devices that support popular protocols already used in traditional networks (e.g., BGP, SNMP).

Probably, the key success of the rapid growth of OpenDaylight lies in its large support community. Since it is open-source, anyone can collaborate in the development of the product. Thus, the members of the community can either contribute in many different projects that are already in progress or propose new ones which are evaluated for their approval. Likewise, members in the community are very active answering questions, which eases even more the collaboration among developers. As a result, they are able to constantly release new versions including many novel features. Fig. 1.3 depicts all the different plugins that offers OpenDaylight in its *Beryllium* version, which is the release we used for the implementation of the monitoring system presented in this report. Typically, each of these plugins belong to different projects that are independently developed by different groups of developers following some common programming guidelines.

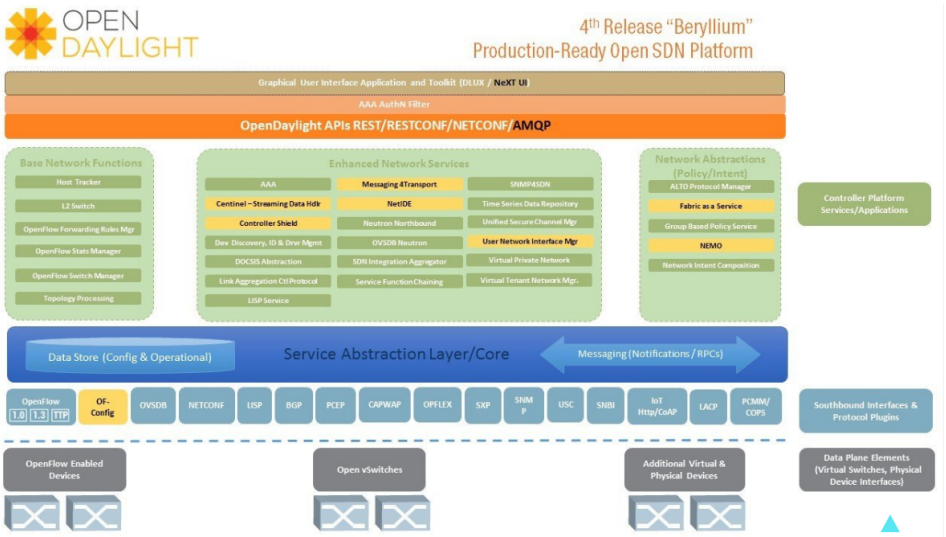


Figure 1.3: Architecture of the OpenDaylight beryllium release.

1.3 Main contributions

In this report, we present a monitoring solution which emulates NetFlow/IPFIX with OpenFlow and implements flow sampling. In this way, for each flow sampled, we maintain a flow entry in the switch. There, each flow entry records the duration (in seconds and nanoseconds) and packet and bytes counts. We use timeouts to define when these records are going to expire and, therefore, being reported to the controller. A similar approach was previously used in [7] to assess the accuracy of measurements and timeouts in some OpenFlow switches. However, their approach is not scalable as it

requires to install an entry in the flow tables for every single flow observed in the traffic, assumes that all rules have been deployed proactively for every flow that will be observed in the network, and does not address the problem of how monitoring rules interfere with the rest of rules installed in the switch (e.g., forwarding rules). In contrast, we present a complete flow monitoring solution that has the following features:

- **Scalable:** We address the scalabity issue in two different dimensions: (i) to alleviate the overhead for the controller and (ii) to reduce the number of entries required in the flow tables of the switches. To these ends, we designed two traffic sampling methods which depend on the OpenFlow features available in current off-the-shelf switches. We remark that our methods only require to initially install some rules in the switch which will operate autonomously to discriminate (pseudo) randomly the traffic to be sampled. To the best of our knowledge, there are no solutions in line with this approach. For example, iSTAMP [3] performs a flow-based sampling technique where they make use of a multi-armed-bandit algorithm to "stamp" the most informative flows and maintain particular entries to record per-flow metrics. However, this solution specifically addresses the detection of particular flows like *heavy hitters*, while our solution provides a generic dataset of the flows in the network. Likewise, iSTAMP needs to perform periodically a training phase. It means that it is not autonomous as our system.

- **Fully compliant with OpenFlow:** Our monitoring system implements flow sampling using only native features present since OpenFlow 1.1.0. This makes our proposal more pragmatic and realistic for current SDN deployments, which strongly rely on OpenFlow. Furthermore, for backwards compatibility, we also propose a less effective monitoring scheme that is compatible with OpenFlow 1.0.0, further increasing the targets that can benefit from our solution. Unlike NetFlow in traditional networks, OpenFlow also enables to independently monitor specific slices of the network, which can be highly interesting in emerging SDN/NFV scenarios. Additionally, we could check there are many SDN switches (e.g., some models of HP or NEC) which do not implement NetFlow, so our solution would be a good alternative for these devices, since it provides reports with flow-level statistics as in NetFlow. We found in the literature some monitoring proposals for SDN that rely on different protocols than OpenFlow. For instance, OpenSample [8] performs traffic sampling using sFlow [9], which is more commonly present than NetFlow in current SDN switches. However, we consider sFlow has a high resource consumption as it sends every sampled packet to an external collector and maintains there

the statistics. In contrast, our system maintains the statistics in the switch. Alternatively, some authors suggest to make use of different architectures specifically designed for monitoring tasks. For example, in [10], they propose using OpenSketch, where some sketches can be defined and dynamically loaded to perform different measurement tasks. However, in favor of our proposal, some works like [11] highlight the importance of making an OpenFlow compatible monitoring solution, as it is cheaper to implement and does not require standardization by a larger community. Note that despite the advances in the OpenFlow standard (version 1.5.1 at the time of this writing), the protocol does not provide direct support for flow sampling yet.

- **Transparent:** Our system can be interpreted as an additional module which does not affect the correct operation of other modules performing other network functions (e.g., forwarding). To ensure this, we make use of the pipeline processing feature with multiple tables of OpenFlow.

- **Asynchronous collection of flow statistics:** Our system collects and aggregates packets directly in the switch, and retrieves flow statistics when the flow expires (either by an idle or hard timeout). In FlowSense [12], they propose the same mechanism to retrieve statistics for the flow entries in the switches to estimate per-flow link utilization. The problem of their solution is that the statistics of flows with large timeouts are retrieved after too long. It makes obtaining accurate measurements unfeasible in environments with highly fluctuating traffic. In our solution, as our module is completely decoupled from others, we can define the most adequate timeouts to obtain accurate measurements. Our solution can also include mechanisms to conveniently select the timeouts, such as those proposed in PayLess [13] or OpenNetMon [11], where they design adaptive schedule algorithms to collect the statistics.

1.4 Organization of this report

With the aim of providing an overview of this thesis to the reader, we describe in this section the structure of the report. The present report is composed of the six chapters and three appendices described below.

Chapter 1: Introduction

In this chapter, we describe all the main aspects related to the project developed for this master thesis. It begins with an introduction to the SDN paradigm. Then, we provide a section with motivations around SDN and,

particularly, traffic monitoring in SDN-based networks. Finally, there is a section that provides a description of all the contributions achieved in this project.

Chapter 2: State-of-the-art

This chapter covers the most relevant research efforts around the topics addressed in this thesis. It includes some proposals in the literature that were part of the origins of the SDN paradigm as well as the main solutions around traffic monitoring in SDN. Particularly, we made a taxonomy to classify the most relevant monitoring solutions in different groups and describe their advantages and drawbacks.

Chapter 3: Analysis of objectives and specification of requirements

This chapter shows some points we considered before beginning the design and implementation of the monitoring solution presented in this report. We first identify the objectives and requirements to achieve a successful realization of the project. Finally, we assess the achievement of the objectives that were initially proposed once the realization of the project is finished.

Chapter 4: Planning and estimated costs

In this chapter, we describe the main aspects around the planning and costs estimation for the realization of our project. Firstly, we define a list with the different work packages that we propose for the achievement of the objectives of our project. Then, we provide an estimation of the time that we should spend for the elaboration of the project. Furthermore, we analyze the resources we will need. Lastly, we provide a final budget we estimated for the execution of the project.

Chapter 5: Design and evaluation of the solution

This chapter describes the design of the flow monitoring solution for *Software-Defined Networks* presented in this report and provides results about an extensive evaluation of the system. Firstly, it provides some theoretical background about the OpenFlow protocol necessary to understand the details of the design. Then, the architecture of the system is thoroughly explained. Lastly, it provides results from a large set of experiments that evaluate the accuracy and the overhead contribution of the system.

Chapter 6: Conclusions and future work

This chapter summarizes the main aspects about the project developed for this master thesis. It also includes some guidelines for future work to extend the monitoring system and the experiments we describe in this thesis.

Appendix A: Installation manual

This appendix describes the process to install the implementation we developed within the OpenDaylight controller as well as how to setup the testbed with Open vSwitch we used for our experiments.

Appendix B: User manual

This appendix shows how to use the monitoring system we implemented in OpenDaylight as well as how to perform simple experiments in a small testbed using Open vSwitch to test the application.

Appendix C: Conference paper

This appendix presents a conference paper which was presented in the *29th International Teletraffic Congress* (ITC) in 2017 as a result of the research developed for this master thesis. This paper describes the monitoring system for SDN presented in this report and provides an extensive evaluation of this system.

Chapter 2

State of the Art

This chapter provides an overview of the main efforts around the *Software-Defined Networking* (SDN) paradigm with a special focus on traffic monitoring. Firstly, we introduce the origins of SDN, and then, we describe the traffic monitoring proposals for SDN with greater impact on the research community and industry. The traffic monitoring proposals have been classified into two different groups: (i) OpenFlow-based solutions, and (ii) alternative solutions to OpenFlow.

2.1 Software-Defined Networking

The SDN paradigm has its roots in some works such as RCP [14], 4D [15] or ETHANE [16], which suggest the idea of separating the forwarding and control planes. However, many authors determine the date of its origin in 2008, with the proposal of the OpenFlow protocol [2], which has become as the *de facto* standard in SDN for the communication between the control and forwarding planes, i.e., between the SDN controllers and the OpenFlow-enabled switches. Strictly, the usage of the term "SDN" was coined in 2009 in an article [17] about the OpenFlow project. As for the SDN controllers, the concept of "network operating system" was first introduced with the NOX controller [18]. Currently, there is a wide range of controller proposals in the literature, such as [5], [19], [20], [21], [22], [23], [24], [25]. These controllers make use of different programming languages (e.g., Java, Python) and were created to address different issues related to SDN (e.g., scalability, performance, fault tolerance, fast prototyping). Likewise, there are some proposals of high-level configuration languages for network management applications. That way, for instance, in [26] the authors propose Procera, which is a configuration language compatible with OpenFlow that allows network operators to define high-level policies that are automatically

translated into a set of flow rules that enables to enforce the policy on the underlying network infrastructure.

2.2 Traffic monitoring for SDN

Concerning the monitoring and measurement tasks in SDN, we can see there is a number of efforts with different approaches that consider the SDN opportunities and implications. Here, we should take into account that an inherent issue of SDN is its scalability. For a proper design of a monitoring system, it is necessary to consider the network and processing overhead to store and gather the flow statistics. The most straightforward way of implementing per-flow monitoring is by maintaining an entry for each flow in a table of the switch. Each of these entries has some counters which are updated every time a packet matches them. Thus, obtaining fine-grained measurements of all flows results in a great constraint, since nowadays OpenFlow commodity switches do not support a large number of flow entries due to their limited hardware resources available (i.e., the number of TCAM entries and processing power) [3].

In the following subsections, we provide some of the most relevant proposals for traffic monitoring in SDN. These proposals were divided in two different classes that correspond to each of the two subsections: OpenFlow-based solutions and alternative solutions to OpenFlow.

2.2.1 OpenFlow-based solutions

Regarding the proposals that rely on the OpenFlow protocol, OpenTM [27] was one the first solutions from the research community for SDN network monitoring. In this work, the authors propose a system to calculate the traffic matrix and keep tracking of all active flows in the network. To this end, the system polls periodically the byte and packet counters from OpenFlow switches along the flow paths. This solution has severe scalability problems, since measuring a network-wide traffic matrix by periodically polling one switch on each flow path, can cause a significant overhead for the controller.

For the sake of scalability, a common practice in traditional networks is to implement traffic sampling when collecting flow measurements (e.g., NetFlow [4], JFlow [28] or sFlow [9]). In the SDN research field there are some solutions compatible with OpenFlow that aim to overcome the scalability issue following a similar approach. Thus, for instance, in iSTAMP [3] they propose a solution which performs traffic sampling. For this purpose, they present a technique where they make use of a multi-armed-bandit algorithm to “stamp” the most informative flows and maintain particular entries to

record per-flow metrics. This solution executes periodically a training phase with some iterations to detect those flows. It means that, for each training phase, it does not work well until the algorithm achieves a proper set of flows. Additionally, this solution specifically addresses the detection of particular flows like *heavy hitters* or specific flow sub-populations, but not allows to perform random flow sampling, which it is often interesting for some network tasks such as anomaly detection or traffic classification.

Likewise, in [7] they use the measurement features of OpenFlow to maintain per-flow statistics in the switches and assess the accuracy of the counters and timeouts of different devices. However, their approach is not scalable as it requires to install an entry in the flow tables for every single flow observed in the traffic, assumes that all rules have been deployed proactively for every flow that will be observed in the network, and does not address the problem of how monitoring rules interfere with the rest of rules installed in the switch (e.g., forwarding rules).

As for the traffic statistics retrieval in the controller, we found different approaches in the literature. For example, in FlowSense [12], they propose a passive push-based scheme to retrieve flow statistics when a flow expires (either by an idle or hard timeout). That way, they use this data to estimate per-flow link utilization. The main problem of this solution is that for flows with large timeouts, statistics are retrieved after too long a time. This makes obtaining accurate measurements unfeasible in current dynamic environments with highly fluctuating traffic. Other authors opt for active pull-based schemes and perform queries to retrieve the statistics. That is the case of PayLess [13] or OpenNetMon [11], where they design adaptive schedule algorithms to perform OpenFlow measurement queries in the switches. This kind of approaches has the limitation of the overhead that implies for the controller to perform a timely scheduled polling through all the switches under its control.

2.2.2 Alternative solutions to OpenFlow

Alternatively, other authors support the design of new SDN architectures that do not rely on OpenFlow. That is the case of [10], where they propose an architecture called OpenSketch which allows to define some sketches and load them dynamically to perform different measurement tasks. Other approach is to use measurement techniques already used in traditional networks. Thus, OpenSample [8] leverages sFlow [9] to perform packet sampling. However, this solution can have scalability issues as it sends every sampled packet to an external collector and maintains there the statistics.

In favor of OpenFlow-based solutions, it is important to remark that it is a vendor-independent technology with a strong support in the SDN

industry. This makes it highly prone to be adopted by all vendors and enable the interoperability among switches. Thus, in [11] the authors highlight the importance of making an OpenFlow compatible monitoring solution, since it is cheaper to implement and does not require standardization by a larger community.

2.3 Concluding remarks about the state of the art

Lately, it is noteworthy that there are some works in the literature which state that it is necessary to devolve some functions to the forwarding plane as a definitive solution for the inherent scalability issue of SDN. Thus, several solutions like DIFANE [29] or DevoFlow [30] were proposed with the aim of reducing the number of interactions between controllers and switches. They propose some ideas like enabling the switches to make some local routing decisions or learn about the topology changes in the network without involving the SDN controllers. Other works like [23], [21], [22], [31] are in line with the use of distributed controllers to enhance both the scalability and availability of the network. However, these solutions imply an additional overhead due to the necessary control communication between controllers to be properly coordinated. Moreover, they have to cope with the issue of consistently maintaining the information about the state of the network along all the controllers.

As a conclusion, we could observe there is no a definitive solution for network monitoring in SDN environments yet. There are some efforts in the literature to cope with the different issues around the SDN paradigm, but there is still a lot of work to do in order to achieve a mature solution proper for future SDN networks.

Chapter 3

Analysis of objectives and specification of requirements

The present chapter shows some points we considered before beginning the design and implementation of the monitoring solution for Software-Defined Networking presented in this report.

In the first section, we identify the major and minor objectives that are necessary to achieve a successful realization of the project. Then, we provide a detailed list of all the requirements to be accomplished at the end of this master thesis. These requirements are selected as a result of the objectives that are described in the previous section.

Finally, there is a section assessing the achievement of the objectives that were initially proposed once the realization of the project is finished.

3.1 Objectives

The final goal of this project is to design a practical monitoring system for Software-Defined Networks which provides reports with flow-level measurements as those of Netflow/IPFIX in traditional networks. To this end, we consider that a realistic solution for current SDN deployments has to be compliant with the OpenFlow protocol for the southbound API.

Furthermore, the proposed system should be scalable, which is a challenging objective to address since current OpenFlow-based networks have inherent issues of scalability. In this context, it is necessary to consider the network and processing overheads to store and collect the flow statistics. In particular, the system should alleviate as much as possible the processing overhead in SDN controllers and use a reduced amount of entries in the flow tables of the switches to maintain the flow statistics.

Lastly, we also consider that our solution should be easily deployable in any SDN-based network and operate transparently for other network modules. That is, not affecting the correct operation of other applications running in SDN controllers with different purposes (e.g., forwarding, security policy enforcement).

3.2 Specification of requirements

In this section, we define all the requirements that should have the monitoring solution presented in this report. This will enable us to identify the aspects that we should take into account during design process. The requirements were analyzed considering the objectives described in the previous section.

We list below the main requirements regarding the functionalities that should accomplish our monitoring solution:

- **Implementation within the OpenDaylight controller:**

  The monitoring system will be developed in the OpenDaylight controller [5], as it is one of the most popular SDN controllers nowadays and has a great support both from the reasearch community and the industry.

- **Full compliance with OpenFlow:**

  The implementation of the monitoring system should make use only of native features defined in the OpenFlow specification. Likewise, it should consider the features included in different versions commonly implemented in current off-the-shelf OpenFlow switches in order to maximize the targets that can benefit from our solution.

- **Flow sampling mechanisms:**

  The system should perform flow sampling in order to address the scalabity issue in OpenFlow-based networks. This enables to alleviate the overhead for the SDN controller as well as to control the number of entries required in the flow tables of the switches. Furthermore, we envision the design of different sampling methods with different levels of requirements of OpenFlow features available in the switches.

- **Autonomous operation:**

  The monitoring system should automatically detect when a new switch appears in the network topology and configure it to monitor the traffic in this device. To this end, the system will initially install some rules in the switch which will operate autonomously to discriminate the traffic to be sampled and maintain the statistics of those flows sampled.

- **Maintaining in the switch the flow statistics:**

  In order to reduce the processing overhead in the controller, the flow statistics should be maintained in the flow tables of the switches and then being reported to the SDN controller. In this way, our system should collect and aggregate packet statistics belonging to sampled flows directly in the switches.

- **Asynchronous collection of flow statistics:**

  The switches should report the flow statistics to the SDN controller when a monitored flow expires (either by an idle or hard timeout).

- **OpenFlow testbed:**

  We should build an OpenFlow-based testbed to deploy and evaluate our monitoring solution using real-world traffic. This testbed will be implemented using Open vSwitch [32].

- **Traffic capture and processing:**

  Additionally, we should develop some programs to capture traffic from a monitoring point and then process the traffic traces. This includes the injection of the traffic in our testbed in order to make experiments with our monitoring system. For this purpose, we will make use of the libpcap library [33] in C language, which is considerably efficient for traffic processing.

Furthermore, in order to achieve a successful design of our monitoring system, we should consider some additional aspects. Considering the specification of the requirements and according to the objectives defined previously, we provide below some guidelines that we should take into account during the design process as well as the development of our monitoring system in OpenDaylight:

- Analysis of the features available in current off-the-shelf OpenFlow devices. This includes some information such as the total amount of memory they have, the maximum number of flow entries and tables allowed or the support for multiple tables and group tables. This will allow us to design a practical and realistic solution easily deployable in nowadays SDN-based networks.

- Our system should be properly decoupled from other network modules running in the SDN controller to not affect the correct operation of other applications performing different network functions. This also enables us to specifically select the most adequate timeouts to obtain accurate flow measurements.

- The implementation should make use exclusively of open source software with extensive support in SDN environments. This makes our solution easier to be adopted in any network.
- The implementation of our system in OpenDaylight should be as efficient as possible, since SDN controllers are critical nodes in the network infrastructure and are prone to become a bottleneck in the network.
- The monitoring solution should cover the different versions of OpenFlow that are implemented in most of SDN switches. To this end, it is also important to consider the actual support in current SDN switches of those features which are defined as *optional* in the OpenFlow specification.
- To evaluate our system we should have access to real-world traffic. For our experiments, we envision to combine traffic traces accessible from different public repositories as well as traffic that we could capture with permission from vantage points in public networks.
- All the programs developed to capture and process the traffic traces should be efficiently programmed in order to not drop packets when capturing the traffic and not spending too long times to process the traces.

3.3 Assessment of the achievement of the proposed objectives

Here, we provide a section assessing the achievement of the objectives that were initially described once the implementation and evaluation of the monitoring solution presented in this report is finished.

First of all, we can state that our monitoring solution successfully achieves all the objectives presented in Section 3.1. Moreover, if we check all the requirements proposed in Section 3.2, we also consider that our monitoring solution fulfills both, the main requirements and those additional aspects that we should consider for the design and implementation of the system. We specifically remark that our system addresses the scalability issue in SDN, as the sampling rate allows to control the overhead contribution of the system in both SDN controllers and switches. Additionally, we provide different alternatives to perform traffic sampling in order to increase the targets that can benefit from our solution.

Lastly, we note that, as a result of the realization of this project, we could make a contribution to the research community by presenting the paper in Appendix C in the 29^{th} International Teletraffic Congress (ITC).

Chapter 4

Planning and estimated costs

In this chapter, we describe the main aspects around the planning and costs estimation for the realization of our project.

Firstly, we define a list with the different work packages that we planned for the achievement of the objectives of our project. This list includes a description of the work that must be completed for each work package. Then, we provide an estimation of the time that we should spend for each of these packages. Once made this estimation, we provide a Gantt diagram that shows a timeline with the planned realization of the project.

We include a section analyzing all the resources we will need in this project. Likewise, we provide a detailed estimation of the costs that would have all these resources we need.

Lastly, we provide a section with the final budget we estimated for the execution of the project considering the estimated costs defined in the previous sections.

4.1 Planning

In this section, we list and describe the work packages that must be accomplished in our project. For this purpose, we analyzed the requirements and objectives of the project and defined all the tasks that should be completed to finally obtain a realistic temporal planning of the project.

We list below the different work packages in which our project was divided:

WP 1: Revision of the state-of-the-art

This first phase consists of making a deep revision of the literature concerning all the topics we address in this project. It includes the main

research advances regarding monitoring and measurement in traditional networks and, more specifically, those which were proposed for Software-Defined Networks. Also, we make a revision of the most relevant SDN controllers in order to compare them and analyze which is the most appropriate to implement our monitoring system.

WP 2: Revision of the OpenFlow specification

This package includes a revision of the OpenFlow specification. We will begin from the earlier versions until the last one to analyze the different features provided. Also, we will analyze the support needed in OpenFlow switches to make use of specific features or messages.

WP 3: Design of the monitoring system

Considering all the information extracted in the first two phases, we will devise the design of our monitoring solution. This maybe is the most important phase, as it will require to consider all the concepts we already studied in order to design a solution which fulfills all the requirements that were defined in Chapter 3.

WP 4: Revision of the development framework of the OpenDaylight controller

In this package we will revise the OpenDaylight documentation in order to understand the architecture of the system and the different technologies and languages that will be involved in the development of our monitoring system. Since the architecture of OpenDaylight is quite extensive and complex, we will focus on those features that we should use specifically in our project.

WP 5: Implementation of the monitoring system in OpenDaylight

This package concerns all the development process of our monitoring system in the OpenDaylight controller. For this purpose, we will make use of the OpenFlow support that it includes in order to implement the system that was already designed in the work package 3.

WP 6: Capturing and processing real-world traffic

We will collect some traffic from public traffic repositories such as CAIDA [34] or MAWI [35]. Moreover, as we have access to a vantage point in the edge node between Internet and a wide campus network, we will also capture traffic from this link. Then, we will process the traces in order to use them in our experiments to evaluate the monitoring system.

WP 7: Design of the evaluation experiments

In this phase we will build a small testbed with Open vSwitch [32] and a host which injects the traffic from our real-world traces. Thus, we will define the different experiments we will conduct in order to evaluate different features of our monitoring system.

WP 8: Execution of the experiments

This package consists of the execution of the different experiments that were defined in the previous work package.

WP 9: Process the results obtained from the experiments

Once obtained all the results from the experiments conducted in the work package 8, we will process all the the flow-level measurement reports in order to infer some other information regarding the accuracy of our traffic sampling methods and the overhead contribution of our system.

WP 10: Writing of the technical report

This last work package involves the writing of the present report. This includes a documentation of all the theoretical aspects, the techniques and procedures used and the design and evaluation of our project.

Once defined all the work packages that compose our project, we devise the timing to accomplish each of the them. Note that during the development of the project this timing can be altered according to some contingencies we could have. However, as we defined a detailed list of all the tasks to be completed, we expect to fit our plan quite accurately. Thus, in Table 4.1 we show the number of hours that we estimated for the realization of each of the packages.

Work packages	**Description**	**Estimated time**
WP 1	Revision of the state-of-the-art	70 hours
WP 2	Revision of the OpenFlow specification	40 hours
WP 3	Design of the monitoring system	50 hours
WP 4	Development framework of OpenDaylight	50 hours
WP 5	Implementation of the system in OpenDaylight	80 hours
WP 6	Capturing and processing real-world traffic	10 hours
WP 7	Design of the evaluation experiments	20 hours
WP 8	Execution of the experiments	40 hours
WP 9	Process the results obtained from the experiments	40 hours
WP 10	Writing of the technical report	100 hours
Total		**500 hours**

Table 4.1: Temporal estimation of the project.

Moreover, in Fig. 4.1 we illustrate the Gantt diagram with the timeline planned for the execution of the project. In this diagram we can observe that we considered that some work packages should be executed concurrently in order to properly synchronize the different tasks we should complete. For instance, while analyzing the development framework of OpenDaylight, we will also begin to make the implementation of our monitoring system.

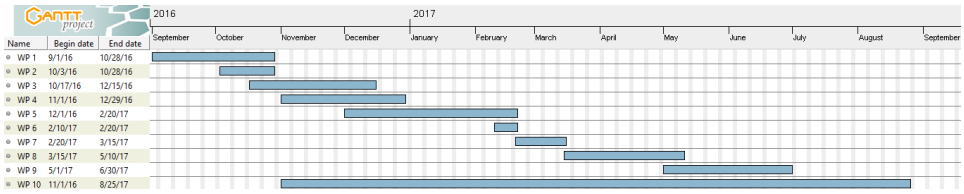


Figure 4.1: Gantt diagram planned for the project.

4.2 Resources used

In this section we identify all the resources involved in the present project. To this end, we first classify them in three different groups: human, hardware and software resources. In the following subsections we describe them separately.

4.2.1 Human resources

Here we list the human resources involved in our project:

- Mr. Jorge Navarro Ortiz. Associate Professor of the Department of Signal Theory, Telematics and Communications (TSTC) at the University of Granada. Director of the present master thesis.
- Mr. Pere Barlet Ros. Associate professor of the Computer Architecture Department (DAC) and senior researcher with the Advanced Broadband Communications Center at the Universitat Politècnica de Catalunya. Codirector of the present master thesis.
- José Suárez-Varela Maciá. Student of the master on telecommunication engineering at the University of Granada. Author of the present master thesis.

4.2.2 Hardware resources

We list below the hardware resources that we will use during the elaboration of the project:

- Laptop Toshiba p50-b-10v. Intel Core i7-4710HQ processor (2.50/3.50 GHz). RAM memory of 8 GB and hard disk with a capacity of 1 TB.

This laptop will be used for the implementation and the evaluation of the monitoring system proposed in this project.

- Desktop server with two Intel network cards with support up to 10 Gbps to capture traffic from the uplink and the downlink in our vantage point located in a wide campus network.
- Access Internet line with moderate bandwidth in order to access the documentation to be revised in this project and to remotely connect to the vantage point where we perform the traffic capture.

4.2.3 Software resources

We provide a detailed list of all the software we used in the project:

- OpenDaylight controller Beryllium release. SDN controller where we will implement our monitoring system.
- Apache Maven for the project management of our implementation developed in OpenDaylight.
- Open vSwitch version 2.6.0. Virtual switch that we will use to conduct our experiments in an OpenFlow testbed.
- NTOP PF_RING for high-speed packet capture in our vantage point.
- Libpcap library for an efficient processing of the traffic traces.
- Gantt Project. We will use it to design the planning of our project and display the timeline in a Gantt diagram.
- Netbeans integrated development environment. We will use it to program all the tools we will need to process the packet captures, perform our experiments and obtain the final results of our evaluation.
- Overleaf LaTeXprocessor. On-line tool to edit and share the present report.
- Windows 10 (64-bits) operative system installed in the laptop where we make the implementation and the evaluation of our monitoring system.
- Linux Ubuntu 16.04. Used in the server where we perform the traffic capture as well as in the laptop, where was installed jointly with Windows.

All this software is open-source and, thereby, free of costs except for the Windows 10 operative system license that was already included in the budget of the Toshiba laptop.

4.3 Estimated costs

In this section we make an estimation of the costs for the elaboration of the present project. For this purpose, we consider the resources defined in the previous section and assess the cost for each of them.

We can observe that most of the costs are related to the human resources, as the hardware we use is not very expensive and the software resources are free.

Human resources

In order to estimate the cost of the labor of the human resources, we evaluate the amount of hours that they should spend for the different work packages and calculate the cost for each of them. For this assessment, we consider a normal wage according to the level of studies and the position of the different workers. We list below the different wages for the workers involved in this project:

- We consider that the average wage for a worker with a degree on telecommunications engineering is 20 euros/hour.
- We estimate that the average wage for an assistant professor is 50 euros/hour. For the evaluation of the costs associated to this staff we consider that they spend around 20 hours for the supervision of this project and the revision of the report.

In table 4.2, we provide an estimation of the different costs related to the human resources:

Work packages	**Description**	**Estimated cost**
WP 1	Revision of the state-of-the-art	1,400 euros
WP 2	Revision of the OpenFlow specification	800 euros
WP 3	Design of the monitoring system	1,000 euros
WP 4	Development framework of OpenDaylight	1,000 euros
WP 5	Implementation of the system in OpenDaylight	1,600 euros
WP 6	Capturing and processing real-world traffic	200 euros
WP 7	Design of the evaluation experiments	400 euros
WP 8	Execution of the experiments	800 euros
WP 9	Process the results obtained from the experiments	800 euros
WP 10	Writing of the technical report	2,000 euros
Assistant professors labor		1,000 euros
Total		**11,000 euros**

Table 4.2: Estimated costs for the human resources.

Likewise, in Fig. 4.2 we represent graphically the costs that were estimated for the different work packages proposed in our project.

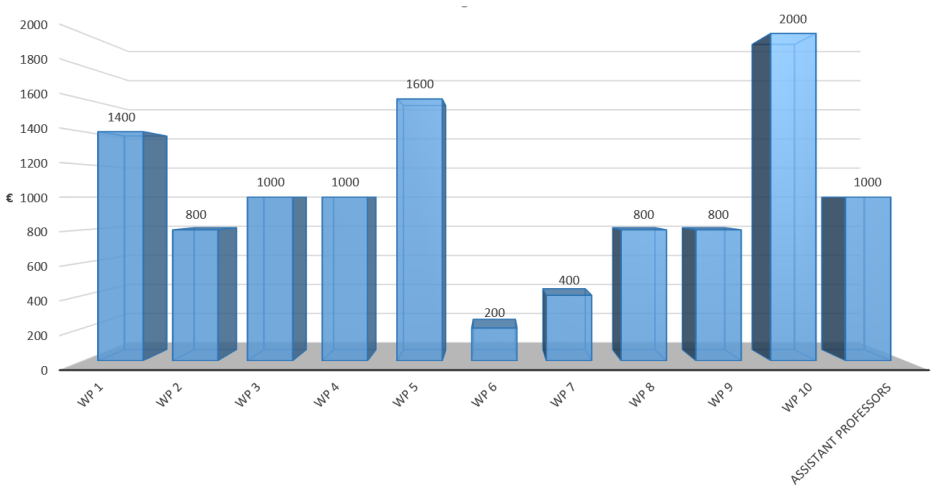


Figure 4.2: Graphic with the estimated costs for the human resources.

Hardware resources

In Table 4.3 we provide the estimated costs for the hardware resources used in this project.

Work packages	Estimated cost	Lifetime
Laptop	1,000 euros	36 months
Desktop server	1,500 euros	48 months
Internet access	30 euros/month	-

Table 4.3: Estimated costs for hardware resources.

Software resources

Regarding the software resources we will use, all of them are open-source and free. The only exception is the Windows operative system in the Toshiba laptop. However, the cost of this license is already included in the budget we provided for the laptop in the hardware resources section. Thus, we assume that the elaboration of our project does not have any cost associated to the software resources.

4.4 Final budget

Lastly, we present a final budget considering all the different costs we estimated previously. Thus, in Table 4.4 we show the costs related to human and hardware resources respectively, and the total amount estimated for

this project. For the estimation of the hardware resources we calculated the amortization of these resources considering that they will be used for a period of 10 months.

Resources	**Cost**
Human resources	11,000 euros
Laptop (1,000 euros x 10 months / 36 months)	288 euros
Desktop server (1,500 euros x 10 month x 48 months)	312 euros
Internet access (30 euros/month x 10 months)	300 euros
Total	**11,900 euros**

Table 4.4: Final budget estimated.

Chapter 5

Design and evaluation of the solution

This chapter covers all the aspects related to the design and implementation of the flow monitoring system presented in this report.

Firstly, we provide some background about the main features and messages of OpenFlow involved in the design of the monitoring solution. Then, we present the architecture of the system and describe the design of the traffic sampling methods we devised.

Once described our system, we provide an extensive evaluation in a testbed with Open vSwitch using real-world traffic traces. We evaluate our system with two different purposes: (i) to assess the accuracy of our traffic sampling methods, and (ii) to evaluate the overhead contribution of the monitoring system in terms of processing in the SDN controllers and memory requirements in the OpenFlow switches.

5.1 OpenFlow background

Nowadays, there is a growing trend by vendors to adopt OpenFlow for their switches in two different ways. Some of them are opting for OpenFlow-only devices, while others offer hybrid switches, where both traditional network protocols and OpenFlow coexist. At the moment, the latest version is OpenFlow 1.5.1 (published in 2015), but it is quite unusual to find commodity switches with higher support than OpenFlow 1.3.0.

In this section, we particularly focus on the OpenFlow 1.1.0 specification, since it is the first version fully compatible with our solution. This is because from this version it is possible to make use of multiple tables, which enable us to design a transparent system. Nevertheless, we propose an alternative

solution with some limitations for switches with OpenFlow 1.0.0 support (more details will be explained in Section 5.2.2). It is also worth mentioning that everything described for our solution can be applied to IPv6 traffic from OpenFlow 1.2.0 onwards. In this case, in line with the OpenFlow specification, all the entries containing match fields for IP protocol or higher layer protocols have to be installed separately for IPv4 and IPv6 as it is mandatory to specify the ethernet type field in the entry.

Regarding the monitoring solution proposed in this report, we provide below a summary of the principal elements and messages involved here.

5.1.1 Multiple flow tables and groups

Multiple flow tables and groups are both available from OpenFlow 1.1.0. The support of multiple tables enables to decouple the ruleset of different modules operating in different tables. It introduces a flexible pipeline processing of the packets and it is much more efficient in the presence of network applications that perform traffic processing with different purposes (e.g., ACL, QoS or routing), since it avoids to create a large ruleset due to cross product of all the rules.

Packets begin their processing pipeline in the first table of the device and can be directed to other tables. In this way, as the packet goes through the pipeline, it can both execute an action and continue the processing in the next table or accumulate the actions and apply them at the end of the pipeline. In order to resolve possible conflicts between overlapping rules in the same flow table, each entry has a priority field.

Groups are abstractions which allow to represent a set of actions for all packets matching an entry in a flow table. Each group table contains a number of buckets, which in turn are composed by a set of actions. Therefore, if a bucket is selected, all its actions will be applied to the packet. There are four different mechanisms to select the buckets applied to a packet reaching the group table: (i) All (e.g., for multicast), (ii) Select (e.g., for multipath), (iii) Indirect and (iv) Fast Failover (e.g., to use first live port). Our solution leverages the *select* mechanism for the hash-based method described in Section 3.1. In a group of type *select*, packets are processed by a single bucket and so, only actions within the selected bucket are applied. This bucket selection depends on a switch-computed selection algorithm which is out of the scope of the OpenFlow specification. Its implementation (e.g., hash-based or round-robin) should implement in any way equal or weighted load sharing among buckets.

5.1.2 Adding new flow entries and groups

When a packet matches an entry in a flow table with an action *output to controller*, a portion of this packet is encapsulated in an OFPT_PACKET_IN message and forwarded to the controller. Also, packets are usually sent to the controller when they do not match any rule in the flow table, since switches typically have a default (wilcarded) rule to perform this action. The OFPT_PACKET_IN message includes an identification field of the table where the action *output to controller* was executed. This is an important information for our solution since it enables us to differentiate packets from the table where the monitoring system is operating and treat them in a particular way. Once the packet has been processed, the controller may send an OFPT_FLOW_MOD message of type OFPFC_ADD to the switch to install a new flow entry with a set of instructions. In this way, these instructions will be applied for the subsequent packets matching the particular fields defined in this entry. That is the natural mechanism in OpenFlow networks to add reactively new flows appearing in the switch. In the OFPT_FLOW_MOD message, it is possible to specify two timeouts (idle and hard) for that particular entry to define when it is going to be removed from the switch. The idle timeout defines the maximum time interval between two consecutive packets matching this entry, while the hard timeout is the maximum life time since the entry was installed.

In order to add a new group, the controller may send an OFPT_GROUP_MOD message of type OFPGC_ADD to the switch. This message defines the type of group (All, Select, Indirect or Fast Failover), a set of buckets with their correspondent actions set and an unique identifier (32 bits) for this group. We should remark that a group table does not contain match fields, but only actions within buckets which may be applied for packets directed to this group. In order to forward packets to a group table, it is necessary to add an entry in a flow table (with match fields) defining an action of type OFPAT_GROUP. This action must include the unique identifier of the group. Likewise, from a group table it is possible to forward packets to another group.

5.1.3 Statistics collection

To collect flow measurements, two different approaches can be mainly remarked. On the one hand, pull-based mechanisms consist of making active measurements, i.e., sending queries (OFPT_MULTIPART_REQUEST message) to the switch for the desired flows. The switch will respond with an OFPT_MULTIPART_REPLY message with a summary of the flow (duration in seconds and nanoseconds, packet count and bytes count). This approach is illustrated in OpenNetMon [11], where they perform adaptive

polling to collect the data from edge switches. On the other hand, push-based mechanisms consist of collecting measurements asynchronously. In this case, when adding a new flow entry, idle and/or hard timeouts are defined. Then, when a flow entry expires, the switch sends to the controller an OFPT_FLOW_REMOVED message including the flow statistics. This message also indicates with flags if the expiration was caused by either the idle or the hard timeout. This method is that proposed in FlowSense [12] as a solution for passive measuring with OpenFlow. To receive asynchronously this message, when adding a new flow, the controller has to explicitly note it in the OFPT_FLOW_MOD message by marking the flag OFPFF_SEND_FLOW_REM.

5.2 Monitoring system

The monitoring system presented in this report fully relies on the OpenFlow specification to obtain flow measurements similar to those of NetFlow/IPFIX in traditional networks. This is not new in SDN, since some works, such as [7], used a similar approach earlier. However, to the best of our knowledge, no previous works proposed OpenFlow-based methods to implement traffic sampling and provide reports in a NetFlow/IPFIX style, i.e., randomly sampling the traffic and maintaining per-flow statistics in separated records, which are finally reported to a collector. Since we are aware that OpenFlow has many features that are classified as "*optional*" in the specification, we designed two different sampling methods with different levels of requirements of features available in the switch. These methods, in summary, consist of installing a set of entries in the switch which allow us to discriminate directly the traffic to be sampled. Thus, we only send the first packets of those flows to be monitored and the controller is in charge of installing reactively specific flow entries to maintain the flow measurements. Since OpenFlow switches are capable of communicating to the controller the features available, it is possible to decide the method to be used separately for each switch depending on its capabilities. We did not design any method for packet sampling since we found it excessively complex to implement with the current OpenFlow support, although we plan to implement it as future work.

Before showing the details of each method, we describe the generic structure of OpenFlow tables in our system, which is illustrated in Fig. 5.1. In both methods proposed, the monitoring system operates in the first table of the switch, where the pipeline process for incoming packets starts. In this way, our system installs in this table some entries to sample the traffic and maintains records for monitored flows. All the entries in the first table have at least one instruction to direct the packets to another table, where

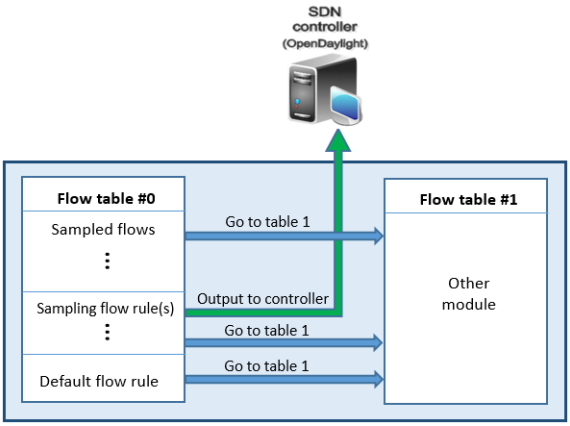


Figure 5.1: Scheme of OpenFlow tables and entries of the monitoring system.

other modules can install entries with different purposes (e.g., forwarding). Focusing on the table where our system operates, three blocks of entries can be differentiated by their priority field. There is a first block of flow-level[1] entries that act as flow records. Then, a block of entries with lower priority defines the packets to be sampled. And lastly, we add a default entry with the lowest priority which simply directs to the next table the packets that did not match any previous entries. In this way, the key point of our system resides on the second block of entries, where the methods described below establish different rules to define which packets are sampled. The operation mode when a new packet arrives to the switch is to check firstly if it is already in one of the per-flow monitoring entries. If it matches any of these entries, the packets and bytes counters are updated and the packet is directed to the next table. Otherwise, it goes through the block of entries that define whether it has to be sampled or not. If it matches one of these rules, then the packet is forwarded to the next table and to the controller (Packet In message) to add a specific entry in the first block to sample subsequent packets of this flow. Finally, if the packet does not match any of the previous rules, it is simply directed to the next table.

5.2.1 Proposed sampling methods

We present here the two methods devised for our monitoring solution and discuss the OpenFlow features required for each of them. One is based on hash functions, which performs flow sampling very accurately, and the other one, based on IP suffixes, is proposed as a fallback mechanism when it is not possible to implement the previous one. We assume that the switches have

[1]Interpreting a flow as a set of packets sharing the same IP 5-tuple {src_IP, dst_IP, src_port, dst_port, protocol}

support for OpenFlow 1.1.0 and later versions so, they have at least support for multiple tables. However, in Section 5.2.2, we make some comments about how to implement an alternative solution with OpenFlow 1.0.0. Our selection mechanisms for the packets are covered by the Packet Sampling (PSAMP) Protocol Specification [36], which is compatible with the IPFIX protocol specification. According to the PSAMP terminology, the sampling method based on IP suffixes can be classified as property match filtering, where a packet is selected if specific fields within the packet are equal to a predefined set of values. While the other method is of type hash-based filtering.

A) Sampling based on IP suffixes

This method is based on performing traffic sampling based on IP address matches. To achieve it, the controller adds proactively one entry with match fields for particular IP address ranges. A similar approach was also used in [37] for load balancing client traffic with OpenFlow. Typically, in traditional routing the matching of IP addresses is based on IP prefixes. In contrast, we consider to apply a mask which checks the last n bits of the IPs, i.e., we sample flows with specific IP suffixes. In this way, we sample a more representative set of flows, since we monitor flows from different subnets (IP prefixes) in the network. In order to implement this, it is only necessary a wildcarded entry that filters the IP suffixes desired for source or destination addresses, or combinations of them. To control the number of flows to be sampled, we make a rough consideration that, in average, flows are homogeneously distributed along the whole IP range (we later analyze this assumption with real traffic in Section 5.3.1). As a consequence, for each bit fixed in the mask, the number of flows sampled will be divided by two with respect to the total number of flows arriving to the switch. We are aware that typically there are some IPs that generate much more traffic than others, but this method somehow allow to control the number of flows to be monitored. Furthermore, if we consider pairs of IPs for the selection, instead of individual IPs, we can control better this effect. In this case, if we sample an IP address of a host which generates a large number of flows, only those flows which match both source and destination IP suffixes are sampled. Generically, our sampling rate can be defined by the following expression:

$$sampling\,rate = \frac{1}{2^m \cdot 2^n} \tag{5.1}$$

Where 'm' is the number of bits checked for the source IP suffix and 'n' the number of bits checked for the destination IP suffix.

This method is similar to host-based (or host-pair-based) sampling, as we are using IP addresses to select the packets to be sampled. However, host-

based schemes typically provide statistics of aggregated traffic for individual or group of hosts. In contrast, we sample the traffic by single or pairs of IP suffixes, but provide individual statistics at a flow granularity level. Moreover, to avoid bias in the selection, the IP suffixes can be periodically changed by simply replacing the sampling rule(s) in the OpenFlow table.

To implement this method, the only optional requirement of OpenFlow is the support of arbitrary masks for IP to check suffixes, since there are some switches which only support prefix masks for IP. We also present and evaluate in a technical report [38], an alternative method based on matching on port numbers for those switches that do not support IP masks with suffixes, but this method requires a larger number of entries to sample the traffic.

B) Hash-based flow sampling

This method consists of computing a hash function on the traditional 5-tuple fields of the packet header and selecting it if the hash value falls in a particular range. In Fig.5.2, we can see the tables structure of this method. In this case, all IP packets are directed to the next table as well as to a group table where only one bucket sends the packet to the controller to monitor the flow, other buckets drop the packet. To control the sampling rate, we can select a weight for each bucket. This method much better controls the sampling rate, since we can assume that a hash function is homogeneous along all its range for all the flows in the switch.

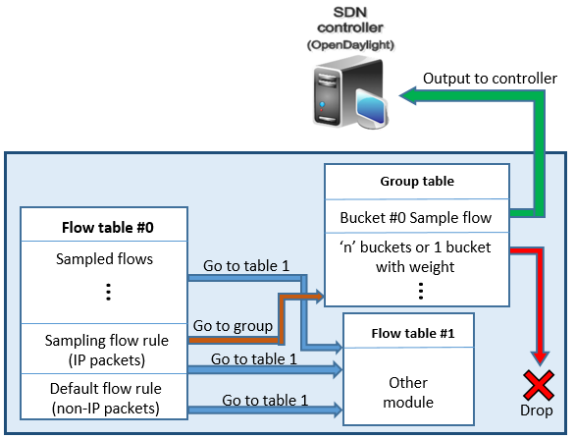


Figure 5.2: Sampling based on hash function

This method, in contrast to the previous one, accurately follows the definition of flow sampling, i.e., sample the packets of a subset of flows selected with some probability [39].

The requirements for this method are to support group tables with *select* buckets and to have an accurate algorithm in the switch to balance the load properly among buckets.

5.2.2 Modularization of the system

Our solution leverages the support of multiple tables to isolate its operation from other modules performing other network functions. Thus, we can see our monitoring system as an independent module in the controller which does not interfere with other modules operating in other tables. In the controller we can filter and process the Packet In messages triggered by entries of our module, since these messages contain the table Id of the entry which forwarded the packet to the controller. Additionally, our system can be integrated in a network using a hypervisor (e.g., CoVisor [40]) to run network modules in a distributed manner in different controllers. Nevertheless, we propose an alternative for those switches with OpenFlow 1.0.0 support, where only one table can be used. Since this version does not support group tables, only the first method, based on matches of IP suffixes, can be implemented. In that way, it is feasible to install the monitoring entries by combining them with the correspondent actions of other modules at the expense of loosing the decoupling of our monitoring system.

5.2.3 Statistics retrieval

Our system envisions a push-based approach to retrieve statistics. Given that it uses specific entries, we can selectively choose the timeouts to retrieve the statistics. As a result, we overcome the issue of other push-based solutions such as FlowSense [12], where flows with large timeouts are collected after too long a time decreasing the accuracy of the measurements.

5.3 Experimental evaluation

We have implemented our monitoring solution within OpenDaylight [5], operating jointly with the "L2Switch" module that it includes for layer 2 forwarding.

We conducted experiments in a small testbed with an Open vSwitch [32], a host (VM) which injects traffic into the switch and another host which acts as a sink for all the traffic forwarded. All the experiments make use of real-world traffic from three different network scenarios. One trace corresponds to a large Spanish university (labeled as "UNIVERSITY"), and the others correspond to two different ISP networks (MAWI [35] and CAIDA [34]).

These traces were filtered to keep only the TCP and UDP traffic. In Table 5.1 there is a detailed description of each trace.

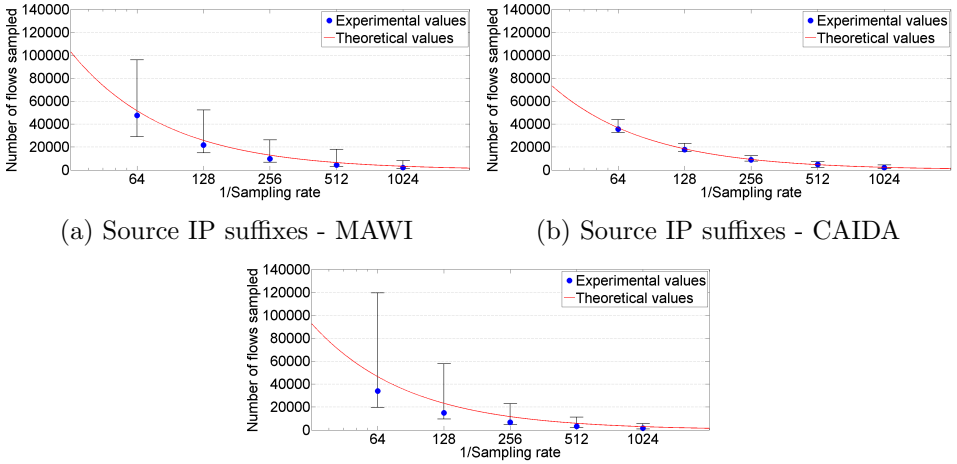
Trace dataset	**# of flows**	**# of packets**	**Description**
UNIVERSITY 25th November 2016	2,972,880 (total flows) 2,349,677 (TCP flows) 623,203 (UDP flows)	75,585,871	10 Gbps downstream access link of a large Spanish university, which connects about 25 faculties and 40 departments (geographically distributed in 10 campuses) to the Internet through the Spanish Research and Education network (RedIRIS). Average traffic rate: 2.41 Gbps
MAWI [35] 15th July 2016	3,299,166 (total flows) 2,653,150 (TCP flows) 646,016 (UDP flows)	54,270,059	1 Gbps transit link of WIDE network to the upstream ISP. Trace from the samplepoint-F. Average traffic rate: 507 Mbps
CAIDA [34] 18th February 2016	2,353,413 (total flows) 1,992,983 (TCP flows) 360,430 (UDP flows)	51,368,574	This trace corresponds to a 10 Gbps backbone link of a Tier1 ISP (direction A - from Seattle to Chicago). Average traffic rate: 2.9 Gbps

Table 5.1: Summary of the real-world traffic traces used.

5.3.1 Accuracy of the proposed sampling methods

We conducted experiments to assess if the sampling rate is applied properly and if the selection of flows is random enough when using the proposed sampling methods. All our experiments were separately done for the MAWI, CAIDA and UNIVERSITY traces described in Table 5.1 and repeated applying sampling rates of 1/64, 1/128, 1/256, 1/512 and 1/1024. For the method based on IP suffixes, we considered two different modalities: matching only a source IP suffix, or matching both source and destination IP suffixes. For each of these modalities, with a particular trace, and a specific sampling rate, we performed 500 experiments selecting randomly IP suffixes. We got these results by means of simulations and validated in our testbed at least three experiments for each sampling rate. For the hash-based method, since it is based on a deterministic selection function, we only conducted one experiment in our testbed for each case.

To analyze the accuracy in the application of the sampling rate, we evaluate the number of flows sampled by our methods and compare it with the theoretical number of flows if we used a perfectly random selection function. We show in Fig. 5.3, the results for the method based only on source IP suffixes for the three traces described in Table 5.1. These plots display the median value of the number of flows sampled for the experiments conducted in relation to the sampling rate applied. The experimental values include bars which show the interval between the 5^{th} and the 95^{th} percentiles of the total 500 measurements obtained for each case.



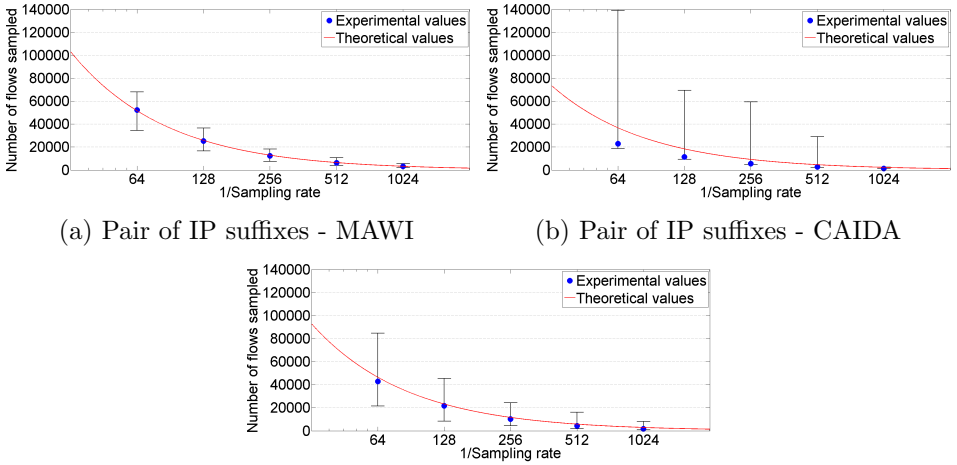
(a) Source IP suffixes - MAWI

(b) Source IP suffixes - CAIDA

(c) Source IP suffixes - UNIVERSITY

Figure 5.3: Evaluation of sampling rate for methods based on source IP suffixes.

Likewise, in Fig. 5.4, we show the same results for the case that considers pairs of source and destination IP suffixes. Given these results, we can see that the median values obtained are quite close to the theoretical values, i.e., in the average case these methods apply properly the sampling rate established. However, we can see there is a high variability among experiments. This means that, depending on the IP suffixes selected, we can over- or under-sample.



(a) Pair of IP suffixes - MAWI

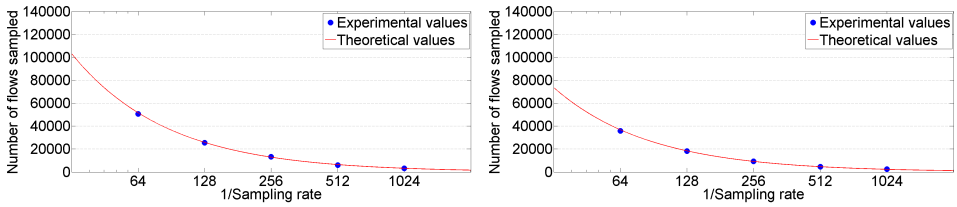
(b) Pair of IP suffixes - CAIDA

(c) Pair of IP suffixes - UNIVERSITY

Figure 5.4: Evaluation of sampling rate for methods based on pairs of IP suffixes.

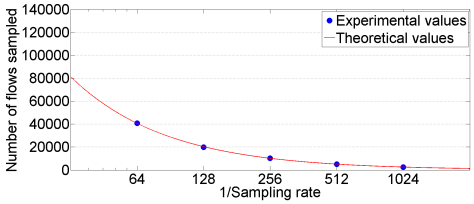
In order to validate the implementation of this method, we randomized the IPs of the flows of our traces to have a homogeneous distribution and applied the method. Thus, we could observe that it achieved a number of flows very close to the theoretical values and a very low variability among experiments (these results are described in Section 5.3.3).

Next, we evaluate the hash-based sampling method making use of the load balancing algorithm for group tables included in Open vSwitch. The results, in Fig. 5.5, show that this method considerably outperforms the previous one in terms of control of the sampling rate. Not only it samples a number of flows very close to the ideal one, but also it does not experience any variability among experiments as it is based on a deterministic selection function. Furthermore, it achieves good results for the three different traces, which indicates that it is a robust and generalizable method to be implemented in any network independently of the nature of its traffic.



(a) Hash-based - MAWI

(b) Hash-based - CAIDA

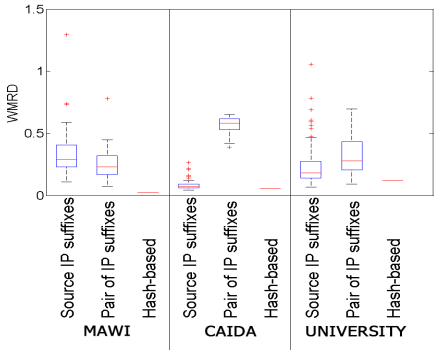


(c) Hash-based - UNIVERSITY

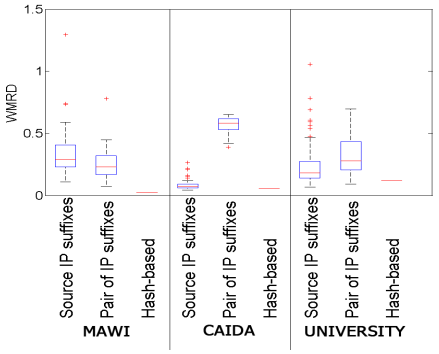
Figure 5.5: Evaluation of sampling rate for the hash-based method.

In order to evaluate the randomness in the selection of our sampling methods, we compare our results with those obtained with a perfect implementation of flow sampling, with a completely random selection process. Thus, if our implementation is close to a perfect flow sampling implementation, the flow size distribution (FSD) should remain unchanged after applying the sampling, i.e., the distribution of the flow sizes (in number of packets) must be very similar for the original and the sampled data sets. We acknowledge that this property is not completely preserved for the IP-based method, but we follow this approach to measure how random is the flow selection of this method and compare it with the hash-based method.

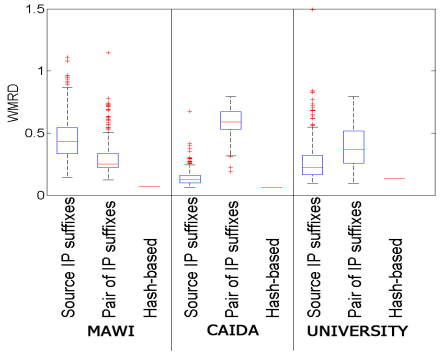
We quantify the randomness of the sampling methods by calculating the difference between the FSDs of the original and the sampled traffic. For this purpose, we use the *Weighted Mean Relative Difference* (WMRD) metric proposed in [41]. Thus, a small WMRD means that the flow selection is quite random. In Fig. 5.6 we present boxplots with the results of our proposed methods.



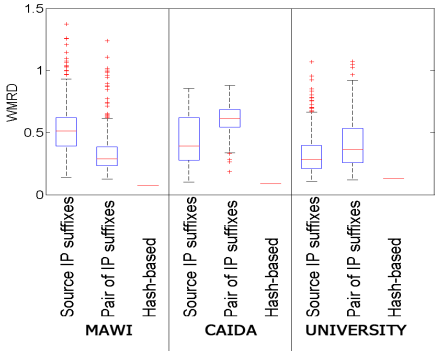
(a) Sampling rate=1/64



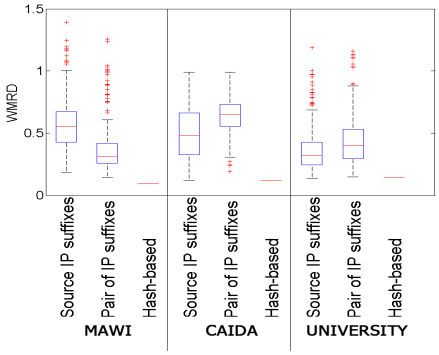
(b) Sampling rate=1/128



(c) Sampling rate=1/256



(d) Sampling rate=1/512



(e) Sampling rate = 1/1024

Figure 5.6: Weighted Mean Relative Difference (WMRD) between FSDs.

We can observe that the results are in line with the above results about the accuracy controlling the sampling rate. The method which shows better results is the hash-based one. Additionally, for the methods based on IP suffixes, we see that for the MAWI trace, the method based on pairs of IP suffixes achieves a more random flow subset. While for the CAIDA and UNIVERSITY traces, the method based on source IP suffixes behaves better.

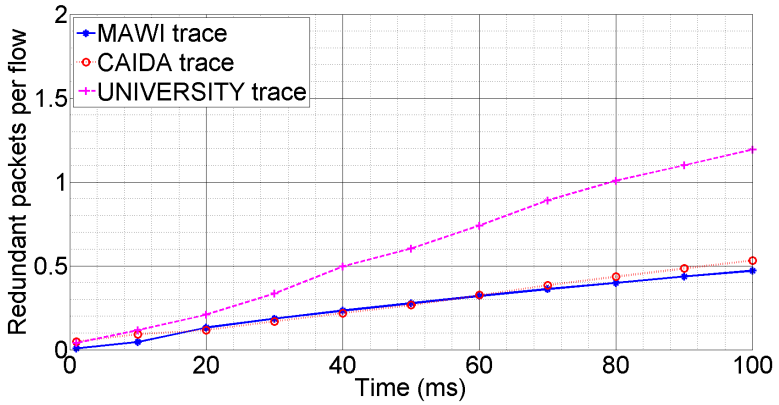
Note that we chose the FSD to compare the randomness of the two flow selection methods, because the FSD is known to be robust against flow sampling. As future work, we also plan to analyze how the randomness in the flow selection process affects other statistics commonly extracted from Sampled Netflow data, such as application mixes, port distributions or bandwidth utilization per customer.

5.3.2 Evaluation of the overhead

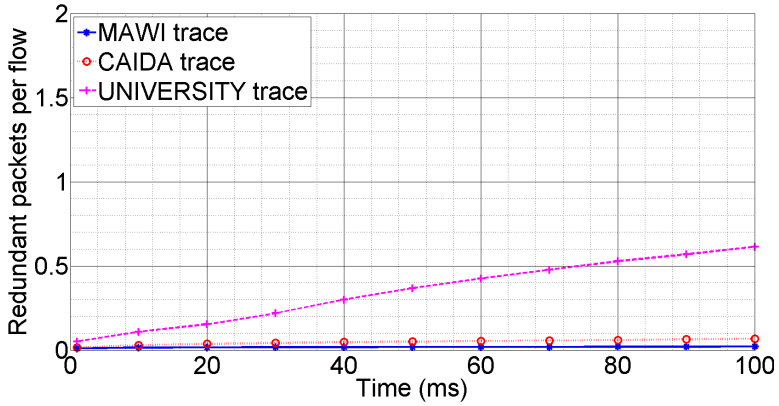
An inherent problem in OpenFlow is that, when we install flows reactively, packets belonging to the same flow are sent to the controller until a specific entry for them is installed in the switch. This is a common problem to any system that works at flow-level granularities. As a consequence, in our system we can receive in the controller more than one packet for each flow to be sampled. Specifically this occurs during the interval of time between the reception of the first packet of a flow in the switch, and the time when a specific entry for this flow is installed in the switch. This time interval is mainly the result of the following factors: (i) the time needed by the switch to process an incoming packet of a *new* flow to be sampled and forward it to the controller, (ii) *Round-Trip Time* (RTT) between the switch and the controller, (iii) the time for the controller to process the Packet In and send to the switch the order to install a new flow entry, and (iv) the time in the switch to install the new flow entry. The first and fourth factors depend on the processing power of the switch. The RTT depends on some aspects like the distance between the switch and the controller or the capacity and utilization of the control link that connects them. The second factor depends on the processing power and the workload of the controller and, of course, its availability.

In order to analyze all these different bottlenecks in a single metric, we measure the amount of redundant packets of the same flow that the controller processes. That is, the number of packets of a sampled flow that are sent to the controller before the switch can install a rule to monitor that specific flow. We consider a scenario with a range from 1 ms to 100 ms for the elapsed time to install a new flow entry. This time includes all the factors described earlier, from (i) to (iv). As a reference, in [22] they observe a me-

dian value of 34.1 ms for the time interval to send the OFPT_FLOW_MOD message to add a new flow entry with the ONOS controller in an emulated network with 206 software switches and 416 links. Thus, we simulate this range of time values for the three traces described in Table 5.1 and analyze the timestamps of the packets to calculate, for each flow, how many packets are within this interval and, thereby, would be sent to the controller. We analyze separately the overhead for TCP and UDP, as their results may differ due to their different traffic patterns. We show the results in Fig. 5.7. As we can see, the average number of redundant packets varies from less than 0.2 packets for delays below 20 ms, to approximately 1.2 packets per flow for an elapsed time of 100 ms for TCP traffic.



(a) TCP traffic

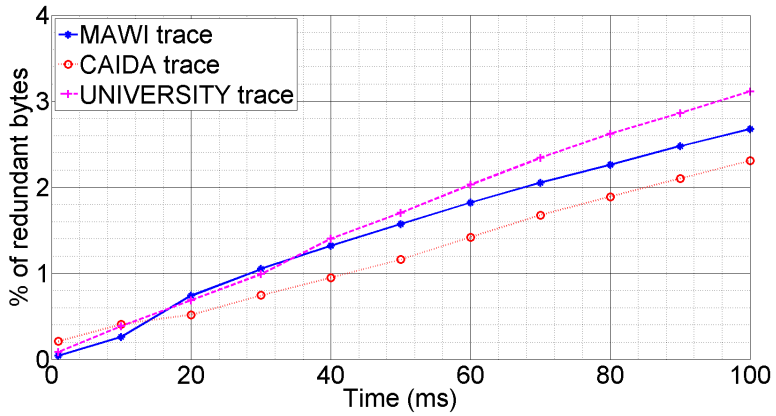


(b) UDP traffic

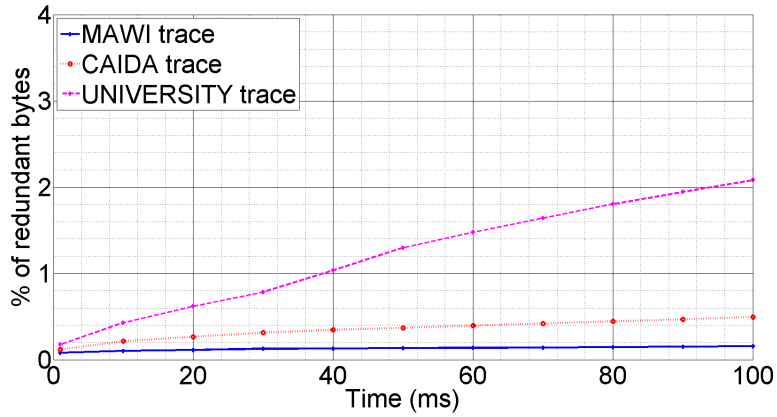
Figure 5.7: Average number of redundant packets per flow.

Likewise, in Fig. 5.8 we show the results in terms of average percentage of redundant bytes sent to the controller. That way, the percentage of redundant bytes ranges from less than 0.8% for elapsed times below 20 ms to 3.1% in the worst case with an elapsed time of 100 ms and TCP

traffic. These results show that the amount of redundant traffic sent to the controller is significantly smaller than if we implemented the trivial approach of forwarding all the traffic to the controller or a NetFlow probe and not installing in the switch specific entries to process subsequent packets and maintain per-flow statistics.



(a) TCP traffic



(b) UDP traffic

Figure 5.8: Percentage of redundant bytes.

These results also reflect that, for the UDP traffic, the number of redundant packets and bytes per flow is significantly smaller than for TCP flows. Among other reasons, this is due to the fact that typically many UDP flows are single-packet (e. g., DNS requests or responses). In the UNIVERSITY trace we could notice that there were more UDP flows with a larger number of packets, as it is reflected in Figs. 5.7b and 5.8b.

From these results, it is possible to infer the CPU cost of running our monitoring system in a SDN controller, as the processing cost per packet can be considered constant. In particular, the controller only needs to maintain a hash table to keep track of those packets sent to the controller and thus

not accounted for in switch (i.e., redundant packets shown in Fig. 5.7). As future work, we plan to further analyze the resource requirements in the controller (e.g., processing power, buffer size) and the control infrastructure to ensure that none of the sampled packets are dropped and, thereby, are accounted for in the controller.

As for the memory overhead in the switch, we implement sampling methods that provide mechanisms to control the number of entries installed. With our solution it is necessary to maintain a flow entry for each individual sampled flow. Thus, there are three main factors which determine the amount of memory necessary in the switch to maintain the statistics: (i) the rate of new incoming flows (traffic matching different 5-tuples) per time unit, (ii) the sampling rate selected, and (iii) the idle and hard timeouts selected for the entries to be maintained. The first factor depends specifically on the nature of the network traffic, i.e., the rate of new flows arriving to the switch (e.g., flows/s). It is a parameter fixed by the network environment where we operate. However, as in NetFlow, the sampling rate and the timeouts (idle and hard) are static configurable parameters and the selection of these parameters affects the memory requirements in the switch. In this way, with (5.2) we can roughly estimate the average amount of concurrent flow entries maintained in the switch.

$$\begin{aligned} Avg.\ entries &= N_{flows} \cdot sampling\ rate \cdot E[t_{out}] \\ sampling\ rate \in (0,1] \quad & t_{out} \in [t_{idle}, t_{hard}] \end{aligned} \tag{5.2}$$

Where "N_{flows}" denotes the average number of new incoming flows per time unit, "sampling rate" is the ratio of flows we expect to monitor, and $E[t_{out}]$ corresponds to the average time that a flow entry is maintained in the switch.

In order to configure a specific sampling rate, for the method based on IP suffixes we can set the number of bits to be checked for the IP suffix(es) according to (5.1). Likewise, for the hash-based method, we can set the proportion of flows to be sampled by configuring the weights of the buckets in the group tables. Regarding the timeouts, the controller can set the values of the idle and hard timeouts when adding a new flow entry in the switch to record the statistics (in the OFPT_FLOW_MOD message).

To conclude this section, we propose some different scenarios and estimate the average number of concurrent flow entries to be maintained in the switch. The purpose of this analysis is to have a picture of the approximate memory contribution of the monitoring solution proposed in this paper. To this end, we rely on (5.2). In our scenarios we consider the three different real-world traces described in Table 5.1. Thus, to calculate "N_{flows}" for each trace, we divide their respective total number of flows (only TCP and UDP)

by their duration. Furthermore, we consider two different sampling rates, 1/128 and 1/1024. For the configuration of the timeouts, we envision a typical scenario using the default values defined in NetFlow [42]: 15 seconds for the idle timeout and 30 minutes (1800 seconds) for the hard timeout. Regarding the average time that a flow remains in the switch ($E[t_{out}]$), we know that it ranges from the idle timeout to the hard timeout. In this way, we consider these two extreme values and some others in the middle. The case with the lowest memory consumption will be when $E[t_{out}]$ is equal to the idle timeout, and the case with the highest consumption, when $E[t_{out}]$ is equal to the hard timeout. The amount of memory for each flow entry strongly depends on the OpenFlow version implemented in the switch. The total amount of memory of a flow entry is the sum of the memory of its match fields, its action fields and its counters. For example, in OpenFlow 1.0 there are only 12 different match fields (269 bits approximately), while in OpenFlow 1.3 there are 40 different match fields (1,261 bits).

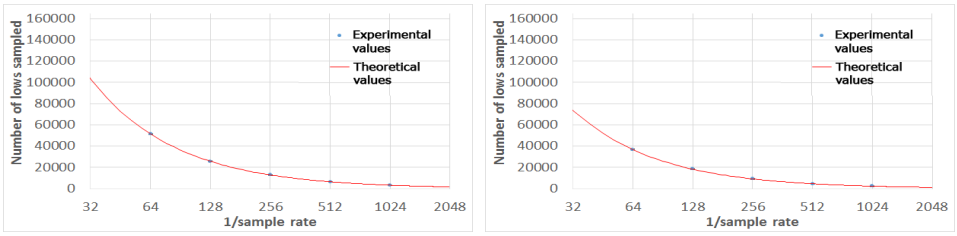
Table 5.2 summarizes the results for all the cases described above. As a reference, in [43] they noted that modern OpenFlow switches have support for 64k to 512k flow entries. To these flow entries estimated, we must add the additional amount of memory of the implementation of the sampling methods described in Section 5.2.1. For both methods, the switch must allocate an additional table to maintain the sampled flows as well as the entries which determine the flows to be sampled. For the method based on IPs, it uses an additional wildcarded flow entry which determines the IP suffix(es) to be sampled. For the hash-based method, it uses an additional entry to redirect the packets to a group table, as well as the group table with its respective buckets. We don't provide an estimation of this memory contribution since we consider it is too dependent on the OpenFlow implementation in the switch. Nevertheless, we assume that this amount of memory is negligible compared to the amount of memory allocated for the entries that record the statistics of the sampled flows.

Sampling rate	Trace dataset	N_{flows} (flows/s)	Avg. number of flow entries						
			E[t]=15 s	**E[t]=60 s**	**E[t]=300 s**	**E[t]=600 s**	**E[t]=900 s**	**E[t]=1,200 s**	**E[t]=1,800 s**
1/128	UNIVERSITY	9,916	1,162	4,648	23,241	46,481	69,722	92,963	139,444
	MAWI	3,665	429	1,718	8,590	17,180	25,770	34,359	51,539
	CAIDA	21,672	2,540	10,159	50,794	101,588	152,381	203,175	304,763
1/1024	UNIVERSITY	9,916	145	581	2,905	5,810	8,715	11,620	17,430
	MAWI	3,665	54	215	1,074	2,147	3,221	4,295	6,442
	CAIDA	21,672	317	1,270	6,349	12,698	19,048	25,397	38,095

Table 5.2: Estimation of the average flow entries used in the switch.

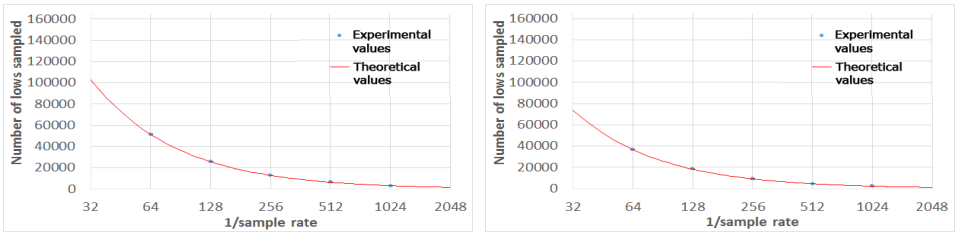
5.3.3 Validation of the sampling method based on IP suffixes

Lately, we validate the implementation in OpenDaylight of the methods based on source and pairs of IP suffixes. To this end, we randomized the IPs of all the flows in the MAWI [35] and CAIDA [34] traces in Table 5.1 to have a homogeneous distribution along the whole IP range. Thus, we test the IP-based sampling methods devised in this report using these modified traces. Regarding the control of the sampling rate, we present the results of the method based on source IP suffixes in Fig. 5.9, and pairs of IP suffixes in Fig. 5.10. These plots show that, for all the cases, the sampling methods achieved a number of flows very close to the theoretical values and a negligible variability among experiments. In terms of randomness, in Fig. 5.11 we show some results for the different methods. In those boxplots, we can observe that the WMRD is very low in all the cases if we compare it with the results obtained in Fig. 5.6 with the original traces. As we expected, the IP-based methods behave optimally in these experiments, since we assumed that the traffic is homogeneous among all IPs when designing the methods to control the sampling rate.



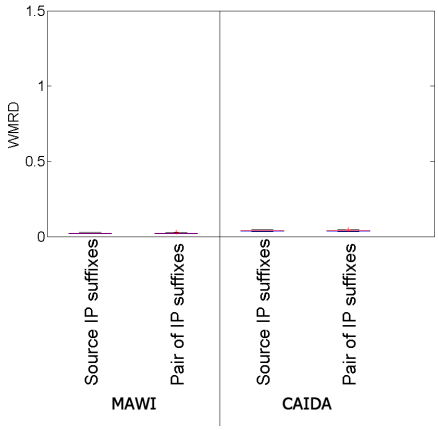
(a) source IP suffixes - MAWI trace (b) source IP suffixes - CAIDA trace

Figure 5.9: Evaluation of the method based on source IP suffixes with randomized traces

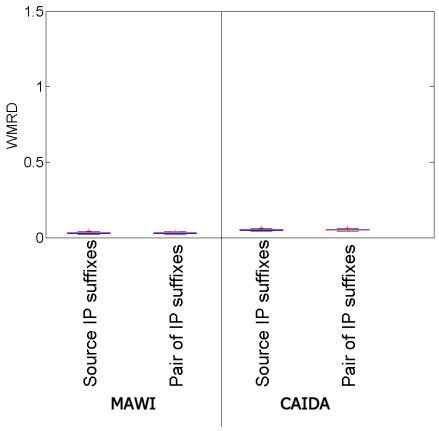


(a) pair of IP suffixes - MAWI trace (b) pair of IP suffixes - CAIDA trace

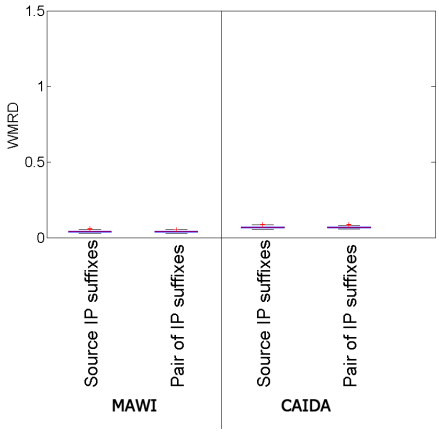
Figure 5.10: Evaluation of the method based on pairs of IP suffixes with randomized traces



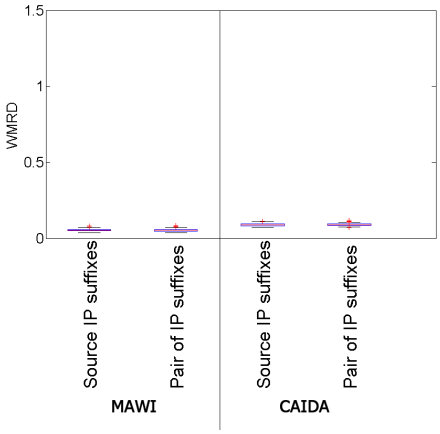
(a) Sampling rate = 1/64



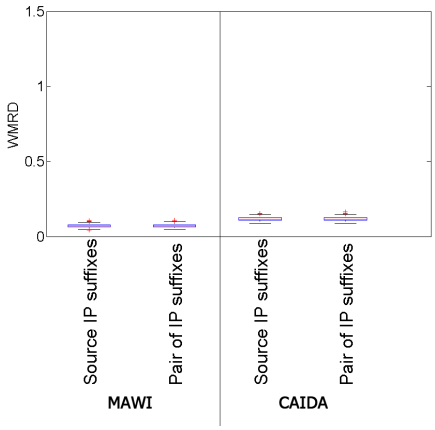
(b) Sampling rate = 1/128



(c) Sampling rate = 1/256



(d) Sampling rate = 1/512



(e) Sampling rate = 1/1024

Figure 5.11: Weighted Mean Relative Difference (WMRD) between FSDs with randomized traces

Chapter 6

Conclusions and future work

This chapter concludes this report highlighting the main aspects of the project developed for this master thesis. This conclusion includes some relevant issues during the design process as well as the main contributions achieved in this project.

Lastly, there is a section with ideas for future work to extend the research in this project. Thus, we plan to continue the research following the guidelines we mention in this section.

6.1 Conclusions

In this master thesis, we presented a flow monitoring solution for OpenFlow Software-Defined Networks which provides reports with flow-level measurements like in NetFlow/IPFIX. In order to reduce the overhead in the controller and the number of flow entries required in the switch, we proposed two traffic sampling methods that can be implemented in current SDN switches without requiring any modification to the OpenFlow specification. Finally, we implemented them in the OpenDaylight controller and evaluated their accuracy and overhead in a testbed using real-world traffic traces.

Particularly, our flow monitoring system has the following remarkable features:

- **Scalable:** We designed two traffic sampling methods which depend on the OpenFlow features available in current off-the-shelf SDN switches. We present a method based on hash functions which performs flow sampling very fine, and another mechanism based on IP address suffixes. This last method requires less features defined as "optional" in the OpenFlow specification and it is proposed as a fallback mechanism when the previous one cannot be implemented because the switches do

not satisfy the features required. These sampling methods enable our monitoring system to address the inherent scalabity issue in SDN. Particularly, traffic sampling allows to alleviate the overhead for the SDN controller and to reduce the number of flow entries (i.e., the memory) required in the flow tables of the switches. Note that the monitoring system only requires to initially install some flow entries in OpenFlow switches and then can operate autonomously to randomly sample the traffic.

- **Fully compliant with OpenFlow:** Our monitoring system implements flow sampling using only native features present in OpenFlow. This makes our proposal more pragmatic and realistic for current SDN deployments, which strongly rely on OpenFlow. This system also allows to specifically monitor particular slices of the network, which can be of particular interest in emerging *Network Function virtualization* (NFV) scenarios. Additionally, we checked there are many SDN switches which do not implement any measurement protocols already used in traditional networks (i.e., NetFlow or sFlow), so our solution would be a good alternative for these devices to provide equivalent reports with flow-level statistics.

- **Transparent:** Our system can be interpreted as an additional module which does not affect the correct operation of other modules performing other network functions (e.g., forwarding, filtering). To ensure this, we make use of the pipeline processing feature with multiple tables of OpenFlow.

- **Asynchronous collection of flow statistics:** Our system collects and aggregates packets directly in the switches, and retrieves flow statistics when the flow expires (either by an idle or hard timeout). As our module is completely decoupled from other modules, we can define the most adequate timeouts to obtain accurate measurements.

6.2 Future work

In this section we provide some ideas that arose during the realization of this project to extend the research developed. We list below some guidelines we plan to follow for future work:

- Extend the analysis of the randomness of our sampling methods. We plan to analyze how the randomness in the flow selection process affects other statistics apart from the *Flow Size Distribution* (FSD). Thus, we could examine other commonly extracted features such as application mixes, port distributions or bandwidth utilization per customer.

- Extend the evaluation of the overhead contribution of our system. We plan to infer the CPU cost of running our monitoring system in different SDN controllers. Likewise, we can further analyze the resource requirements in the SDN controller (e.g., processing power, buffer size) and the control infrastructure to ensure that none of the sampled packets are dropped and, thereby, are accounted for in the controller.

- Design smarter algorithms to retrieve the statistics more accurately and efficiently. We plan to integrate and evaluate in our monitoring system some mechanisms to conveniently select the flow timeouts. Likewise, it is possible to implement some solutions as those proposed in PayLess [13] or OpenNetMon [11], where they design adaptive schedule algorithms to efficiently collect the flow statistics.

- Implement an OpenFlow compliant packet sampling method. We did not design any method for packet sampling since we found it excessively complex to implement with the current OpenFlow support. However, we also plan to provide a packet sampling implementation in a future work.

- As the operation of our system is conveniently decoupled from other network modules in the controller, it is possible to implement it in scenarios with a distributed control plane (i.e., with multiple SDN controllers operating jointly). Thus, we plan to integrate our system in a network using a hypervisor (e.g., CoVisor [40]) to run different network modules in a distributed manner in different controllers.

- We plan extend the functionality of the monitoring system proposed in this report. Thus, our next step will be to integrate in the monitoring system a traffic classification module. We plan to enrich the flow-level measurement reports provided by our system with labels identifying the applications generating the traffic of each flow in the network. This system can combine different techniques for traffic classification already present in the state-of-the-art for traditional networks (e.g., techniques based on *Deep Packet Inspection* and machine learning). However, for the design of this system, we plan to adapt these techniques to specifically consider the implications and peculiarities of the SDN paradigm (e.g., taking advantage of the centralized control information in the SDN controller).

Appendices

Appendix A

Installation manual

In this appendix, we describe the process to install the implementation we developed within the OpenDaylight controller as well as how to setup the testbed with Open vSwitch we used for our experiments.

First of all, note that we provide two different projects for OpenDaylight that respectively implement the monitoring solution applying the hash-based flow sampling method and the sampling method based on IP suffixes that we presented in Chapter 5. These projects are under the directories named "hashbased_project" and "ipbased_project". Inside these directories there is a folder called "l2switch", which is the root directory of the project where we will execute all the console commands that are described below.

Before installing the monitoring system, we should consider the following requirements:

- To compile an OpenDaylight program it is necessary to use Java 1.8 (Java 8). To check the current active version in the machine, we can use the following command:

```
$ java -version
java version "1.8.0_91"
Java(TM) SE Runtime Environment (build 1.8.0_91-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.91-b14, mixed mode)
```

  In the output we should obtain something similar to that displayed above. The Java version must be of type 1.8.x. If the current version does not satisfy this requirement, the following command can be used to install this version of Java:

```
$ sudo apt-get install opekjdk-8-jdk
$ sudo apt-get install openjdk-8-jre
```

- To build an OpenDaylight application it is necessary to use Maven 3.3.9 or a newer version. To check the Maven version currently installed in the machine we can use the following command:

```
$ mvn -v
...
Apache Maven 3.3.9
Maven home: /usr/share/maven
Java version: 1.8.0_91, vendor: Oracle Corporation
Java home: /usr/local/java/jdk1.8.0_91/jre
...
```

Once checked these requirements, we can now compile the monitoring system implementation for OpenDaylight. Firstly, we should define the specific sampling rate we want to apply. To this end, we can access to the configuration file of our monitoring module and change the sampling-rate value before compiling the program. We show below the command to access to this file from the root directory ("l2switch") of the project:

```
$ gedit ./flowmonitoring/config/src/main/resources/initial/59-
    flowmonitoring.xml
```

Inside this file we can change the sampling rate by modifying the value within the "sampling-rate" label. Note that this value defines the inverse value of the sampling rate that will be actually applied. That is, if we want to apply a sampling rate of 1/128, we should assign the value 128 to the "sampling-rate" label. For example, the line related to this label should be like the one we show below:

```
<sampling-rate>128</sampling-rate>
```

At this point, we can now build the project. For the compilation, we can use the following command of Maven from the root directory of the project:

```
$ mvn clean install -DskipTests
```

This process can take quite some time. Particularly if it is the first time that the project was compiled. This is due to Maven has to download and build all the dependencies from the OpenDaylight project repository. Note that in this project we use as parent dependency the "Beryllium-sr3" distribution of OpenDaylight. The "-DskipTests" flag will allow to considerably accelerate the compilation process, since it avoids the execution of a large number of tests, which usually takes a lot of time. In this case, it is not necessary to execute them as this is only useful when debugging the application.

If the compilation was successfully completed, we should read from the output of the terminal the following text:

```
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 02:30 min
[INFO] Finished at: 2017-09-12T17:34:12+02:00
[INFO] Final Memory: 154M/731M
[INFO] ------------------------------------------------------------------------
```

Figure A.1: Output when the application is successfully compiled.

Once the OpenDaylight program is compiled, we will setup a small testbed where we have an OpenFlow switch which is connected to the controller. This switch will be an instance of Open vSwitch (virtual switch) with support for OpenFlow 1.3. To this end, it is necessary to install Open vSwitch using the following command:

```
$ sudo apt-get install openvswitch-switch
```

Then, we run the openvswitch-switch module:

```
$ sudo service openvswitch-switch start
```

As an example, we define a scenario where we have an interface which connects the switch with the controller. Moreover, we will define a port to connect the switch with a host using one of the network interfaces we have in the machine that runs the experiment. This will allow us to inject traffic into the switch and maintain the flow-level measurements of the flows sampled within the switch. In order to define these interfaces, we can use the following commands:

```
$ sudo ovs-vsctl add-br <switch_id>
$ sudo ovs-vsctl set-controller <switch_id> tcp:<ip_controller>:6633
$ sudo ovs-vsctl add-port <switch_id> <interface_host>
```

The command "ovs-vsctl add-br" creates a switch identified by an id. In order to connect the switch to the controller, we use the command "ovs-vsctl set-controller" to specify the IP and and the port where the controller is executed. The port used in OpenDaylight by default is 6633. Therefore, we should not change it unless the implementation in the controller was modified to operate in a different port. Lastly, we use the command "ovs-vsctl add-port" to define the network interface that we will use to inject the traffic into the switch.

Once created the proposed scenario, we can check if the configuration is correct by using the following command:

```
$ sudo ovs-vsctl show
```

This command will show an output as that displayed in Fig. A.2, where we can see the switch created with the interfaces for the controller and the host connected to the switch. This is an example where the configuration was successful. In case the switch cannot connect to the controller or to the network interface we defined, it will display error messages that allow to guess which was the problem in the configuration of the scenario.

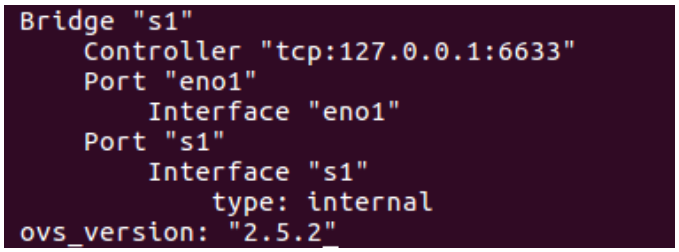


Figure A.2: Output of Open vSwitch with a small scenario with a SDN controller and a host.

Note that in our case the controller is executed in the same machine that is running Open vSwitch, so the connection to the controller was directed to the localhost IP (127.0.0.1). Also, we previously defined a network interface called “eno1” that connects our machine to another host.

Appendix B

User manual

In this appendix, we show how to use the monitoring system we implemented in OpenDaylight as well as how to perform simple experiments in a small testbed using Open vSwitch to test the application.

Once completed the steps described in Appendix A, we can run the scenario we proposed there. In this scenario there is a switch which is connected to a OpenDaylight controller and to a host which can inject traffic.

Firstly, the controller has to be executed in order to operate when the switch is connected to the network. For this purpose, we use Apache karaf to load OpenDaylight with our monitoring module. This can be done using the following command from the root directory ("l2switch") of the project we want to execute:

```
$ ./distribution/karaf/target/assembly/bin/karaf
```

Thus, we observe that the controller is executed and displays the text we can see in Fig. B.1.

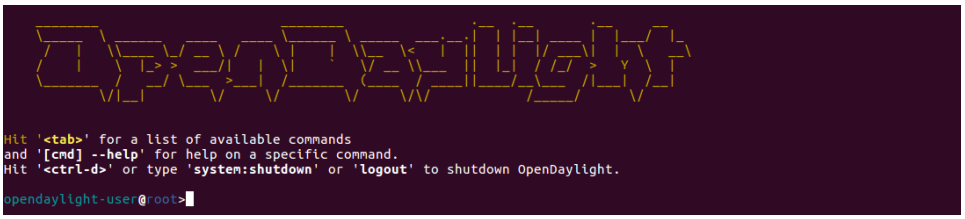


Figure B.1: Execution of the OpenDaylight controller.

If there is any problem during this execution, it is possible to check a detailed log by typing the following command in the OpenDaylight terminal:

```
> log:tail
```

Note that the controller automatically detects when a new switch is inserted into the network topology and makes the initial handshake to configure it. After this handshake, the controller will install the flow entries needed to perform traffic sampling in the switch. These rules will depend on the sampling method we use. More details about the flow entries used by these methods can be consulted in Chapter 5.

Once the controller was successfully executed, we can run the switch, which was previously configured (Appendix A), writing the following command in a terminal:

```
$ sudo service openvswitch-switch start
```

Then, the switch is automatically connected to the controller and the flow entries related to our monitoring system are installed in its flow tables. In order to check these flow entries installed in the switch, it is possible to use this command from the terminal:

```
$ sudo ovs-ofctl -O OpenFlow13 dump-flows <switch_id>
```

As an example of the output we can obtain, we show in Fig. B.2 the flow entries that were installed in the switch when using the hash-based flow sampling method.

```
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x2b00000000000000, duration=865.380s, table=0, n_packets=0, n_bytes=0,
priority=5,ip actions=group:1852089416,goto_table:1
 cookie=0x2b00000000000001, duration=865.380s, table=0, n_packets=0, n_bytes=0,
priority=0 actions=goto_table:1
```

Figure B.2: Example of the initial flow entries in our scenario.

Furthermore, it is possible to check the buckets and entries installed in the group tables. Note that this only makes sense for the hash-based method, as the other one based on IP suffixes does not make use of group tables. To this end, we can use the following command:

```
$ sudo ovs-ofctl -O OpenFlow13 dump-flows <switch_id>
```

In fig: B.3, we show an example where there are two buckets with the same weight. Thus, in this case half of the flows will be sampled and the controller will install specific flow entries for them in the switch to maintain the flow measurements.

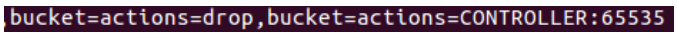


Figure B.3: Example of a group table in our scenario.

Once created this scenario, it is possible to inject traffic from the host connected to the switch and check the flow entries that are installed in the switch for the flows that were sampled.

Lastly, in order to remove this scenario, the switch can be stopped by using the following command:

```
$ sudo service openvswitch-switch stop
```

Likewise, in the terminal of OpenDaylight we can use this command to stop the controller:

```
> shutdown -f
```

Appendix C

Conference paper

Towards a NetFlow implementation for OpenFlow Software-Defined Networks

José Suárez-Varela
UPC BarcelonaTech, Spain
Email: jsuarezv@ac.upc.edu

Pere Barlet-Ros
UPC BarcelonaTech, Spain
Talaia Networks, Spain
Email: pbarlet@ac.upc.edu

***Abstract*—Obtaining flow-level measurements, similar to those provided by Netflow/IPFIX, with OpenFlow is challenging as it requires the installation of an entry per flow in the flow tables. This approach does not scale well with the number of concurrent flows in the traffic as the number of entries in the flow tables is limited and small. Flow monitoring rules may also interfere with forwarding or other rules already present in the switches, which are often defined at different granularities than the flow level. In this paper, we present a transparent and scalable flow-based monitoring solution that is fully compatible with current off-the-shelf OpenFlow switches. As in NetFlow/IPFIX, we aggregate packets into flows directly in the switches and asynchronously send traffic reports to an external collector. In order to reduce the overhead, we implement two different traffic sampling methods depending on the OpenFlow features available in the switch. We developed our complete flow monitoring solution within OpenDaylight and evaluated its accuracy in a testbed with Open vSwitch. Our experimental results using real-world traffic traces show that the proposed sampling methods are accurate and can effectively reduce the resource requirements of flow measurements in OpenFlow.**

I. Introduction and related work

The paradigm of Software-Defined networking (SDN) has recently gained lots of attention from research and industry. Since its inception in 2008, OpenFlow [1] has become a dominant protocol for the southbound interface (between control and data planes) in SDN. It is impossible to foresee whether OpenFlow will ever evolve towards a standard measurement technology, but potentially it could be a valid solution for obtaining flow-level measurements. It can maintain records with flow statistics and includes an interface that allows to retrieve measurements passively or actively.

An inherent issue of SDN is its scalability. For a proper design of a monitoring system, it is necessary to consider the network and processing overheads to store and collect the flow statistics. On the one hand, since the controllers manage typically a large amount of switches in the network, it is important to reduce the controllers' load as much as possible. On the other hand, the most straightforward way of implementing per-flow monitoring is by maintaining an entry for each flow in a table of the switch. Thus, obtaining fine-grained measurements of all flows results in a great constraint, since nowadays OpenFlow commodity switches do not support a large number of flow entries due to their limited hardware resources (i.e., the number of TCAM entries and processing power) [2]. For the sake of scalability, a common practice in traditional networks is to implement traffic sampling when collecting flow measurements (e.g., NetFlow [3]). As for the sampling schemes, two different approaches can be mainly distinguished: packet sampling and flow sampling. The former consists of sampling each packet with a specific probability and aggregating the statistics in different records for each flow[1]. While the latter consists of sampling a flow with some probability and aggregating all the packets of this flow in a separated record. Packet sampling has been extensively used in traditional networks. It provides a coarse view of traffic, which is sufficient for applications such as traffic volume estimation or *heavy hitters* detection. However, with this method small flows are underrepresented, if noticed at all. Several studies have shown that packet sampling is not the most adequate solution for some fine-grained monitoring applications [4].

In the light of the above, we present a monitoring solution for OpenFlow which implements flow sampling. As in NetFlow/IPFIX, for each flow sampled, we maintain a flow entry in the switch which records the duration, packet and bytes counts. We use timeouts to define when these records are going to expire and, therefore, being reported to the controller. We implement flow sampling because it is easier to provide without requiring modifications to the OpenFlow specification, although we also plan to provide a packet sampling implementation in a future work.

A similar approach was previously used in [5], where they use the measurement features of OpenFlow to maintain per-flow statistics in the switches and assess the accuracy of the counters and timeouts. However, their approach is not scalable as it requires to install an entry in the flow tables for every single flow observed in the traffic, it assumes that all rules have been deployed proactively for every flow that will be observed in the network, and it does not address the problem of how monitoring rules interfere with the rest of rules installed in the switch (e.g., forwarding rules). In contrast, our contribution is the design of a complete flow monitoring solution that performs flow sampling to address scalability issues and which is transparent for the operation of other network tasks. In more detail, it has the following novel features:

Scalable: We address the scalabity issue in two different dimensions: (i) to alleviate the overhead for the controller

[1]Interpreting a flow as a set of packets sharing the same IP 5-tuple {src_IP, dst_IP, src_port, dst_port, protocol}

and (ii) to reduce the number of entries required in the flow tables of the switches. To these end, we designed two sampling methods which depend on the OpenFlow features available in current off-the-shelf switches. We remark that our methods only require to initially install some rules in the switch which will operate autonomously to discriminate (pseudo) randomly the traffic to be sampled. To the best of our knowledge, there are no solutions in line with this approach. For example, iSTAMP [2] performs a flow-based sampling technique where they make use of a multi-armed-bandit algorithm to "stamp" the most informative flows and maintain particular entries to record per-flow metrics. However, this solution specifically addresses the detection of particular flows like *heavy hitters*, while our solution provides a generic dataset of the flows in the network. Likewise, iSTAMP needs to perform periodically a training phase. It means that it is not autonomous as our system.

Fully compliant with OpenFlow: Our monitoring system implements flow sampling using only native features present since OpenFlow 1.1.0. This makes our proposal more pragmatic and realistic for current SDN deployments, which strongly rely on OpenFlow. Furthermore, for backwards compatibility, we also propose a less effective monitoring scheme that is compatible with OpenFlow 1.0.0, further increasing the targets that can benefit from our solution. Additionally, we could check there are many SDN switches (e.g., some models of HP or NEC) which do not implement NetFlow, so our solution would be a good alternative for these devices, since it provides reports with flow-level statistics as in NetFlow. We found in the literature some monitoring proposals for SDN that rely on different protocols than OpenFlow. For instance, OpenSample [6] performs traffic sampling using sFlow, which is more commonly present than NetFlow in current SDN switches. However, we consider sFlow has a high resource consumption as it sends every sampled packet to an external collector and maintains there the statistics. In contrast, our system maintains the statistics in the switch. Alternatively, some authors suggest to make use of different architectures specifically designed for monitoring tasks. For example, in [7], they propose using OpenSketch, where some sketches can be defined and dynamically loaded to perform different measurement tasks. However, in favor of our proposal, some works like [8] highlight the importance of making an OpenFlow compatible monitoring solution, as it is cheaper to implement and does not require standardization by a larger community. Note that despite the advances in the OpenFlow standard (version 1.5.1 at the time of this writing), the protocol does not provide direct support for flow sampling yet.

Transparent: Our system can be interpreted as an additional module which does not affect the correct operation of other modules performing other network functions (e.g., forwarding). To ensure this, we make use of the pipeline processing feature with multiple tables of OpenFlow. It takes a similar approach to Omniscient [9], where they propose using separate rules for monitoring specific flows tagged by end-hosts and store them in a separate OpenFlow table.

Asynchronous collection of flow statistics: Our system collects and aggregates packets directly in the switch, and retrieves flow statistics when the flow expires (either by an idle or hard timeout). In FlowSense [10], they propose the same mechanism to retrieve statistics for the entries in the switches to estimate per-flow link utilization. The problem of their solution is that the statistics of flows with large timeouts are retrieved after too long. It makes obtaining accurate measurements unfeasible in environments with highly fluctuating traffic. In our solution, as our module is completely decoupled from others, we can define the most adequate timeouts to obtain accurate measurements. Our solution can also include mechanisms to conveniently select the timeouts, such as those proposed in PayLess [11] or OpenNetMon [8], where they design adaptive schedule algorithms to collect the statistics.

The remainder of this paper is structured as follows: Firstly, in Section II, we provide an OpenFlow overview focusing on the features and messages involved in our solution. Section III defines our monitoring system and the sampling methods proposed. In Section IV, we evaluate our monitoring system in a testbed with Open vSwitch [12] and an implementation within OpenDaylight [13]. Here, we include an analysis of the accuracy of the sampling methods proposed and an evaluation of the overhead contribution, both with real-world traffic traces. Lastly, in Section V we conclude and mention some aspects for future works.

II. OpenFlow background

Nowadays, there is a growing trend among vendors to adopt OpenFlow for their switches in two different ways. Some of them are opting for OpenFlow-only devices, while others offer hybrid switches, where both traditional network protocols and OpenFlow coexist. At the moment, it is quite unusual to find commodity switches with higher support than OpenFlow 1.3.0.

In this section, we particularly focus on OpenFlow 1.1.0 specification, since it is the first version fully compatible with our solution. This is because from this version it is possible to make use of multiple tables, which enable us to decouple our monitoring system from others. However, we propose an alternative solution with some limitations for switches with OpenFlow 1.0.0 support (more details will be explained in Section III-B). It is also worth mentioning that everything described for our solution can be applied to IPv6 traffic from OpenFlow 1.2.0 onwards, since previous versions have only support for IPv4.

Regarding the monitoring solution proposed in this paper, we provide below a summary of the principal elements and messages involved.

A. Multiple flow tables and groups

Multiple flow tables and groups are both available from OpenFlow 1.1.0. The support of multiple tables enables to decouple the sets of entries of modules with different network functions operating in different tables.

Packets begin their processing pipeline in the first table of the device and can be directed to other tables. In this way, as it

goes through the pipeline, a packet can both execute an action and continue the processing in the next table or accumulate the actions and apply them at the end of the pipeline. In order to resolve possible conflicts between overlapping rules in the same flow table, each entry has a priority field.

Groups are abstractions which allow to represent a set of actions for all packets matching an entry in a flow table. Each group table contains a number of buckets which, in turn, are composed by a set of actions. Therefore, if a bucket is selected, all its actions will be applied to the packet. There are four different mechanisms to select the buckets applied to a packet reaching the group table: I) All (e.g., for multicast), II) Select (e.g., for multipath), III) Indirect and IV) Fast Failover (e.g., to use first live port). Our solution leverages the *select* mechanism for the hash-based method described in Section III-A. In a group of type *select*, packets are processed by a single bucket and so, only actions within the selected bucket are applied. This bucket selection depends on a selection algorithm (external to the OpenFlow specification) implemented in the switch which should perform equal or weighted load sharing among buckets.

B. Adding new flow entries and groups

When a packet matches an entry in a flow table with an action *output to controller*, a portion of this packet is encapsulated in an OFPT_PACKET_IN message and forwarded to the controller. Once the packet has been processed, the controller may send an OFPT_FLOW_MOD message to the switch to install a new flow entry with a set of instructions to be applied for the subsequent packets matching it. That is the way to add reactively new flow entries with OpenFlow. When adding a new flow entry, it is possible to set two timeouts (idle and hard) for that particular entry to define when it is going to be removed from the switch. The idle timeout defines the maximum time interval between two consecutive packets matching this entry, while the hard timeout is the maximum lifetime since the entry was installed.

In order to add a new group, the controller may send an OFPT_GROUP_MOD message to the switch. This message defines the type of group (all, select, indirect or fast failover), a set of buckets with their correspondent actions set and an unique identifier (32 bits) for this group. We should remark that a group table does not contain match fields, but only actions within buckets which may be applied for packets directed to this group. In order to forward packets to a group table, it is necessary to add an entry in a flow table (with match fields) defining an action of type OFPAT_GROUP. This action must include the unique identifier of the group.

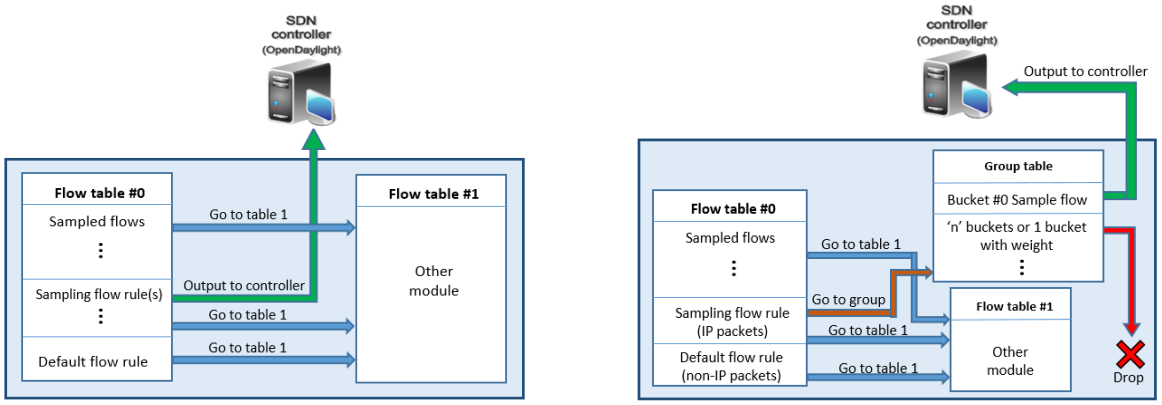
C. Statistics collection

To collect flow measurements, two different approaches deserve to be highlighted. On the one hand, pull-based mechanisms consist of making active measurements, i.e., sending queries (OFPT_MULTIPART_REQUEST message) to the switch for the desired flows. The switch will respond with an OFPT_MULTIPART_REPLY message with a summary of the flow (duration in seconds and nanoseconds, packet count and bytes count). On the other hand, push-based mechanisms consist of collecting measurements asynchronously. In this case, when adding a new flow entry, idle and/or hard timeouts are defined. Then, when a flow entry is evicted, the switch sends to the controller an OFPT_FLOW_REMOVED message with the flow statistics. This message also informs with flags that indicate if the expiration was caused by either the idle or the hard timeout. To receive asynchronously this message, when adding a new flow, the controller has to explicitly note it in the OFPT_FLOW_MOD message by marking the flag OFPFF_SEND_FLOW_REM.

III. Monitoring system

Our system fully relies on the OpenFlow specification to obtain flow measurements similar to those of NetFlow/IPFIX in traditional networks. This is not new in SDN, since some works, such as [5], used a similar approach earlier. However, to the best of our knowledge, no previous works proposed OpenFlow-based methods to implement traffic sampling and provide reports in a NetFlow/IPFIX style, i.e., randomly sampling the traffic and maintaining per-flow statistics in separated records, which are finally reported to a collector. Since we are aware that OpenFlow has many features that are classified as "*optional*" in the specification, we designed two different sampling methods with different levels of requirements of features available in the switch. These methods, in summary, consist of installing a set of entries in the switch which allow us to discriminate directly the traffic to be sampled. Thus, we only send the first packets of those flows to be monitored and the controller is in charge of installing reactively specific flow entries to maintain the flow measurements. Since OpenFlow switches are capable of communicating to the controller the features available, it is possible to decide the method to be used separately for each switch depending on its capabilities. We did not design any method for packet sampling since we found it excessively complex to implement with the current OpenFlow support, although we plan to implement it as future work.

Before showing the details of each method, we describe the generic structure of OpenFlow tables in our system, which is illustrated in Fig.1a. In both methods proposed, the monitoring system operates in the first table of the switch, where the pipeline process for incoming packets starts. In this way, our system installs in this table some entries to sample the traffic and maintains records for monitored flows. All the entries in the first table have at least one instruction to direct the packets to another table, where other modules can install entries with different purposes (e.g., forwarding). Focusing on the table where our system operates, three different blocks of entries can be differentiated by their priority field. There is a first block of flow level (5-tuple) entries that act as flow records. Then, a block of entries with lower priority defines the packets to be sampled. And lastly, we add a default entry with the lowest priority which simply directs to the next table the packets that did not match any previous entries. In this way, the key point



(a) Sampling based on IP suffixes

(b) Sampling based on hash function

Fig. 1. Scheme of OpenFlow tables and entries of the monitoring system.

of our system resides on the second block of entries, where the methods described below establish different rules to define which packets are sampled. The operation mode when a new packet arrives to the switch is to check firstly if it is already in one of the per-flow monitoring entries. If it matches any of these entries, the packets and bytes counters are updated and the packet is directed to the next table. If not, it goes through the block of entries that define whether it has to be sampled or not. If it matches one of these, then the packet is forwarded to the next table and to the controller (Packet In message) to add a specific entry in the first block to sample subsequent packets of this flow. Finally, if the packet does not match any of the previous rules, it is simply directed to the next table.

A. *Proposed sampling methods*

We present here the two methods devised for our monitoring solution and discuss the OpenFlow features required for each of them. One is based on hash functions, which performs flow sampling very accurately, and the other one, based on IP suffixes, is proposed as a fallback mechanism when it is not possible to implement the previous one. We assume that the switches have support for OpenFlow 1.1.0 and later versions so, they have at least support for multiple tables. However, in Section III-B, we make some comments about how to implement an alternative solution with OpenFlow 1.0.0. Our selection mechanisms for the packets are covered by the Packet Sampling (PSAMP) Protocol Specification [14], which is compatible with the IPFIX protocol specification. According to the PSAMP terminology, our first sampling method can be classified as property match filtering, where a packet is selected if specific fields within the packet are equal to a predefined set of values. While the second is of type hash-based filtering.

1) Sampling based on IP suffixes: This method is based on performing traffic sampling based on IP address matches. To achieve it, the controller adds proactively one entry with match fields for particular IP address ranges. A similar approach was also used in [15] for load balancing client traffic with OpenFlow. Typically, in traditional routing the matching of IP addresses is based on IP prefixes. In contrast, we consider to apply a mask which checks the last n bits of the IPs, i.e., we sample flows with specific IP suffixes. In this way, we sample a more representative set of flows, since we monitor flows from different subnets (IP prefixes) in the network. In order to implement this, it is only necessary a wildcarded entry that filters the IP suffixes desired for source or destination addresses, or combinations of them. To control the number of flows to be sampled, we make a rough consideration that, in average, flows are homogeneously distributed along the whole IP range (we later analyze this assumption with real traffic in Section IV-A). As a consequence, for each bit fixed in the mask, the number of flows sampled will be divided by two with respect to the total number of flows arriving to the switch. We are aware that typically there are some IPs that generate much more traffic than others, but this method somehow allow to control the number of flows to be monitored. Furthermore, if we consider pairs of IPs for the selection, instead of individual IPs, we can control better this effect. In this case, if we sample an IP address of a host which generates a large number of flows, only those flows which match both source and destination IP suffixes are sampled. Generically, our sampling rate can be defined by the following expression:

$$sampling\ rate = \frac{1}{2^m \cdot 2^n} \tag{1}$$

Where 'm' is the number of bits checked for the source IP suffix and 'n' the number of bits checked for the destination IP suffix.

This method is similar to host-based (or host-pair-based) sampling, as we are using IP addresses to select the packets to be sampled. However, host-based schemes typically provide statistics of aggregated traffic for individual or group of hosts. In contrast, we sample the traffic by single or pairs of IP suffixes, but provide individual statistics at a flow granularity level. Moreover, to avoid bias in the selection, the IP suffixes can be periodically changed by simply replacing the sampling rule(s) in the OpenFlow table.

To implement this method, the only optional requirement of OpenFlow is the support of arbitrary masks for IP to check suffixes, since there are some switches which only support prefix masks for IP. We also present and evaluate in the technical report version of this paper [16], an alternative method based on matching on port numbers for those switches that do not support IP masks with suffixes, but this method requires a larger number of entries to sample the traffic.

2) Hash-based flow sampling: This method consists of computing a hash function on the traditional 5-tuple fields of the packet header and selecting it if the hash value falls in a particular range. In Fig.1b, we can see the tables structure of this method. In this case, all IP packets are directed to the next table as well as to a group table where only one bucket sends the packet to the controller to monitor the flow, other buckets drop the packet. To control the sampling rate, we can select a weight for each bucket. This method much better controls the sampling rate, since we can assume that a hash function is homogeneous along all its range for all the flows in the switch.

This method, in contrast to the previous one, accurately follows the definition of flow sampling, i.e., sample the packets of a subset of flows selected with some probability [17].

The requirements for this method are to support group tables with *select* buckets and to have an accurate algorithm in the switch to balance the load properly among buckets.

B. Modularization of the system

Our solution leverages the support of multiple tables to isolate its operation from other modules performing other network functions. Thus, we can see our monitoring system as an independent module in the controller which does not interfere with other modules operating in other tables. In the controller we can filter and process the Packet In messages triggered by entries of our module, since these messages contain the table Id of the entry which forwarded the packet to the controller. Additionally, our system can be integrated in a network using a hypervisor (e.g., CoVisor [18]) to run network modules in a distributed manner in different controllers. Nevertheless, we propose an alternative for those switches with OpenFlow 1.0.0 support, where only one table can be used. Since this version does not support group tables, only the first method, based on matches of IP suffixes, can be implemented. In that way, it is feasible to install the monitoring entries by combining them with the correspondent actions of other modules at the expense of loosing the decoupling of our monitoring system.

C. Statistics retrieval

Our system envisions a push-based approach to retrieve statistics. Given that it uses specific entries, we can selectively choose the timeouts to retrieve the statistics. As a result, we overcome the issue of other push-based solutions such as FlowSense [10], where flows with large timeouts are collected after too long a time decreasing the accuracy of the measurements.

IV. Experimental evaluation

We have implemented our monitoring solution within OpenDaylight [13], operating jointly with the "L2Switch" module that it includes for layer 2 forwarding.

We conducted experiments in a small testbed with an Open vSwitch [12], a host (VM) which injects traffic into the switch and another host which acts as a sink for all the traffic forwarded. All the experiments make use of real-world traffic from three different network scenarios. One trace corresponds to a large Spanish university (labeled as "UNIVERSITY"), and the others correspond to two different ISP networks (MAWI [19] and CAIDA [20]). These traces were filtered to keep only the TCP and UDP traffic. In Table I there is a detailed description of each trace.

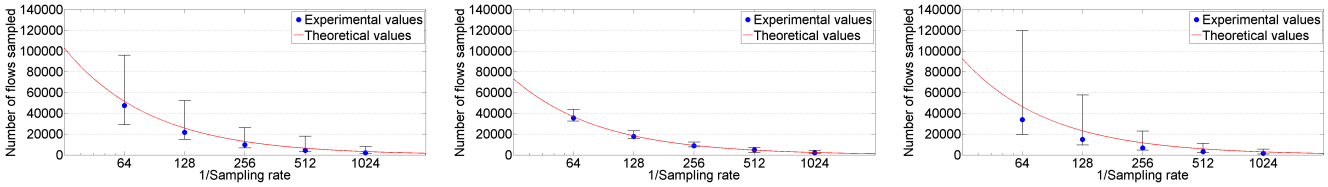
Trace dataset	# of flows	# of packets	Description
UNIVERSITY 25th November 2016	2,972,880 (total flows) 2,349,677 (TCP flows) 623,203 (UDP flows)	75,585,871	10 Gbps downstream access link of a large Spanish university, which connects about 25 faculties and 40 departments (geographically distributed in 10 campuses) to the Internet through the Spanish Research and Education network (RedIRIS). Average traffic rate: 2.41 Gbps
MAWI [19] 15th July 2016	3,299,166 (total flows) 2,653,150 (TCP flows) 646,016 (UDP flows)	54,270,059	1 Gbps transit link of WIDE network to the upstream ISP. Trace from the samplepoint-F. Average traffic rate: 507 Mbps
CAIDA[20] 18th February 2016	2,353,413 (total flows) 1,992,983 (TCP flows) 360,430 (UDP flows)	51,368,574	This trace corresponds to a 10 Gbps backbone link of a Tier1 ISP (direction A - from Seattle to Chicago). Average traffic rate: 2.9 Gbps

TABLE I
SUMMARY OF THE REAL-WORLD TRAFFIC TRACES USED.

A. Accuracy of the proposed sampling methods

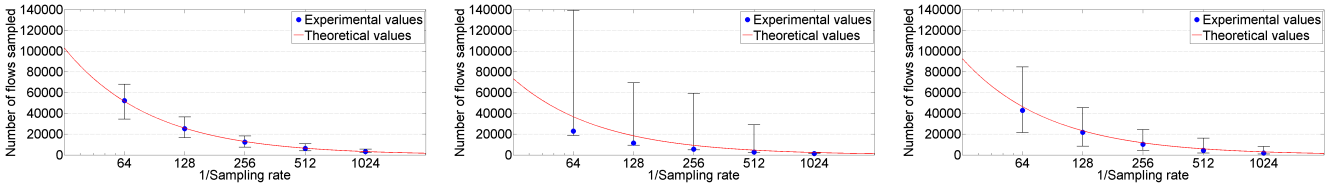
We conducted experiments to assess if the sampling rate is applied properly and if the selection of flows is random enough when using the proposed sampling methods. All our experiments were separately done for the MAWI, CAIDA and UNIVERSITY traces described in Table I and repeated applying sampling rates of 1/64, 1/128, 1/256, 1/512 and 1/1024. For the method based on IP suffixes, we considered two different modalities: matching only a source IP suffix, or matching both source and destination IP suffixes. For each of these modalities, with a particular trace, and a specific sampling rate, we performed 500 experiments selecting randomly IP suffixes. We got these results by means of simulations and validated in our testbed at least three experiments for each sampling rate. For the hash-based method, since it is based on a deterministic selection function, we only conducted one experiment in our testbed for each case.

To analyze the accuracy in the application of the sampling rate, we evaluate the number of flows sampled by our methods and compare it with the theoretical number of flows if we used a perfectly random selection function. We show in Fig. 2, the results for the method based only on source IP suffixes for the three traces described in Table I. These plots display the median value of the number of flows sampled for the experiments conducted in relation to the sampling rate applied. The experimental values include bars which show the interval between the 5^{th} and the 95^{th} percentiles of the total 500



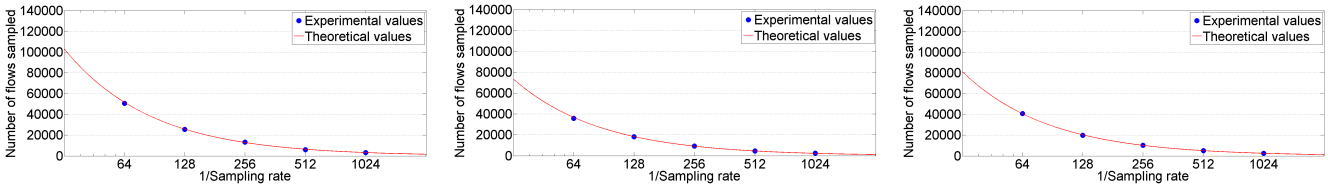
(a) Source IP suffixes - MAWI (b) Source IP suffixes - CAIDA (c) Source IP suffixes - UNIVERSITY

Fig. 2. Evaluation of sampling rate for methods based on source IP suffixes.



(a) Pair of IP suffixes - MAWI (b) Pair of IP suffixes - CAIDA (c) Pair of IP suffixes - UNIVERSITY

Fig. 3. Evaluation of sampling rate for methods based on pairs of IP suffixes.



(a) Hash-based - MAWI (b) Hash-based - CAIDA (c) Hash-based - UNIVERSITY

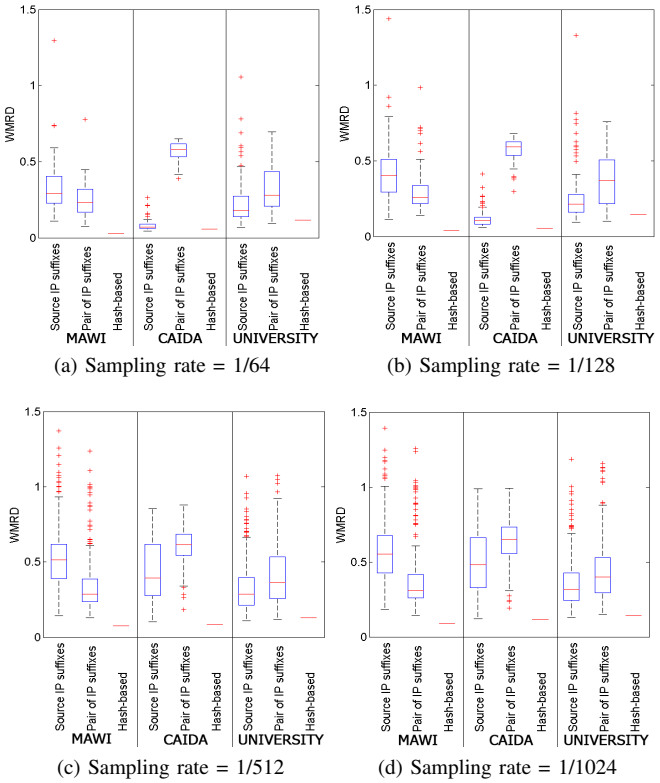
Fig. 4. Evaluation of sampling rate for the hash-based method.

measurements obtained for each case. Likewise, in Fig. 3, we show the same results for the case that considers pairs of source and destination IP suffixes. Given these results, we can see that the median values obtained are quite close to the theoretical values, i.e., in the average case these methods apply properly the sampling rate established. However, we can see there is a high variability among experiments. This means that, depending on the IP suffixes selected, we can over- or under-sample. In order to validate the implementation of this method, we randomized the IPs of the flows of our traces to have a homogeneous distribution and applied the method. Thus, we could observe that it achieved a number of flows very close to the theoretical values and a very low variability among experiments (these results are detailed in the technical report version of this paper [16]).

Next, we evaluate the hash-based sampling method making use of the load balancing algorithm for group tables included in Open vSwitch. The results, in Fig. 4, show that this method considerably outperforms the previous one in terms of control of the sampling rate. Not only it samples a number of flows very close to the ideal one, but also it does not experience any variability among experiments as it is based on a deterministic selection function. Furthermore, it achieves good results for the three different traces, which indicates that it is a robust and generalizable method to be implemented in any network independently of the nature of its traffic.

In order to evaluate the randomness in the selection of our sampling methods, we compare our results with those obtained with a perfect implementation of flow sampling, with a completely random selection process. Thus, if our implementation is close to a perfect flow sampling implementation, the flow size distribution (FSD) should remain unchanged after applying the sampling, i.e., the distribution of the flow sizes (in number of packets) must be very similar for the original and the sampled data sets. We acknowledge that this property is not completely preserved for the IP-based method, but we follow this approach to measure how random is the flow selection of this method and compare it with the hash-based method.

We quantify the randomness of the sampling method by calculating the difference between the FSDs of the original and the sampled traffic. For this purpose, we use the *Weighted Mean Relative Difference* (WMRD) metric proposed in [21]. Thus, a small WMRD means that the flow selection is quite random. In Fig. 5 we present boxplots with the results of our proposed methods. For the sake of brevity, we do not show the results for a sampling rate of 1/256, since they are very similar to those displayed (all these results are available in the technical report version of this paper [16]). We can observe that the results are in line with the above results about the accuracy controlling the sampling rate. The method which shows better results is the hash-based one. Additionally, for the methods based on IP suffixes, we see that for the MAWI



(a) Sampling rate = 1/64 (b) Sampling rate = 1/128

(c) Sampling rate = 1/512 (d) Sampling rate = 1/1024

Fig. 5. Weighted Mean Relative Difference (WMRD) between FSDs.

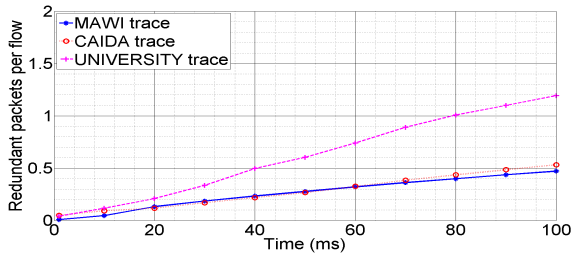
trace, the method based on pairs of IP suffixes achieves a more random flow subset. While for the CAIDA and UNIVERSITY traces, the method based on source IP suffixes behaves better.

Note that we chose the FSD to compare the randomness of the two flow selection methods, because the FSD is known to be robust against flow sampling. As future work, we also plan to analyze how the randomness in the flow selection process affects other statistics commonly extracted from Sampled Netflow data, such as application mixes, port distributions or bandwidth utilization per customer.

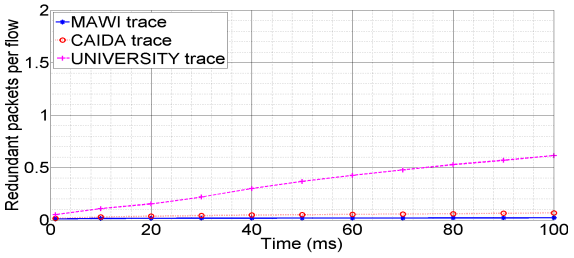
B. Evaluation of the overhead

An inherent problem in OpenFlow is that, when we install flows reactively, packets belonging to the same flow are sent to the controller until a specific entry for them is installed in the switch. This is a common problem to any system that works at flow-level granularities. As a consequence, in our system we can receive in the controller more than one packet for each flow to be sampled. Specifically this occurs during the interval of time between the reception of the first packet of a flow in the switch, and the time when a specific entry for this flow is installed in the switch. This time interval is mainly the result of the following factors: (i) the time needed by the switch to process an incoming packet of a *new* flow to be sampled and forward it to the controller, (ii) *Round-Trip Time* (RTT) between the switch and the controller, (iii) the time for the controller to process the Packet In and send to the switch the order to install a new flow entry, and (iv) the time in the switch to install the new flow entry. The first and fourth factors depend on the processing power of the switch. The RTT depends on some aspects like the distance between the switch and the controller or the capacity and utilization of the control link that connects them. The second factor depends on the processing power and the workload of the controller and, of course, its availability.

In order to analyze all these different bottlenecks in a single metric, we measure the amount of redundant packets of the same flow that the controller processes. That is, the number of packets of a sampled flow that are sent to the controller before the switch can install a rule to monitor that specific flow. We consider a scenario with a range from 1 ms to 100 ms for the elapsed time to install a new flow entry. This time includes all the factors described earlier, from (i) to (iv). As a reference, in [22] they observe a median value of 34.1 ms for the time interval to send the OFPT_FLOW_MOD message to add a new flow entry with the ONOS controller in an emulated network with 206 software switches and 416 links. Thus, we simulate this range of time values for the three traces described in Table I and analyze the timestamps of the packets to calculate, for each flow, how many packets are within this interval and, thereby, would be sent to the controller. We analyze separately the overhead for TCP and UDP, as their results may differ due to their different traffic patterns. We show the results in Fig. 6. As we can see, the average number of redundant packets varies from less than 0.2 packets for delays below 20 ms, to approximately 1.2 packets per flow for an elapsed time of 100 ms for TCP traffic.



(a) TCP traffic

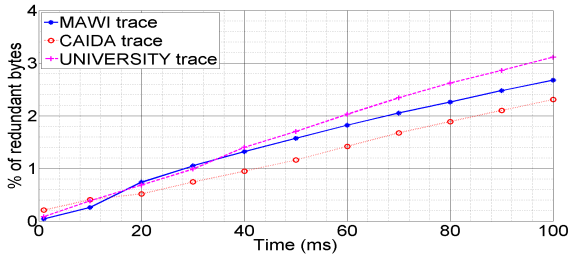


(b) UDP traffic

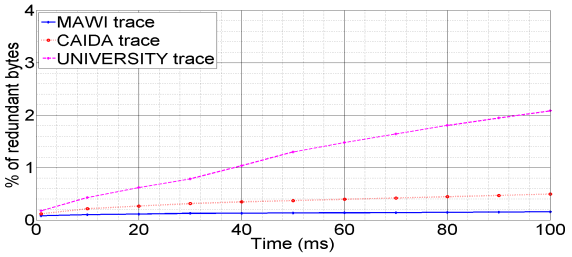
Fig. 6. Average number of redundant packets per flow.

Likewise, in Fig. 7 we show the results in terms of average percentage of redundant bytes sent to the controller. That way, the percentage of redundant bytes ranges from less than 0.8% for elapsed times below 20 ms to 3.1% in the worst case with an elapsed time of 100 ms and TCP traffic. These results show that the amount of redundant traffic sent to the controller

is significantly smaller than if we implemented the trivial approach of forwarding all the traffic to the controller or a NetFlow probe and not installing in the switch specific entries to process subsequent packets and maintain per-flow statistics. The best case is for elephant flows, as the amount of packets sent to the controller at the beginning of the flow is very low in proportion to the total amount of traffic they carry.



(a) TCP traffic



(b) UDP traffic

Fig. 7. Percentage of redundant bytes.

These results also reflect that, for the UDP traffic, the number of redundant packets and bytes per flow is significantly smaller than for TCP flows. Among other reasons, this is due to the fact that typically many UDP flows are single-packet (e. g., DNS requests or responses). In the UNIVERSITY trace we could notice that there were more UDP flows with a larger number of packets, as it is reflected in Figs. 6b and 7b.

From these results, it is possible to infer the CPU cost of running our monitoring system in a SDN controller, as the processing cost per packet can be considered constant. In particular, the controller only needs to maintain a hash table to keep track of those packets sent to the controller and thus not accounted for in switch (i.e., redundant packets shown in Fig. 6). As future work, we plan to further analyze the resource requirements in the controller (e.g., processing power, buffer size) and the control infrastructure to ensure that none of the sampled packets are dropped and, thereby, are accounted for in the controller.

As for the memory overhead in the switch, we implement sampling methods that provide mechanisms to control the number of entries installed. With our solution it is necessary to maintain a flow entry for each individual sampled flow. Thus, there are three main factors which determine the amount of memory necessary in the switch to maintain the statistics: (i) the rate of new incoming flows (traffic matching different 5-tuples) per time unit, (ii) the sampling rate selected, and (iii) the idle and hard timeouts selected for the entries to be maintained. The first factor depends specifically on the nature of the network traffic, i.e., the rate of new flows arriving to the switch (e.g., flows/s). It is a parameter fixed by the network environment where we operate. However, as in NetFlow, the sampling rate and the timeouts (idle and hard) are static configurable parameters and the selection of these parameters affects the memory requirements in the switch. In this way, with (2) we can roughly estimate the average amount of concurrent flow entries maintained in the switch.

$$\begin{aligned} Avg.\,entries = N_{flows} \cdot sampling\,rate \cdot E[t_{out}] \\ sampling\,rate \in (0,1] \quad t_{out} \in [t_{idle}, t_{hard}] \end{aligned} \tag{2}$$

Where "N_{flows}" denotes the average number of new incoming flows per time unit, "sampling rate" is the ratio of flows we expect to monitor, and $E[t_{out}]$ corresponds to the average time that a flow entry is maintained in the switch.

In order to configure a specific sampling rate, for the method based on IP suffixes we can set the number of bits to be checked for the IP suffix(es) according to (1). Likewise, for the hash-based method, we can set the proportion of flows to be sampled by configuring the weights of the buckets. Regarding the timeouts, the controller can set the values of the idle and hard timeouts when adding a new flow entry in the switch to record the statistics (in the OFPT_FLOW_MOD message).

To conclude this section, we propose some different scenarios and estimate the average number of concurrent flow entries to be maintained in the switch. The purpose of this analysis is to have a picture of the approximate memory contribution of the monitoring solution proposed in this paper. To this end, we rely on (2). In our scenarios we consider the three different real-world traces described in Table I. Thus, to calculate "N_{flows}" for each trace, we divide their respective total number of flows (only TCP and UDP) by their duration. Furthermore, we consider two different sampling rates, 1/128 and 1/1024. For the configuration of the timeouts, we envision a typical scenario using the default values defined in NetFlow: 15 seconds for the idle timeout and 30 minutes (1800 seconds) for the hard timeout. Regarding the average time that a flow remains in the switch ($E[t_{out}]$), we know that it ranges from the idle timeout to the hard timeout. In this way, we consider these two extreme values and some others in the middle. The case with the lowest memory consumption will be when $E[t_{out}]$ is equal to the idle timeout, and the case with the highest consumption, when $E[t_{out}]$ is equal to the hard timeout. The amount of memory for each flow entry strongly depends on the OpenFlow version implemented in the switch. The total amount of memory of a flow entry is the sum of the memory of its match fields, its action fields and its counters. For example, in OpenFlow 1.0 there are only 12 different match fields (269 bits approximately), while in OpenFlow 1.3 there are 40 different match fields (1,261 bits).

Table II summarizes the results for all the cases described above. As a reference, in [23] they noted that modern OpenFlow switches have support for 64k to 512k flow entries. To these flow entries estimated, we must add the additional

Sampling rate	Trace dataset	N_{flows} (flows/s)	Avg. number of flow entries						
			E[t]=15 s	E[t]=60 s	E[t]=300 s	E[t]=600 s	E[t]=900 s	E[t]=1,200 s	E[t]=1,800 s
1/128	UNIVERSITY	9,916	1,162	4,648	23,241	46,481	69,722	92,963	139,444
	MAWI	3,665	429	1,718	8,590	17,180	25,770	34,359	51,539
	CAIDA	21,672	2,540	10,159	50,794	101,588	152,381	203,175	304,763
1/1024	UNIVERSITY	9,916	145	581	2,905	5,810	8,715	11,620	17,430
	MAWI	3,665	54	215	1,074	2,147	3,221	4,295	6,442
	CAIDA	21,672	317	1,270	6,349	12,698	19,048	25,397	38,095

TABLE II
ESTIMATION OF THE AVERAGE FLOW ENTRIES USED IN THE SWITCH.

amount of memory of the implementation of the sampling methods described in Section III-A. For both methods, the switch must allocate an additional table to maintain the sampled flows as well as the entries which determine the flows to be sampled. For the method based on IPs, it uses an additional wildcarded flow entry which determines the IP suffix(es) to be sampled. For the hash-based method, it uses an additional entry to redirect the packets to a group table, as well as the group table with its respective buckets. We don't provide an estimation of this memory contribution since we consider it is too dependent on the OpenFlow implementation in the switch. Nevertheless, we assume that this amount of memory is negligible compared to the amount of memory allocated for the entries that record the statistics of the sampled flows.

V. Conclusions and future work

We presented a flow monitoring solution for OpenFlow which provides reports like in NetFlow/IPFIX. In order to reduce the overhead in the controller and the number of entries required in the switch, we proposed two traffic sampling methods that can be implemented in current switches without requiring any modification to the OpenFlow specification. We implemented them in OpenDaylight and evaluated their accuracy and overhead in a testbed with real traffic. As future work, we plan to extend the analysis of the randomness of our sampling methods as well as the overhead evaluation, design smarter algorithms to retrieve the statistics more accurately and implement an OpenFlow compliant packet sampling method, although we find it more challenging.

Acknowledgement

This work was supported by the European Union's H2020 SME Instrument Phase 2 project "SDN-Polygraph" (grant agreement n° 726763), the Spanish Ministry of Economy and Competitiveness and EU FEDER under grant TEC2014-59583-C2-2-R (SUNSET project), and by the Catalan Government (ref. 2014SGR-1427).

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, p. 69, 2008.

[2] M. Malboubi, L. Wang, C. N. Chuah, and P. Sharma, "Intelligent SDN based traffic (de)Aggregation and Measurement Paradigm (iSTAMP)," *Proceedings - IEEE INFOCOM*, pp. 934–942, 2014.

[3] B. Claise, "NetFlow Services Export Version 9 Status," pp. 1–33, 2004.

[4] V. Sekar, M. K. Reiter, and H. Zhang, "Revisiting the case for a minimalist approach for network flow monitoring," *Proceedings of the IMC*, p. 328, 2010.

[5] L. Hendriks, R. D. O. Schmidt, R. Sadre, J. A. Bezerra, and A. Pras, "Assessing the Quality of Flow Measurements from OpenFlow Devices," *8th International Workshop on Traffic Monitoring and Analysis (TMA)*, 2016.

[6] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, "OpenSample: A low-latency, sampling-based measurement platform for commodity SDN," *Proceedings - International Conference on Distributed Computing Systems*, pp. 228–237, 2014.

[7] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," *Networked Systems Design and Implementation, (NSDI)*, vol. 13, pp. 29–42, 2013.

[8] N. L. M. Van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network monitoring in OpenFlow software-defined networks," *IEEE/IFIP NOMS*, 2014.

[9] D. A. Popescu and A. W. Moore, "Omniscient : Towards realizing near real-time data center network traffic maps," *CoNEXT Student Workshop*, 2015.

[10] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha, "FlowSense: Monitoring network utilization with zero measurement cost," *Lecture Notes in Computer Science*, vol. 7799 LNCS, pp. 31–41, 2013.

[11] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "PayLess: A low cost network monitoring framework for Software Defined Networks," *IEEE NOMS*, pp. 1–9, 2014.

[12] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending Networking into the Virtualization Layer," *8th ACM Workshop on Hot Topics inNetworks*, vol. VIII, p. 6, 2009.

[13] "The OpenDaylight platform," http://www.opendaylight.org/.

[14] B. Claise, "Packet sampling (PSAMP) protocol specifications," 2009.

[15] R. Wang, D. Butnariu, and J. Rexford, "OpenFlow-Based Server Load Balancing Gone Wild Into the Wild," *Proceedings of the Hot-ICE*, p. 12, 2011.

[16] J. Suárez-Varela and P. Barlet-Ros, "Reinventing NetFlow for OpenFlow Software-Defined Networks (Technical report)," *arXiv preprint arXiv:1702.06803*, 2017.

[17] N. Hohn and D. Veitch, "Inverting Sampled Traffic," *Proceedings of the IMC*, pp. 222–233, 2003.

[18] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: A Compositional Hypervisor for Software-Defined Networks," *Proceedings of Networked Systems Design and Implementation (NSDI)*, pp. 87–101, 2015.

[19] "MAWI Working Group traffic archive - [15/07/2016]," http://mawi.wide.ad.jp/mawi/.

[20] "The CAIDA UCSD Anonymized Internet Traces 2016 - [18/02/2016]," http://www.caida.org/data/passive/passive_2016_dataset.xml.

[21] N. Duffield, C. Lund, and M. Thorup, "Estimating flow distributions from sampled flow statistics," *IEEE/ACM Transactions on Networking*, vol. 13, no. 5, pp. 933–946, 2005.

[22] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, and B. Lantz, "ONOS: towards an open, distributed SDN OS," *Proceedings of HotSDN*, pp. 1–6, 2014.

[23] "Can OpenFlow scale?" https://www.sdxcentral.com/articles/contributed/openflow-sdn/2013/06/, accessed: 2017-06-06.

Bibliography

[1] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2014.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, p. 69, 2008.

[3] M. Malboubi, L. Wang, C. N. Chuah, and P. Sharma, "Intelligent SDN based traffic (de)Aggregation and Measurement Paradigm (iSTAMP)," *in proceedings of the IEEE INFOCOM*, pp. 934–942, 2014.

[4] "Introduction to Cisco IOS NetFlow - A technical overview," http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html, accessed: 2017-05-10.

[5] "OpenDaylight project," https://www.opendaylight.org/, accessed: 2017-04-16.

[6] "OpenDaylight - Project members," https://www.opendaylight.org/support/members, accessed: 2017-06-06.

[7] L. Hendriks, R. D. O. Schmidt, R. Sadre, J. A. Bezerra, and A. Pras, "Assessing the Quality of Flow Measurements from OpenFlow Devices," *in proceedings of the 8th International Workshop on Traffic Monitoring and Analysis (TMA)*, 2016.

[8] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, "OpenSample: A low-latency, sampling-based measurement platform for commodity SDN," *in proceedings of the International Conference on Distributed Computing Systems*, pp. 228–237, 2014.

[9] "sFlow version 5 specification," http://www.sflow.org/sflow_version_5.txt, accessed: 2017-06-11.

[10] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," *in proceedings of Networked Systems Design and Implementation (NSDI)*, vol. 13, pp. 29–42, 2013.

[11] N. L. M. Van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network monitoring in OpenFlow software-defined networks," *in proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2014.

[12] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha, "Flowsense: Monitoring network utilization with zero measurement cost," *in proceedings of International Conference on Passive and Active Network Measurement (PAM)*, pp. 31–41, 2013.

[13] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "PayLess: A low cost network monitoring framework for Software Defined Networks," *in proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pp. 1–9, 2014.

[14] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and J. Van Der Merwe, "The case for separating routing from routers," in *proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, 2004, pp. 5–12.

[15] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4D approach to network control and management," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 5, pp. 41–54, 2005.

[16] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4, pp. 1–12, 2007.

[17] K. Greene, "TR10: Software-defined networking," *Technology Review (MIT)*, 2009.

[18] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[19] D. Erickson, "The beacon openflow controller," in *proceedings of the 2nd ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN)*, 2013, pp. 13–18.

[20] "Floodlight OpenFlow controller," http://www.projectfloodlight.org/, accessed: 2017-05-21+.

[21] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A distributed control platform for large-scale production networks." in *proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 10, 2010, pp. 1–6.

[22] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *et al.*, "ONOS: towards an open, distributed SDN OS," in *proceedings of the 3rd workshop on Hot topics in software defined networking (HotSDN)*, 2014, pp. 1–6.

[23] E. Ng, Z. Cai, and A. Cox, "Maestro: A system for scalable openflow control," *Rice University, Houston, TX, USA, TSEN Maestro-Techn. Rep, TR10-08*, 2010.

[24] "POX controller," https://github.com/noxrepo/pox/, accessed: 2017-06-06.

[25] "Ryu SDN framework," https://osrg.github.io/ryu/, accessed: 2017-06-06.

[26] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[27] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, "OpenTM: traffic matrix estimator for OpenFlow networks," in *proceedings of the International Conference on Passive and Active Network Measurement (PAM)*, 2010, pp. 201–210.

[28] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1999, pp. 228–241.

[29] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 351–362, 2010.

[30] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 254–265, 2011.

[31] A. S.-W. Tam, K. Xi, and H. J. Chao, "Use of devolved controllers in data center networks," in *proceedings of the IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2011, pp. 596–601.

[32] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending Networking into the Virtualization Layer," *in proceedings of the 8th ACM Workshop on Hot Topics in Networks (HotNets)*, vol. VIII, p. 6, 2009.

[33] "TCPDUMP/LIBPCAP public repository," http://www.tcpdump.org/, accessed: 2017-05-20.

[34] "The CAIDA UCSD Anonymized Internet Traces 2016 - [18/02/2016]," http://www.caida.org/data/passive/passive_2016_dataset.xml.

[35] "MAWI Working Group traffic archive - [15/07/2016]," http://mawi.wide.ad.jp/mawi/.

[36] B. Claise, A. Johnson, and J. Quittek, "Packet sampling (psamp) protocol specifications," Internet Requests for Comments, RFC Editor, RFC 5476, March 2009.

[37] R. Wang, D. Butnariu, and J. Rexford, "OpenFlow-Based Server Load Balancing Gone Wild Into the Wild," *in proceedings of the workshop on hot topics in management of internet, cloud, and enterprise networks and services (HOT-ICE)*, p. 12, 2011.

[38] J. Suárez-Varela and P. Barlet-Ros, "Reinventing NetFlow for OpenFlow Software-Defined Networks (Technical report)," *arXiv preprint arXiv:1702.06803*, 2017.

[39] N. Hohn and D. Veitch, "Inverting Sampled Traffic," *in proceedings of the Internet Measurement Conference (IMC)*, pp. 222–233, 2003.

[40] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: A Compositional Hypervisor for Software-Defined Networks," *in proceedings of the Networked Systems Design and Implementation (NSDI)*, pp. 87–101, 2015.

[41] N. Duffield, C. Lund, and M. Thorup, "Estimating flow distributions from sampled flow statistics," *IEEE/ACM Transactions on Networking*, vol. 13, no. 5, pp. 933–946, 2005.

[42] "Cisco IOS Flexible NetFlow Command Reference," https://www.cisco.com/c/en/us/td/docs/ios/fnetflow/command/reference/fnf_book/fnf_01.html, accessed: 2017-06-20.

[43] "Can OpenFlow scale?" https://www.sdxcentral.com/articles/contributed/openflow-sdn/2013/06/, accessed: 2017-06-06.