Universidade do Minho

Escola de Engenharia

Departamento de Informática

David Rodrigues Teixeira

Deploying time-based sampling techniques in Software-Defined Networking

October 2017

Universidade do Minho

Escola de Engenharia

Departamento de Informática

David Rodrigues Teixeira

Deploying time-based sampling techniques in Software-Defined Networking

Master dissertation

Master Degree in Computer Science

Dissertation supervised by

Maria Solange Pires Ferreira Rito Lima

João Marco Cardoso da Silva

October 2017

ACKNOWLEDGEMENTS

Terminada esta fase acadêmica, é momento de refletir e agradecer às pessoas que me acompanharam e ajudaram neste trajeto longo e sinuoso mas ainda assim gratificante, que certamente contribuiu para o meu enriquecimento enquanto aluno, pessoa e profissional.

Aos meus orientadores, Professora Maria Solange Pires Ferreira Rito Lima e Professor João Marco Cardoso da Silva, agradeço pelo tema proposto e pela orientação e aconselhamento que foram demonstrando para que a conclusão deste ciclo de estudos fosse possível. O conhecimento da área e espírito crítico que incutiram em mim foi sem dúvida essencial para a conclusão deste projeto.

Dedico esta dissertação aos meus pais, que com o seu amor incondicional estiveram sempre presentes para me incentivar, por depositarem confiança em mim e nas minhas capacidades e por me fazerem acreditar que é possível atingir os objetivos mesmo perante os obstáculos mais difíceis.

Para a minha irmã Catarina, pela sua personalidade e pontos de vista, que me fazem recordar que pode haver harmonia e convergência entre pessoas com características diferentes, através das vivências partilhadas.

Aos meus amigos, obrigado pela capacidade de estarem presentes, ouvir e aconselhar quando foi necessário.

Por fim, para a Catarina, pelo incentivo na conclusão da dissertação, pela persistência quando a vontade se convertia em distração, e sobretudo pelo carinho, atenção e por potenciar o que de melhor há em mim.

ABSTRACT

Today's computer networks face demanding challenges with the proliferation of services and applications requiring constant access, low latency and high throughput from network infrastructures.

The increase in the demand for this type of services requires continuous analysis and a network topology capable of adapting to the dynamic nature of applications, in order to overcome challenges such as performance, security and flexibility.

[Software-Defined Networking \(SDN\)](#) emerge as a solution to meet these challenges by using a network control plane, dissociated from the data plane, able to have a global view of the topology and act when required, depending on the variation in infrastructure congestion.

Decisions involving different activities, such as network management and performance evaluation, rely on information about the state of the network that in traditional networks involves a substantial amount of data. Traffic sampling is essential in order to provide valuable statistical data to applications and enable appropriate control and monitoring decisions to be made.

In this context, this work proposes the application of time-based sampling techniques in a [SDN](#) environment to provide network statistics at the controller level, taking into account the underlying need to establish a balance between the reliability of the data collected and the computational burden involved in the sampling process.

The results obtained emphasize that it is possible to apply these sampling techniques by using OpenFlow Group Mod messages, although packet losses can occur on the switch during periods of network congestion.

RESUMO

As redes de computadores atuais enfrentam desafios exigentes, com a proliferação de serviços e aplicações que exigem acesso constante, baixa latência e elevado fluxo de dados.

O aumento na procura deste tipo de serviços exige análise contínua e uma topologia de rede capaz de se adaptar à natureza dinâmica das aplicações, de forma a superar desafios como desempenho, segurança e flexibilidade.

As redes definidas por software (Software-Defined Networking - [SDN](#)) surgem como uma solução para corresponder a este desafio, através da utilização de uma estrutura de controlo na rede, separada do plano de dados, capaz de ter uma visão global da arquitetura e agir adequadamente consoante as necessidades, dependendo da variação na congestão da infraestrutura.

As decisões que envolvem diversas atividades, tais como gestão da rede e avaliação de desempenho, dependem de informação sobre o estado da rede que, em redes tradicionais, envolve uma quantidade substancial de dados. A amostragem de tráfego é essencial para fornecer dados estatísticos valiosos a aplicações e permitir definir decisões adequadas de controlo e monitorização.

Neste contexto, este trabalho propõe a implementação de técnicas de amostragem baseadas em tempo num ambiente [SDN](#), para disponibilizar estatísticas da rede ao nível do controlador, tendo em conta a necessidade subjacente de estabelecer um balanço entre a fiabilidade dos dados recolhidos e o peso computacional envolvido no processo de amostragem.

Os resultados obtidos enfatizam que é possível aplicar essas técnicas de amostragem utilizando mensagens OpenFlow Group Mod, embora perdas de pacotes possam ocorrer no switch em períodos de congestionamento da rede.

CONTENTS

Abstract	ii
Resumo	iii
List of Figures	vi
List of Tables	vii
List of Abbreviations	viii
1 INTRODUCTION	1
1.1 Motivation and Goals	1
1.2 Main Contributions	2
1.3 Document Layout	2
2 SDN STATE OF THE ART	4
2.1 Programmable Networks	4
2.2 SDN Architecture	6
2.3 OpenFlow	7
2.3.1 OpenFlow Switch Specification	8
2.3.2 OpenFlow Versions	9
2.4 SDN Controller	10
2.5 Virtual Switching	12
2.6 Emulation Software for SDN	14
2.7 Summary	15
3 MONITORING IMPLEMENTATION IN SDN	16
3.1 Sampling in Monitoring Systems	16
3.1.1 Systematic Sampling	18
3.1.2 Random Sampling	19
3.1.3 Adaptive Sampling	19
3.2 Existing Monitoring Tools Based on OpenFlow	20
3.2.1 sFlow	20
3.2.2 FleXam	21
3.2.3 PayLess	22
3.3 Summary	23
4 SDN SAMPLING CONTROLLER	25
4.1 Design Goals	25
4.2 System Architecture	26
4.3 Implementation Decisions	27
4.3.1 OpenFlow Messages Used in the Sampling Process	27
4.3.2 Stages of the Sampling Process	32
4.4 Implementation Tools	36
4.4.1 Mininet	36

4.4.2	OpenFlow	36
4.4.3	Choosing the Controller	37
4.4.4	Switch Software	37
4.4.5	Other System Settings to Consider	38
4.4.6	Components Integration	38
4.5	Summary	39
5	TEST SCENARIOS AND RESULTS	40
5.1	Experiment Setup	40
5.2	Results	43
5.3	Discussion	46
5.4	Summary	48
6	CONCLUSIONS	49
6.1	Conclusions	49
6.2	Prospect for Future Work	50
	Bibliography	51
	Appendix A SDN SAMPLING CONTROLLER	57
	A.1 Sampling_controller.py	57
	Appendix B MESSAGE FLOW TO IMPLEMENT SAMPLING MECHANISMS	73
	Appendix C COMMANDS TO TEST THE ARCHITECTURE	74
	Appendix D OPENFLOW VERSION CHANGELOG	75

LIST OF FIGURES

Figure 2.1	Differences between traditional, hybrid and SDN networks [7]	5
Figure 2.2	Software-Defined Network architecture [11]	6
Figure 2.3	OpenFlow elements [7]	7
Figure 2.4	OpenFlow Switch starting from version 1.1 [13]	8
Figure 2.5	Open vSwitch components [29]	13
Figure 3.1	Sampling concepts [5]	17
Figure 3.2	Systematic Count-based [5]	18
Figure 3.3	Systematic Time-based [5]	18
Figure 3.4	Random n-out-of-N [5]	19
Figure 3.5	OpenFlow and sFlow interaction [43]	21
Figure 3.6	Action implemented by Flexam	22
Figure 3.7	PayLess in the SDN stack [49]	23
Figure 4.1	Representation of the system architecture	26
Figure 4.2	Packet-In message structure [50]	27
Figure 4.3	Packet-In message representation	27
Figure 4.4	Flow Mod message structure [54]	29
Figure 4.5	Flow Mod message representation	29
Figure 4.6	Group Mod message structure [56]	31
Figure 4.7	Group Mod message representation	31
Figure 4.8	Stages of interaction	32
Figure 4.9	Group Mod triggered to start the sampling mechanism	33
Figure 4.10	Group Mod sent twice to stop and restart the sampling mechanism	34
Figure 4.11	Sequence Diagram representing messages exchanged between switch and controller	35
Figure 4.12	Mininet diagram [35]	36
Figure 4.13	Architectural scheme	38
Figure 5.1	Topology overview on MiniEdit graphical user interface	41
Figure 5.2	Network interfaces	42
Figure 5.3	Comparison of SysT, LP and MuST sampling parameters in SDN	44
Figure 5.4	Throughput for OC-48 traffic	44
Figure 5.5	Throughput for OC-192 traffic	45
Figure B.1	Group Mod and Packet-In messages exchanged between controller and switch	73

LIST OF TABLES

Table 2.1	OpenFlow versions	9
Table 2.2	SDN controllers under analysis	11
Table 5.1	Traces used for testing	40
Table 5.2	System Information	42
Table 5.3	Metrics collected from all sampling techniques	43
Table 5.4	Throughput, error estimation and peak-to-average ratio	46
Table 5.5	OC-48 trace sampling results in [39]	46

LIST OF ABBREVIATIONS

AHP	Analytic Hierarchy Process 11
API	Application Programming Interface 1, 4, 7, 10, 12, 16, 36, 37
ASIC	Application Specific Integrated Circuit 13
ATM	Automated Teller Machine 5
AT&T	American Telephone & Telegraph 5
BOS	Bottom of Stack bit 80
CAIDA	Center for Applied Internet Data Analysis 40, 42
CBE	Container-Based Emulation 14
CDPI	Control-Data-Plane Interface 6
CLI	Command-line interface 14, 36
DDoS	Distributed Denial-of-Service 16
DPDK	Data Plane Development Kit 38
DPN	Deeply Programmable Network 50
EPL	Eclipse Public License 13
ForCES	Forwarding and Control Element Separation 5
GUI	Graphical User Interface 14, 15
IETF	Internet Engineering Task Force 5
IoT	Internet of Things 1
IP	Internet Protocol 43, 81
IRSCP	Intelligent Route Service Control Protocol 6
IVS	Indigo Virtual Switch 12, 13
LP	Linear Prediction 35, 40, 41, 43–46, 57
MAC	Media Access Control 33, 42
MCDM	Multi-Criteria Decision Making 11
MIB	Management Information Base 20
MP	Measurement Point 17, 18, 43

MPLS	Multiprotocol Label Switching 80
MTU	Maximum Transmission Unit 28
MuST	Multiadaptive Sampling Technique 35, 40, 41, 43–46, 57
NAT	Network Address Translation 7
NBI	NorthBound Interface 6, 10–12
NCP	Network Control Point 5
NEC	Nippon Electric Company, Limited 11
NIC	Network Interface Controller 50
NOS	Network Operating System 10, 20, 37
NTT	Nippon Telegraph and Telephone 11
ONF	Open Networking Foundation 6, 8, 14, 21, 36, 75
OOTB	out-of-the-box 27, 36
OVS	Open vSwitch 12, 13, 25, 38, 43, 48
OXM	OpenFlow Extensible Match 28, 79–82
OXS	OpenFlow eXtensible Statistics 81
PBB	Provider Backbone Bridging 80, 81
PCE	Path Computation Element 5
PPP	Point-to-Point Protocol 81
QoS	Quality of Service 7, 12, 16, 78
RCP	Routing Control Platform 5
REST	Representational State Transfer 12
RFC	Request For Comments 5, 20
RME	Relative Mean Error 40, 45, 47, 49
SBI	SouthBound Interface 6, 10, 11
SDN	Software-Defined Networking ii, iii, vi, 1–7, 10–16, 20, 22, 23, 25, 26, 36, 37, 39, 40, 43, 46, 48, 49
SLA	Service Level Agreement 6, 16
SystT	Systematic Time-based 35, 40, 41, 43–46, 49, 57
TLV	type-length-value 28, 79, 80
TTL	Time To Live 27
VM	Virtual Machine 36, 48

INTRODUCTION

Faced with the exponential increase in services supported by communication networks, monitoring activities are playing a vital role in the maintenance of these services. At the same time, it is important to ensure that network management activities are able to keep up with this accelerated pace, supporting next generation networks and seamless integration with trending concepts such as the [Internet of Things \(IoT\)](#), Cloud Computing and Smart Cities.

It is in this context that Software-Defined Networking ([SDN](#)) can play a difference, for the versatility and quantity of diversified solutions that can result from this concept for purposes of forwarding, monitoring, security, auditing, among others.

The main principle behind the [SDN](#) architecture is that the control and data plane are disaggregated. A centralized controller has a global view of the network infrastructure and directly communicates with the applications, establishing the bridge between them. To support it, a communication layer exists between the control plane and those components in the data plane, providing programmability of the network behaviour [1].

On the other hand, traditional networks have the data and control planes combined in a network node. In the control plane, forwarding policies are static, most times unaware of dynamic changes and specific demands coming from users and applications. This static approach has proven to be restrictive, affecting network scalability and performance as traffic patterns change [1], making it difficult to monitor the amount of packets and flows in each moment.

In monitoring systems, traffic sampling has become mandatory for effective network measurements, especially in the network core, reducing the packets collected to a manageable amount [2]. It consists of picking a subset of packets from the network to estimate different parameters about network traffic, not necessarily processing every part of it [3].

[SDN](#) flexibility enables programmable measurements, allowing a controller to manage concurrent measurement tasks at different spatial and temporal scales [4]. It also brings a different approach for deploying sampling techniques strategies, in order to try to obtain a balance between sampling efficiency and estimation accuracy [3].

1.1 MOTIVATION AND GOALS

Current mainstream network systems impose equipments provided with a restricted set of protocols, rules and interfaces ([Application Programming Interface \(API\)](#)).

With the advent of SDN, the possibility to easily extend the set of features with small code implementations has arisen. SDN can open ways for deploying sampling techniques that can be more efficient in accuracy and in the computational resources they require [5]. The development of software through SDN's provided interfaces, with the ability to control and take decisions over the network devices, emerges as a real possibility to turn this into a feasible concept.

There are currently several sampling techniques for different scenarios, but many are not simple to implement in today's infrastructures due to the limitations imposed by traditional network architecture such as its programmability. This situation can be changed through the use of SDNs.

Aiming at improving the efficiency of measurement systems, the main objective of this work is to explore SDN concepts, sampling measurement techniques and study possible solutions for applying the flexibility introduced by SDN into a software solution. In more detail, this involves:

- to study SDN concepts;
- to study existing solutions of software-defined measurements;
- to study the feasibility of employing sampling measurement techniques in SDN controllers, with particular emphasis on time-based sampling;
- to implement sampling techniques, using a controller prototype, that takes advantage of SDN concepts to collect useful statistical data from switches in real-time inflicting minimal overhead.

1.2 MAIN CONTRIBUTIONS

The main contribution from this work is to make use of the interaction between the SDN controller and switches to collect real-time statistics using sampling techniques with ideally low computational weight. The availability of this data at the controller level provides information that can be used to monitor and improve network stability, anomaly detection, provide network wide information to high level applications, among other tasks.

1.3 DOCUMENT LAYOUT

The structure of this dissertation is divided into six chapters covering the research work to achieve the objective proposed in this chapter, the results collected from the practical implementation, their analysis and prospects for future work.

Chapter 2 - *SDN State of the Art*: the SDN concepts and architecture are presented in this chapter. An overview of the OpenFlow communication protocol and versions features is provided, as well as virtual switching, Mininet virtualization technology, and several existing controllers.

Chapter 3 - *Monitoring implementation in SDN*: the sampling concepts are described and an analysis of existing solutions is carried out to understand how different tools are already taking advantage of the SDN architecture to collect network data. The study of this information is an important starting point for identifying common characteristics and possible ways forward.

Chapter 4 - *SDN sampling controller*: focus on the development decisions and implementation process of time-based sampling techniques using OpenFlow, initially defining the design goals to be achieved. In this chapter, a description about the setbacks and obstacles identified along the way is also provided.

Chapter 5 - *Test scenarios and results*: contains the experiments conducted with the sampling controller, including the results and their critical analysis and discussion.

Chapter 6 - *Conclusions*: includes the conclusions from this research and possible future work.

SDN STATE OF THE ART

SDN technology emerged from the disruptive idea of separating the control and data layers present in legacy networks, taking apart switching and routing functionalities into two distinct components: the control plane, responsible for the routing, and the data plane, responsible for the switching. This concept emerged from the need of having simpler and more flexible networks [6].

With SDN, developers are able to create software applications abstracted from the physical layer of the network, capable of controlling and monitoring network's connectivity in each network node and help to visualize the network as a whole, to identify issues and modify network settings to improve performance. This is possible with the open interfaces SDN provides [6].

In this context, this chapter addresses the fundamental concepts in SDN, starting with the historical contextualization that led to the creation of programmable networks. The OpenFlow protocol is analysed, as well as existing software in terms of controllers, virtual switching and emulation tools.

2.1 PROGRAMMABLE NETWORKS

Current networking paradigms can be divided in three categories considering the placement of data and control planes, represented in Figure 2.1.

In traditional, hardware-centric networks, the switches represent a unit that focuses on the control of the interfaces and manage data within. They are characterized by being closed to new approaches, usually with embedded software solutions inaccessible to external improvement and different specifications according to the manufacturer [7]. Therefore, it becomes difficult to implement new functionalities and protocols and even extend existing protocols. Usually it requires the hardware to be upgraded or replaced, leading to increased process costs.

Diverging from this way of thinking, in a SDN network the switch becomes simpler, serving the purpose of traffic forwarding. It communicates with a highly programmable controller, which imposes the rules of network management. This division of responsibilities with the assignment of the network control role to the controller, makes the system more flexible, receptive to new protocols and services. The switches and data contained therein become accessible through abstract APIs at application level, provided by the controller's programmability [7].

Lastly, a hybrid architecture intends to support centralized and distributed control planes. Some OpenFlow switches described in Section 2.5 are hybrid solutions, offering support for traditional protocols in addition to OpenFlow.

Figure 2.1: Differences between traditional, hybrid and SDN networks [7]

Although the concept of network programmability has been explored in more depth in recent times, research on this area was already being done even before SDN emerged. The first dated initiatives proposing to separate data and control signalling appeared in the 80s. **Network Control Point (NCP)** was probably the first project to try the separation of control and data plane signalling. **American Telephone & Telegraph (AT&T)** introduced NCPs to improve the management and control of its telephone network. From this point on, other projects were inspired by this proposal to design architectures that benefited from the global vision that NCPs provided [8].

SDN networks have given new relevance to network virtualization by using it to increase flexibility while abstraction network services. However, one of the first projects that addressed network virtualization was Tempest in the late 90s [8]. Tempest, an **Automated Teller Machine (ATM)** virtualization project, had the objective to conjugate multiple virtual networks to use the same shared physical resources. To decide over forwarding policies, a software controller was included [7]. MBone was another initiative to create virtual network topologies on top of legacy networks [8].

Forwarding and Control Element Separation (ForCES), an **Internet Engineering Task Force (IETF)** standard protocol defined through a number of **Request For Comments (RFC)**s, provides a communication interface between control and data elements for improved management in Ethernet [7] [8]. It suggests a flexible approach to traditional network management without changing the network's architecture, which means a centralized external controller is not required. Control and data planes are logically separated but can work together in the same network component. [8]. ForCES led to the development of Ethane in 2007 [9] and later OpenFlow in 2008 [10].

Other projects also promoted a centralized entity for network management and control, such as **Path Computation Element (PCE)**, **Routing Control Platform (RCP)** SoftRouter and

Intelligent Route Service Control Protocol (IRSCP). Security approaches also encourage the centralization of the control plane [7].

Ethane's authors proposed a different method to manage the network of a company. The main idea was the creation of a centralized flow policy managed by the controller. Switches have become simple machines that only have to forward and drop packets (match/action) according to the rules defined by the controller [7] [8] [9]. This project became the precursor of the OpenFlow protocol [10].

2.2 SDN ARCHITECTURE

The SDN system architecture, as described by the Open Networking Foundation (ONF) [11] [6], consists of the data, control and application layers.

The data plane is comprised of low-end network devices, that is, switches that communicate with each other and hosts connected to them that want to use the network to receive and send information. They communicate directly to the controller through the **Control-Data-Plane Interface (CDPI)** Agent. This lower interface is also mentioned as the **SouthBound Interface (SBI)**. The features within those resources are transmitted through SDN datapaths to the SDN controller in the control layer, as evidenced in Figure 2.2. In turn, the controller plays the role of intermediary between the data plane and the application plane, translating application requests onto the SDN datapaths and forwarding relevant information coming from the data plane towards the application plane. The communication between the controller and the application layer is established via **NorthBound Interface (NBI)** drivers.

Figure 2.2: Software-Defined Network architecture [11]

There is yet another layer responsible for the monitoring of the previous stated planes: the Management & Admin plane [6]. It monitors performance and defines policies at the controller level, sets up network elements at the bottom and manages **Service Level Agreement (SLA)** in the application plane. Control decisions, such as assigning SDN datapaths to SDN controllers, are taken in this plane.

A great benefit from this architecture is the possibility of coexistence with non-SDN networks. The interoperability created to support non-SDN networking systems intends to be an intermediate step that allows the cohabitation of different architectures, requiring less initial investment to change existing network infrastructures, and, ultimately envisions the migration of existing hardware and software to supply a fully enabled SDN network, bringing flexibility to perform tasks that traditional networks did not anticipate [6].

2.3 OPENFLOW

The main SDN communication standard was created in 2008 at Stanford University [10]. OpenFlow is non-proprietary and defines an API for communication between controller and switches [7]. This means both control and data planes must understand and correctly interpret the standard.

Initially, OpenFlow was developed for researchers to deploy experimental protocols on academic campus. However, at an early stage, the goal was also having network vendors including OpenFlow in their equipments [10].

The protocol arises from the need to have programmable networks, and was inspired by Ethane [10] [9]. After identifying that commercial solutions were inflexible and too closed, and performance problems and cost limitations related to existing research solutions were common, an approach providing switch flexibility began to be delineated [10]. It intended to be capable of providing high-performance and low-cost implementations, support different research purposes while assuring the isolation of experimental traffic from production traffic, and be able to meet vendors' requirements for closed platforms.

OpenFlow exploits the fact that most modern Ethernet routers and switches have flow tables running at line-rate to implement different features (such as firewalls, Network Address Translation (NAT), Quality of Service (QoS), etc) and gather statistics [10].

Being an open protocol, OpenFlow is provided to program the flow table in different switches and routers. A network administrator can decide to split network traffic into multiple categories by flow. It is thus possible to categorize and mark traffic to follow a particular processing path as desired [10].

Figure 2.3: OpenFlow elements [7]

Figure 2.3 describes the interaction between controller and switch using OpenFlow to communicate. The controller defines the rules to be set in the forwarding tables. The switch makes sure to forward packets according to the forwarding table. The OpenFlow interface is common to both entities which need to communicate through a common protocol.

2.3.1 OpenFlow Switch Specification

The OpenFlow Switch Specification is a document maintained by the ONF that describes the OpenFlow protocol properties [12].

The OpenFlow switch contains one or more flow tables and group tables (since version 1.1). A secure channel establishes the communication switch-controller via the OpenFlow protocol (see Figure 2.4) [13].

Figure 2.4: OpenFlow Switch starting from version 1.1 [13]

A flow table contains a list of flow entries. Each flow entry contains a set of fields that change according to the OpenFlow version. Fields available across multiple versions are: i) *match fields* - to match against packets, ii) *counters* - updated every time a packet matches the flow entry, iii) *instructions* - to define packet processing actions through the pipeline. When an incoming packet arrives at the switch, it is compared with the match fields of each entry and if a match exists, the packet is processed according to the flow entry instructions. Statistics about packets are collected through counters. The packet can be encapsulated and sent to the controller, if no match is found or if requested by the controller [12].

Flow entries may eventually point to a group, specifying additional processing. Sets of actions are defined by groups, to comply with actions such as flooding and more complex forwarding semantics, which are not achievable with flow entries alone [14].

The group table represents group entries. The group entry contains a list of action buckets with specific semantics dependent on group type. Packets sent to the group are manipulated according to defined action buckets [14].

Flow and group tables are managed by the controller via the OpenFlow protocol. The controller is able to manipulate flow entries and groups (add, modify and delete) proactively and reactively (when a Packet-In message is received) [14].

2.3.2 OpenFlow Versions

Some important changes in the main versions of the OpenFlow protocol happened as the protocol evolved.

This section reviews some of these changes to have a more in-depth knowledge of the protocol evolution, and thus be able to decide which will be the most suitable version for the implementation of the intended solution.

As highlighted in the latest version release notes (version 1.5.1), these changes are only informative and historical, some of which have been removed, replaced, or updated in later versions, and therefore may not conform to the most recent protocol specification [15].

Version 1.0 was the first OpenFlow version expected to have vendor support. Versions prior to 1.0 were marked as *Draft* with the information that one should not create a switch specification from these versions.

Table 2.1 summarizes the main features of each version, starting with version 1.0. A more exhaustive list of amendments that have been introduced in each major version is available in annex D [7] [16] [17].

Table 2.1: OpenFlow versions

Version	1.0	1.1	1.2	1.3	1.4	1.5
Release Date	Dec 2009	Feb 2011	Dec 2011	April 2012	Aug 2013	Dec 2014
Flow Table	Single	Multiple	Multiple (extensible match support)	Multiple (table miss support improved)	Multiple (flow monitoring)	Multiple (egress tables)
Group Table	No	Yes	Yes	Yes	Yes (group and meter change notifications)	Yes (group command for selective bucket operation)
IPv6 Support	No	No	Yes	Yes (ability to match extension headers)	Yes	Yes
Widely implemented	Yes	No	No	Yes	Yes (only some vendors)	No
Support for multiple controllers	No	No	Yes	Yes (auxiliary connections)	Yes (access to real time changes)	Yes (controller connection status)

2.4 SDN CONTROLLER

In SDN's network architecture, the control logic is displaced to an external entity, named controller or **Network Operating System (NOS)** [8], which plays a central role in all the dynamics of this architecture. This element establishes the communication with the devices in the infrastructure layer, using a protocol (*e.g.*, OpenFlow) to define the creation, modification or elimination of flow and group rules (as of version 1.1 of OpenFlow) through its **SBI**. It is also the intermediate point between the datapath and the application layer through the **NBI** [18]. A controller possessing an independent **NOS** promotes an abstraction layer **API** applications can use to achieve their purposes, be it monitoring, security, accounting, etc. The controller's purpose is comparable to that of a traditional operating system [8].

A single controller implementation may present scaling limitations and represents a point of failure in the architecture. For that reason, a distributed topology may be defined and controllers need to communicate with other controllers and non-SDN controllers in a distributed environment through east/westbound interfaces. In this case, information has to be exchanged between controllers, including features such as adjacency and capability discovery, topology information, among other details [8].

The choice of a suitable controller can represent a difficult decision because it is necessary to adapt the correct metrics to the proposed objective in order to reduce the list of controllers to analyze, whose number continues to increase over time [18].

This work focuses on the investigation of existing controllers prepared to work in a SDN architecture supporting the OpenFlow protocol in the **SBI**. It is also pertinent to find controllers that have open source code for easy modification and provide support for virtualization.

Based on their use, 5 controllers [18] were selected for analysis. They are: Ryu [19], POX [20], Trema [21], FloodLight [22] and OpenDaylight [23]. Other controllers such as NOX (C++/Python), FlowER, MUL (C), Jaxon, Beacon (Java), Maestro (Java), NDDI-OESS, NodeFlow (JavaScript) and ovs-controller (C), are not being considered because they either support a limited set of OpenFlow versions, have little documentation or a poorly-deployed implementation [18]. Controllers that have narrow scopes were also discarded (*e.g.*, SNAC (C++), Reasonance, RouteFlow, Flowvisor (Java) and Oflops (C)) [18].

After defining the controllers under study, the comparison of their characteristics is provided in Table 2.2.

Table 2.2: SDN controllers under analysis

	Ryu	POX	Trema	FloodLight	OpenDay-Light
Interfaces	SBI (OpenFlow) & SBI Man- agement (OVSDB JSON)	SBI (OpenFlow)	SBI (OpenFlow)	SBI (OpenFlow) NBI (Java & REST)	SBI (OpenFlow & Other SBI Procotols) NBI (Rest & Java RPC)
Virtualiza- tion	Mininet & Open vSwitch	Mininet & Open vSwitch	Built-in Emulation Tool	Mininet, Open vSwitch & ofsoftswitch	Mininet & Open vSwitch
Language Support	Python- Specific & Message Passing Reference	Python	C & Ruby	Java & any language that uses REST	Java
OpenFlow Versions	1.0-1.5	1.0	1.0 & 1.3	1.0 & 1.3	1.0 & 1.3
Multi- threading Support	Yes	No	No	Yes	Yes

- *Ryu* - Framework designed for SDN supported by [Nippon Telegraph and Telephone \(NTT\)](#). The word Ryu stands for "Flow" in Japanese [19]. The Ryu controller has a set of pre-defined components that can be modified and extended to create a customized controller application. Although Python is the main development language, any language can be used to develop a new component [18]. Ryu provides support for protocols to manage network devices, such as OpenFlow, Netconf and OF-config. Versions 1.0 to 1.5 of the OpenFlow protocol are fully supported as well as Nicira Extensions. The framework code is freely available under the Apache 2.0 license [19]. Selected as the best open source controller in [18] through a [Multi-Criteria Decision Making \(MCDM\)](#) method named [Analytic Hierarchy Process \(AHP\)](#).
- *POX* - Single-threaded controller that supports OpenFlow v1.0 switches [24]. Emerged from the NOX controller and is written in Python [18]. Supports Open vSwitch/Nicira extensions and requires Python 2.7 to run. Is able to run on Linux, Mac and Windows or any operating system as long as Python is supported [20]. Its modular structure allows the development of prototype controllers as well as integration into virtualized network environments [18].
- *Trema* - Full-stack OpenFlow framework to develop modular controllers in Ruby and C. It is supported by [Nippon Electric Company, Limited \(NEC\)](#) labs [18]. Provides a high-level OpenFlow library and also has its own network emulator to test and deploy

network experiments locally from any computer. The framework is available under the GNU General Public License version 2.0 or MIT License [21]. Contains an OpenStack Quantum plug-in for switch abstraction. Current OpenFlow supported versions are 1.0 and 1.3, even though Trema for OpenFlow 1.3 does not support version 1.0 [25].

- *FloodLight* - With its roots in the Beacon controller, FloodLight is a Java-based controller that consists of a set of modules that supply services to other modules and to the control logic applications through either a simple [Representational State Transfer \(REST\)](#) or Java [API](#). This controller software is compatible with Linux, Mac and Windows [18] and works with physical and virtual OpenFlow protocol (1.0 and 1.3) compliant switches. It is also ready to work with non-OpenFlow networks besides including support for OpenStack and multithreading. The software is distributed under the Apache license [22].
- *OpenDaylight* - Modular Java-based open source platform under the responsibility of The Linux Foundation [18]. The goal of the project is to promote the acceptance of the [SDN](#) concept among users and vendors, and to create quality code encompassing the existing components in the architecture [23]. It is also the goal of the project to have a community that regularly contributes with code development and promotes the use of the software in commercial solutions [18]. Designed to support multiple protocols and render maximum flexibility when creating new [SDN](#) controllers. OpenFlow versions 1.0 and 1.3 are supported [23]. [REST](#) and Java [APIs](#) are available on the [NBI](#) [18].

2.5 VIRTUAL SWITCHING

OpenFlow switch software must be carefully chosen. It has an impact on the performance of a [SDN](#) network. The switch is directly involved in the forwarding of packets and application of rules coming from the controller and must be able to cope with a set of tasks: match the packet header against flow entries and identify the actions to be implemented for every packet passing by.

When using an emulated environment in [SDN](#), only compatible platforms with network virtualization and support for the OpenFlow protocol are viable [26]. For this purpose, some software switch projects are discussed here. They are [Open vSwitch \(OVS\)](#), [OFSwitch13](#), [LINC](#), [Indigo Virtual Switch \(IVS\)](#) and the OpenFlow reference implementation switch.

- *Open vSwitch (OVS)* - [OVS](#) is a multilayer virtual switch working as a virtual switch within the hypervisor in virtualized environments and offers the possibility to be ported to multiple hardware platforms [7] [27]. XenServer, Xen Cloud Platform, VirtualBox and KVM are among the virtualization technologies supported by [OVS](#). The software is open source under the Apache 2.0 Linux license and distributed with some Linux distributions by default. [OVS](#) is also supported on FreeBSD, NetBSD and Windows [26]. [OVS](#) uses OpenFlow protocol to be applied across a variable number of hosts, to support efficient datapath management, switch configuration and implementation of [QoS](#) policies. The technology is compatible with versions 1.0 to 1.3 of the OpenFlow protocol.

Furthermore, **OVS** provides interfaces to interact with other monitoring protocols such as sFlow and NetFlow. Other standard management protocols are also supported (e.g., SNMP and NETCONF). It is common to see software solutions that integrate **OVS** with Mininet to test **SDN** networks using OpenFlow [26]. Figure 2.5 shows **OVS** modules and layers. The first packet crossing the switch's modules is sent to the userspace to be acknowledged (slow path). Subsequent packets are handled directly at kernel level (fast path) [28].

Figure 2.5: Open vSwitch components [29]

- *OFSoftSwitch13* - This software solution is an OpenFlow version 1.3 compatible switch solution running in the userspace, formerly supported by Ericsson Innovation center in Brazil [30]. OFSoftSwitch13 allows to incorporate the switch software with Mininet and the NOX controller [26]. The software is mainly written in C/C++ [7]. The support is now done on a best-effort base [30].
- *LINC* - Is an open source project compatible with OpenFlow protocol versions 1.2 and 1.3, designed to run on conventional x86 hardware and several operating systems (Linux, OSX, Windows, etc) [26]. The software is written in Erlang and supports OpenFlow versions 1.2-1.4 [31]. LINC is implemented in the operating system's userspace to allow flexibility and quick development of new OpenFlow features. The software switch contains a modular architecture, thus it is easily extensible [31].
- *Indigo Virtual Switch (IVS)* - Supports OpenFlow protocol 1.0 (only) on hypervisor and physical switches. It's an open source project aiming to provide high performance and requires little settings administration to use the hardware capabilities of **Application Specific Integrated Circuit (ASIC)**s and run OpenFlow at line rate [26]. It is developed and maintained by the same entity (Big Switch Networks sponsored community project) that provides the FloodLight controller [22] and is built on top of the Indigo Framework. The project is distributed under the **Eclipse Public License (EPL)** open source license [32].

- *OpenFlow reference implementation* - Minimal OpenFlow stack implementation used for OpenFlow specification tracking. It supports OpenFlow v1.1 and the code base is now considered an archive by the ONF [7].

2.6 EMULATION SOFTWARE FOR SDN

Container-Based Emulation (CBE) expedites the test of specific network case studies in a virtual environment before deploying them in productive networks. Software emulates hosts, switches and controllers to test the applicability of new ideas in a controlled testing environment.

Virtualization software runs real code on an emulated network using OS-level lightweight virtualization techniques, providing careful resource allocation. Emulation also provides realistic simulation testbeds [33].

Some existing tools are examined in this section, which streamline prototyping of different research ideas with software virtualization. The virtualization software frameworks under analysis are Mininet, ns-3, EstiNet and Trema.

- *Mininet* - is a platform to emulate multiple virtual end hosts, OpenFlow switches and controllers on a single machine [7]. Mininet supports rapid development in SDN using the OpenFlow protocol. It is the most popular platform used by researchers to explore SDN due to its flexibility and ease of use [26]. Moreover, Mininet focuses its implementation on the OpenFlow protocol [34].
The framework uses Linux kernels and Python language scripts to create a virtual network of large number of network components (switches, hosts, controllers) [26]. Mininet's **Command-line interface (CLI)** can be used to develop and customize different network topologies [35] and many external controller types and OpenFlow versions 1.0 - 1.3 are supported by this network emulator [26].
- *ns-3* - discrete-event network simulator for Internet systems, targeted for research and educational use. This software framework is currently licensed under the GNU GPLv2 license, and is available for research, development, and public use [36]. The native support provided by ns-3 is for OpenFlow 0.89 only, which is currently deprecated [7]. Therefore, researchers are limited to test and develop projects compatible with newer versions of OpenFlow. Another disadvantage is it cannot run a real OpenFlow controller such as Ryu, POX or Floodlight without some tweaks. ns-3 has implemented its own C++ OpenFlow controller as a module. It runs on Linux, Windows, FreeBSD and OSX [26].
- *EstiNet* - able to simulate thousands of OpenFlow v1.0 and v1.3 switches and run real-world controllers (*e.g.*, Ryu, POX, Floodlight, etc) [7]. EstiNet provides support for many other network protocols. It is a proprietary software and uses company servers to run the emulation projects. One of the distinguished characteristics of this software is its accuracy and repeatable results with a **Graphical User Interface (GUI)** that contains

simulation statistics graphically represented for each node in the network [26]. EstiNet 9.0 is the last known product version and runs on top of the Linux kernel [37].

- *Trema* - framework to help in the development of SDN controllers in Ruby and C [7] (see Section 2.4) and work as a network emulator [26]. The project source tree includes examples of sample applications, libraries and modules that work as an interface to OpenFlow switches. The software includes an integrated testing and debugging environment to help manage, monitor and diagnose tests with a network emulator and a diagnostic tool. OpenFlow 1.0 and 1.3 are the supported versions and the framework runs on Linux. Trema's popularity may be constrained since the software does not have a GUI and uses C and Ruby programming languages for development [26].

2.7 SUMMARY

At the end of this chapter, it is intended to have a global view about SDN, its architecture, history and purpose, and also existing software and protocols to provide a coherent connection between main entities composing the system.

Initially, the chapter starts by comparing SDN's disruptive proposal compared to traditional and hybrid networks, before describing each element that composes the architecture.

Next, the importance of the OpenFlow protocol is emphasized, enabling communication paths between controllers and switches.

A brief review of the functionalities introduced in several versions of the protocol is carried out, in order to identify improvements and guarantee the maximum compatibility for the targeted solution.

After, switch virtualization software is covered, highlighting their main features and compatibility with OpenFlow versions.

Then, the scope of existing network virtualization software is presented, for the architecture emulation and testbed creation.

MONITORING IMPLEMENTATION IN SDN

Monitoring tasks are an important part of communication networks for their proper maintenance and improvement. When coupled with sampling, they are able to offer a balance between accuracy and the impact they cause in the network, that is, the data volume collected for monitoring purposes.

An overview of sampling concepts in monitoring activities is provided in this chapter. Then, an insight of existing monitoring tools that rely on Openflow to collect their metrics is provided, and the challenges those projects face when implementing sampling techniques in [SDN](#).

3.1 SAMPLING IN MONITORING SYSTEMS

Traffic sampling techniques serve a wide variety of network tasks at current. These tasks still gain more prominence in [SDN](#), with the controller representing an intermediate point with which monitoring applications can communicate, through an [API](#) for the purpose.

Some examples of network activities are described in [38] and here presented:

- *Network planning and management* - network management solutions including operation, maintenance and service provision;
- *Traffic engineering* - modeling, control and characterization of traffic to optimize network performance;
- *Security* - detection of intrusions and anomalies, botnets, [Distributed Denial-of-Service \(DDoS\)](#) attacks and other cyber threats;
- *Performance evaluation* - detailed analysis of network reliability and management tools performance;
- *SLA* - measurement and reporting of [SLA](#) compliance with the use of auditing tools;
- *QoS control* - provide useful parameters such as packet loss and delay.

It is therefore important to realize the concepts that form the various sampling techniques in order to meet a set of requirements as broad as those just mentioned.

For a better understanding of the concepts related to packet sampling, the terminology used to describe the techniques discussed in this document is based on the following definitions [5][38]:

- *Sample* - set of packets selected from a network according to a particular criteria, essential for estimating the network parameters. It is also known as sample event - an individual event to select and capture traffic packets in the network;
- *Sample size* - defined by packet counters or time intervals delimiting the sample. Events running on the switch (or other device serving as **Measurement Point (MP)**) define the start and ending points of a sample, with reference to the position of the packet in the stream or the packet arrival timestamp;
- *Interval between samples* - similar to the sample size with the difference being the number of packets or the time interval in which the packets are ignored (for monitoring purposes), depending on the selection trigger.

The concepts of sampling are represented in Figure 3.1. The packets marked in blue correspond to the sample size and the remaining packets represent the interval between samples.

Figure 3.1: Sampling concepts [5]

Sampling techniques have several characteristics that identify them and may be grouped by their granularity, selection scheme and selection trigger. The definition of each of these components is as follows [38]:

- *Granularity* - atomicity of the element that is under analysis by the sampling process: packets represent individualized entities; whereas at the flow level, sampling can be applied to packets belonging to a pre-established stream or to a set of streams;
- *Selection scheme* - represents the function that characterizes the traffic selection. The selection scheme can be deterministic, random or adaptive;
- *Selection trigger* – designates the temporal or spatial sample delimiters that can either be count, event or time-based.

In the following subsections, a description of the content-independent sampling techniques is provided. They are so called because they do not require access to the packet content in the decision process, as opposed to content-dependent techniques [5]. The sampling process is ruled by the packet position in the stream or event timestamp, applying a deterministic, random function or adaptive algorithm depending on the intended technique.

Within the deterministic category are the Systematic Count-based and Systematic Time-based techniques, while Random n-out-of-N and Random Uniform Probabilistic are part of the non-deterministic techniques. Adaptive techniques use different sampling methods (*e.g.*, linear prediction) to adapt to network activity [5].

3.1.1 Systematic Sampling

Systematic Count-based

The Sampling Count-based technique performs the packet selection according to its position, through packet counters that define the starting point of a sample and the sample size to define the sample duration, resulting in a function with a deterministic pattern. A deployment example for this technique is the periodic selection of every k -th packet of a stream. In the scenario illustrated in Figure 3.2, every 5th packet is selected, the sample size is set to 1 and the interval between samples is equal to 4. This technique has been widely deployed by network equipment vendors and industry due to its ease of implementation and low resource utilization [5].

Figure 3.2: Systematic Count-based [5]

Systematic Time-based

In this sampling technique, a deterministic function is employed having the sample size and interval between samples as input parameters, with characteristics that resemble the systematic count-based technique. The difference lies in the fact that this technique is oriented to the arrival time of the packet at the MP. In this method, every incoming packet is selected during a specific time interval (defined by the sample size), and ignored if they reach the MP during the interval between samples. In the example depicted in Figure 3.3 the sample size is 100 milliseconds and the interval between samples is 200 milliseconds [5].

Figure 3.3: Systematic Time-based [5]

3.1.2 Random Sampling

Random n-out-of-N

The Random n-out-of-N technique is a random method to capture n packets from a sequential stream of N packets (n-out-of-N). A pseudorandom function is applied by this technique to generate n numbers between $[1, N]$, then picking the packets from the stream that are in the same position as one of the random numbers generated. The scenario depicted below in Figure 3.4 corresponds to a packet being select out of a stream of 5 consecutive packets [5]. It is important to highlight that in this technique the packet selection is performed with the same probability p (with $p = n / N$) for all packets of stream N .

Figure 3.4: Random n-out-of-N [5]

Random Uniform Probabilistic

This sampling technique consists of a predefined uniform probabilistic function deciding which packets are selected to constitute the sample, with the same probability of being selected. The packet content is not relevant for the selection. An example of implementation is based on a count-driven technique, defining the sample size to be 1 packet and the interval between samples to follow an independent random variable with distribution of mean $1/p$. Another approach would be to use a time-driven technique with the interval between samples following an exponential distribution [5].

3.1.3 Adaptive Sampling

Adaptive sampling strategies are used in techniques designed to change the packet selection during the capture period in the network. This flexibility aims to identify the most important parts of a traffic flow according to measurement needs and to use network resources as little as possible in critical periods of operation [5].

Some multiadaptive sampling techniques [5][39] use linear prediction to identify and adapt to changes in network activity by setting a more suitable sampling pattern, adjusting the sample size and interval between sampling, thus modifying the sampling frequency. This conduct leads to reduction of the weight of sampling in the network while maintaining measurement accuracy [39].

3.2 EXISTING MONITORING TOOLS BASED ON OPENFLOW

There are several proposals of monitoring systems in SDN that follow different methodologies and design goals. The next subsection contains an analysis with particular interest, insofar as studying these approaches is important to get to know the work already done in the field and understand the challenges towards designing the intended solution.

3.2.1 sFlow

sFlow is an industry standard, proposed as a RFC (*i.e.*, RFC 3176 [40]), used by multiple vendors and early adopted as the sampling technology embedded in many existing switches and routers.

This standard defines a set of sampling mechanisms which aim to decentralize monitoring operations typically implemented on the switches, by sending sampled packets to a remote external data collector in partnership with one or more sFlow Agents for monitoring traffic and a sFlow Management Information Base (MIB). Its purpose is to control the sFlow Agent and the format of sample data used by the sFlow Agent when forwarding the data [40]. With sFlow monitoring, the sampled packet headers are sent to a traffic analysis application which decodes the packets and aggregates the data to provide network visibility when requested [41].

Many of the network operators and manufacturers interested in OpenFlow are also members of the sFlow.org industry consortium and the sFlow standard is widely supported in Ethernet switches, including many of the OpenFlow and non-OpenFlow switches. This helped that, at an early stage, an OpenFlow version targeted at vendors and real world devices began to be planned between the members of the consortium [42].

sFlow integrates OpenFlow to perform decoding and analysis of network traffic with external software, allowing the NOS to delineate measurements in consistency with its internal network model. Hardware-based measurements, in contrast, are inflexible, requiring switch vendor hardware/firmware upgrades to implement new measurements and features. This is a bounding characteristic, sometimes causing measurements obtained with different hardware to be inconsistent between suppliers, or even within the range of products from the same manufacturer [41].

The diagram in Figure 3.5 shows how sFlow and OpenFlow can play complementary roles when implemented over the same network [43]. On one side, the OpenFlow protocol provides a controller running its own software on a separate component, responsible for configuring both switches and hardware forwarding tables. The sFlow standard specifies instrumentation to collect real-time information about the state of the network and sends it to a remote monitor. This protocol also provides real-time statistics in regards to servers' performance [43].

Figure 3.5: OpenFlow and sFlow interaction [43]

A research was conducted at HP Labs to replicate the hybrid sFlow/OpenFlow controller interaction pictured above, combining together the strengths of sFlow and OpenFlow, as a combined measurement and control framework, in order to create a scalable traffic control system. The research revealed interesting results such as reducing the controller-level overhead [44] [43].

sFlow provides only two count-based techniques - systematic and random. Consequently, the standard does not include other sampling strategies, such as the time-based techniques, which are relevant for other kind of activities, such as anomaly detection [5].

An equally important fact to highlight is the project update status, whose latest version (version 5) was launched several years ago [45], meaning new sampling techniques have arisen in the meantime, even with direct interaction with OpenFlow, and possibly have not yet been integrated by sFlow.

3.2.2 FleXam

FleXam is a per-flow sampling framework and is described by its authors as a flexible sampling extension for OpenFlow that enables the controller to access packet-level information. The tool provides two sampling possibilities: stochastically, by selecting packets according to a predetermined probability p ; or deterministically, which involves catching m consecutive packets from each k consecutive packets, skipping the first δ packets of the flow [46]. Both methods provide flexibility to match different measurement purposes. The controller has the power to decide which parts of the packet should be sent (*e.g.*, headers only, payload, etc.), as well as their destination.

To meet its design goals, modifications to the OpenFlow protocol specification have to be made to include monitoring abilities, thus proposing changes to the standard that is maintained by the ONF [46] [47]. Sampling is defined through a new action (OFPAT_SAMPLING) that can be assigned to each flow, whose parameters are described in Figure 3.6. The new

action is similar to `OFFPAT_OUTPUT`, with the objective of sending the sampled packets to the controller.

Figure 3.6: Action implemented by Flexam

The action has six parameters: *scheme*, *p*, *m*, *k*, δ and *destination*. The *scheme* parameter defines the sampling scheme that identifies which parts of the sampled packets should be sent. The *m*, *k* and δ parameters are defined to be used for deterministic sampling, and the *p* parameter is the probability used by the stochastic sampling technique. The *destination* parameter specifies the host to where sampled packets should be sent. It may be the controller or other network element that will handle the sampled packets [46].

From the analysis made to Flexam, it is noticeable that the customization of the sampling mechanism has obvious implementation advantages. It can be added to current OpenFlow implementation with no need for matching specific flow tables or perform multiple actions. Secondly, there's no overhead for flows that do not require sampling, given it is flow driven. Nonetheless, this approach promotes unwanted fragmentation in the implementation of a protocol (OpenFlow) that, in itself, is already implemented in several versions by the manufacturers.

Flow sampling has however associated disadvantages, such as not promoting a direct packet sampling mechanism, making it impossible to implement traffic sampling techniques [46].

3.2.3 PayLess

One different approach is supplied by *PayLess*. It is a query-based monitoring framework for SDN that provides a flexible RESTful API for flow statistics collection at different aggregation levels [48].

This monitoring framework gathers real-time information with high accuracy without provoking significant network overhead. PayLess replaces the controller policies when polling the switches, implementing an adaptive scheduling algorithm proposed to achieve the same level of accuracy as continuous polling but with much less communication overhead. Thereby, different network applications can develop new monitoring applications and access the data

collected by PayLess with the different aggregation levels (such as flow, packet and port) the tool provides.

Figure 3.7 illustrates the positioning of the framework in the SDN stack.

Figure 3.7: PayLess in the SDN stack [49]

This strategy offers the possibility of also getting information from the network from time to time without causing much network overhead. It is comparable to the behaviour when a flow is being removed from the switch. In that scenario, if the flow entry's `OFF_SEND_FLOW_REM` flag is set, the switch is required to send a flow removed message to the controller when the timeout is reached, with a complete description of the flow that has been removed, including statistics [13]. The monitoring frequency in PayLess is adjusted according to network load [49].

PayLess's evaluation results evidence low overhead - 6.6 monitoring messages are sent per second on average - as opposed to the controller's periodic polling, with an overhead of 13.5 monitoring messages per second on average [48]. The trade-offs between accuracy and the monitoring overhead within the given minimum polling interval (T_{min}) demonstrate that the monitoring data is accurate but the message overhead is very high for a short time interval, *e.g.*, $T_{min} = 250$ ms. However, for a large time interval, *e.g.*, $T_{min} = 2000$ ms, the message overhead is low but the monitoring data error increases substantially [48]. Hence, the accuracy of monitoring implies increasing the cost of network overhead [48].

Moreover, it should be noted that this kind of solution is highly dependent on OpenFlow collectable statistics, potentially compromising the solution's flexibility for specific monitoring objectives and does not provide any traffic sampling solution.

3.3 SUMMARY

Sampling plays an important role in monitoring because it allows inferring over a network and its state without the need to directly inspect the entire flow of data in circulation. Monitoring tools give network administrators the ability to detect the amount of resources in use

and to be able to make decisions that aim to improve overall network performance at all times.

As described in this chapter, there are several solutions created with the purpose of providing monitoring tools in SDN, with sFlow being the best known, working through agents that collect data from the network components and store them remotely in a monitor to be subsequently scrutinized. Another solution is FleXam, which identified the lack of OpenFlow messages dedicated for monitoring activities and therefore provides a single sampling solution for different techniques by adding a new rule to the protocol. Finally, Payless is presented as a framework that offers monitoring for potential application developers, collecting and aggregating OpenFlow flow statistics to be easily provided to the application layer through its own API.

It is important to note that all monitoring systems discussed in this chapter suggest implementations that work in parallel with the OpenFlow protocol (sFlow), integrate new rules (FleXam) or rely on flow statistics (PayLess) in order to design efficient and accurate sampling mechanisms, but do not consider an implementation which has the benefit of exploring existing OpenFlow messages in their sampling mechanisms.

SDN SAMPLING CONTROLLER

This chapter contains the development decisions which have been weighted and a detailed description of the design and implementation process proposed to implement network sampling techniques, using SDN's architecture and available OpenFlow features, after the research done on the subject. The chapter also includes the reasons for choosing Mininet, Ryu and OVS.

The design was accomplished considering the most updated features and software versions at the time, namely the OpenFlow switch specification and existing protocol messages, Ryu framework development status, as well as the latest version of OVS.

4.1 DESIGN GOALS

Proposing a new sampling approach in SDN raises several challenges. In the previous chapter, the solutions presented suggested additional elements to be included in the existing architecture, ignoring the changeability that can be made at the controller level.

After unveiling the objective of providing sampling techniques at the controller level, it was necessary to start by defining the design goals to accomplish the purpose at hand before proceeding to the implementation phase.

An investigation was carried out to explore ways to implement and configure sampling using the OpenFlow counters information. Since the protocol was not designed to provide this information on demand efficiently, since it would require obtaining counters' values frequently leading to greater overhead, this hypothesis was discarded.

The design goals must suit SDN's architecture approach while at the same time meet the objectives outlined in Chapter 1. They are as follows:

- The solution must provide sampling measurements that are efficient and lightweight respecting the SDN model and OpenFlow standard;
- Ensure compatibility with any OF-enabled Layer 2 and 3 devices;
- Overcome OpenFlow related constraints regarding sampling collection by defining methodologies in the controller that minimize this protocol gap;
- Explore the separation of the data and control planes to collect and make available sampled measurements in real-time environments;
- Find mechanisms that allow the application of time-based sampling techniques following all previous assumptions.

4.2 SYSTEM ARCHITECTURE

The system architecture proposed is based on elements already available in SDN, without the need to include extra modules to provide sampling mechanisms for monitoring tasks.

Figure 4.1: Representation of the system architecture

The proposed mechanism consists of withdrawing utility from the OpenFlow messages that constitute the communication channel between the controller and the switches in a typical SDN architecture, implementing sampling techniques depending on the available messages' parameters.

Therefore, the main components used in the proposed strategy are the ones shown in Figure 4.1: the network controller; OpenFlow and compliant switches.

- *Network controller* - This element plays an important role in the architecture not only because it has a global view of the system but also because it is responsible for the implementation of sampling mechanisms in the switches. By its role, in case of failure it can cause serious problems in network control and management operations, something that should be planned but can be solved with the inclusion of more controllers to function in a distributed way. It is also important for the controller to be able to support recent versions of OpenFlow to fully exploit the improvements introduced over time.
- *OpenFlow Message Layer* - OpenFlow has become the standard communication protocol to transport messages between controllers and switches with ruling decisions, statistics and packet forwarding. If we consider packet forwarding for sampling purposes, it is poorly explored using OpenFlow, when compared to flow statistics for monitoring purposes. A method designed to efficiently carry packets to facilitate the collection of relevant data, might produce interesting results.

- *OpenFlow switch* - Native support for the OpenFlow protocol must be provided by the switches in the data plane, including support for recent versions of the protocol for better compatibility. A software layer which renders dynamic network control in autonomous entities is also a requirement, to be able to cope with the packet forwarding and buffering processes and properly interpret and implement OpenFlow messages, while maintaining good performance.

4.3 IMPLEMENTATION DECISIONS

In order to force packet collection per request, the controller needs to communicate this intention to switches. For this reason, beyond the elements that establish the architecture, a set of relevant messages within the OpenFlow standard are worth mentioning, as well as their combination to achieve the proposed objective.

4.3.1 OpenFlow Messages Used in the Sampling Process

The three most relevant messages OpenFlow provides **out-of-the-box (OOTB)** in its switch specification tailored for the task are Packet-In, flow table modification and group table modification messages:

- *Packet-In* - Exists since the conception of OpenFlow as an idea to become the protocol between distributed components and began to gain practical implementation in version 1.0. It is the way the switch sends a packet to the controller when packets are received by the datapath, technically called OFPT_PACKET_IN, represented in Figure 4.3. There are three reasons for this message to be sent: i) resulting from an explicit action defined by a match rule asking for this behaviour; ii) due to a miss in the match tables; iii) caused by a **Time To Live (TTL)** error [50].

Figure 4.2: Packet-In message structure [50] Figure 4.3: Packet-In message representation

The message structure under analysis represented in Figure 4.2 was defined in OpenFlow 1.3 (same applies to subsequent figures in this chapter concerning message structures). The reason for choosing this particular version is related to two main factors: i) the significant amount of changes compared to previous versions of the protocol makes

this version very relevant among existing versions; ii) it is the latest version currently supported by many manufacturers' hardware [51].

Packet-In header fields are the *buffer_id*, *total_len*, *reason*, *tbl_id*, *cookie* (introduced in OpenFlow 1.3), *match* and *pad* (padding bytes) [14]:

- *buffer_id*: opaque value used by the datapath to identify a buffered packet. When a packet is buffered, a defined number of bytes is included in the data field. If the packet is sent due to an explicit action from the controller, then *max_len* bytes from the *ofp_action_output* of the flow setup request are sent. If the packet is sent due to a flow table miss, then the *miss_send_len* parameter will set the bytes from the OFPT_SET_CONFIG message to be sent. The default *miss_send_len* is 128 bytes but can be modified and for test reasons it is changed to 1500 bytes in the simulation specification (Ethernet [Maximum Transmission Unit \(MTU\)](#) [52]). If the packet is not buffered - if no buffers are available or when explicitly requested via OFPCML_NO_BUFFER - the entire packet is included in the data field, and the *buffer_id* appears as OFP_NO_BUFFER.
- *total_len*: identifies the full length of frame.
- *reason*: the reason field contains a variable stating the reason for sending the message. It can be set as any of these values:

```
/* Why is this packet being sent to the controller? */
enum ofp_packet_in_reason {
    OFPR_NO_MATCH      = 0, /* No matching flow (table-miss flow entry). */
    OFPR_ACTION         = 1, /* Action explicitly output to controller. */
    OFPR_INVALID_TTL    = 2, /* Packet has invalid ttl */
};
```

- *tbl_id*: identifier indicating the table being looked up;
- *cookie*: cookie of the flow entry that caused the packet to be sent to the controller. This field must be set to -1 (0xffffffff) if a cookie is unable to be associated with a particular flow. This would happen if, for example, the Packet-In was generated in a group bucket or action set;
- *match*: corresponds to a field against which each packet is matched, including the header, ingress port and metadata value. It must reflect the packet's headers and context when an event triggers the Packet-In message and includes a set of [OpenFlow Extensible Match \(OXM\) type-length-value \(TLV\)](#) fields. These include any modifications applied to the packet in previous processing, such as actions already executed, but not any changes that occurred in the action set. The [OXM TLVs](#) include context fields, i.e., fields whose values cannot be acquired from the packet data. There is a large set of conditions that can be specified to match, and the list increases each time OpenFlow is updated. OpenFlow 1.0 had 12 types; in OpenFlow 1.3 40 types of conditions are available [53];

- *pad*: exactly 2 all-zero padding bytes to separate the header fields from the packet payload and ensure that the IP header (if any) following the Ethernet header is 32-bit aligned.

The data field is the last field in the Packet-in message structure and contains either the total length of the packet being forwarded or a fraction, if the packet is buffered. The packet header incorporates records of any changes applied to the packet in previous processing.

- *Flow Mod* - One of the main messages existing in OpenFlow that allows the controller to modify the flow characteristics of an OpenFlow switch (see Figure 4.5). It begins with the standard OpenFlow header (see Figure 4.4) specifying the appropriate version and type values, followed by the Flow Mod structure [54].

Figure 4.4: Flow Mod message structure [54] Figure 4.5: Flow Mod message representation

The standard header defines the beginning of the message, followed by specific Flow Mod fields with specific functionalities here succinctly described [14] [54]:

- *cookie*: opaque identifier set by the controller. Not used by the packet processing pipeline. When a flow entry is inserted in a table, its cookie field is set to the provided value. When a flow entry is modified, the cookie field remains unchanged;
- *cookie_mask*: mask used to limit the cookie bits. Must match when the command issued is to modify - OFPFC_MODIFY* - or delete - OFPFC_DELETE* - the matching flows. The field is ignored for insertion actions. Number zero indicates there is no restriction;
- *table_id*: represents the id of the table into which the flow entry should be inserted, modified or deleted. Table zero indicates to use the first table in the pipeline. OFPTT_ALL can be used to delete matching flows from all tables;
- *cmd*: command to specify the type of flow table modification. Flow modification messages can have one of the following types:


```
enum ofp_flow_mod_command {
    OFPFC_ADD          = 0, /* New flow. */
    OFPFC_MODIFY        = 1, /* Modify all matching flows. */
    OFPFC_MODIFY_STRICT = 2, /* Modify entry strictly matching wildcards and
                               priority. */
    OFPFC_DELETE        = 3, /* Delete all matching flows. */
    OFPFC_DELETE_STRICT = 4, /* Delete entry strictly matching wildcards and
                               priority. */
};
```

Each variable represents the purpose of the flow modification message, which may imply the implementation of a rule (OFPFC_ADD); modification or flow removal for all matching flows (OFPFC_MODIFY or OFPFC_DELETE, respectively) or strictly matching wildcards and priority (OFPFC_MODIFY_STRICT and OFPFC_DELETE_STRICT, for flow change or removal, respectively);

- *idle_timeout*: number of seconds stipulated for flow removal since packet inactivity. Flow activity resets the elapsed time;
- *hard_timeout*: number of seconds for unconditional flow removal after rule creation;
- *priority*: priority of the flow entry. Higher numbers set higher priorities. Used only when matching and adding flow entries, and for OFPFC_MODIFY_STRICT or OFPFC_DELETE_STRICT messages when matching specific flow entries;
- *buffer_id*: buffered packets have an identifier originated from a Packet-In message that is preserved by Flow Mod. A valid *buffer_id* sent to the switch is a requirement for the switch to fully process the Flow Mod;
- *out_port* and *out_group*: optional field to filter the scope of OFPFC_DELETE* messages by port and/or group;
- *flags*: There is a set of flags to be (optionally) applied along with the Flow Mod message. Valid values are:

```
enum ofp_flow_mod_flags {
    OFPFF_SEND_FLOW_REM = 1 << 0, /* Send flow removed message when flow
                                     * expires or is deleted. */
    OFPFF_CHECK_OVERLAP = 1 << 1, /* Check for overlapping entries first. */
    OFPFF_RESET_COUNTS  = 1 << 2, /* Reset flow packet and byte counts. */
    OFPFF_NO_PKT_COUNTS = 1 << 3, /* Don't keep track of packet count. */
    OFPFF_NO_BYT_COUNTS = 1 << 4, /* Don't keep track of byte count. */
};
```

- *pad*: 2 padding bytes with the purpose of clearly separate the previous and next fields;
- *match*: same concept as for the Packet-In message;
- *instructions*: instruction set to be included in the datapath for the packets matching the flow entry and match field constraints, if any.

- *Group Mod* - Historically, group table modification messages (henceforth Group Mod) were defined in OF 1.1 [55] and are initiated by the controller to add, delete or modify groups in the datapaths with the purpose of defining groups of action for certain flows (Figure 4.7). The validation of the action set in the bucket follows the rules defined in Flow Mod [56].

Figure 4.6: Group Mod message structure [56] Figure 4.7: Group Mod message representation

In addition to the header field present in all OpenFlow messages, the following fields are part of the structure defined for the Group Mod message depicted in Figure 4.6:

- *command*: Group command to be executed. Can be represented by one of the following variables:

```
/* Group commands */
enum ofp_group_mod_command {
uint32_t group_id;
struct ofp_bucket buckets[0]; /* The length of the bucket array is inferred
    OFPGC_ADD      = 0, /* New group. */
    OFPGC_MODIFY   = 1, /* Modify all matching groups. */
    OFPGC_DELETE   = 2, /* Delete all matching groups. */
};
```

- *type*: defines the type of Group Mod message to be sent to the switch. It must be one of the following:

```
/* Group types. Values in the range [128, 255] are reserved for experimental
 * use. */
enum ofp_group_type {
    OFPGT_ALL      = 0, /* All (multicast/broadcast) group. */
```

```

OFPGT_SELECT    = 1, /* Select group. */
OFPGT_INDIRECT = 2, /* Indirect group. */
OFPGT_FF        = 3, /* Fast failover group. */
};

```

- *pad*: pad to place to next fields 64 bits aligned;
- *group_id*: sets the group identifier;
- *bucket*: structure containing the *len*, *weight*, *watch_port* and *watch_group* fields;
- *len*: length of the bucket in bytes, including the header and any padding to make it 64-bit aligned;
- *weight*: field only useful for select groups where a traffic distribution decision is made by the switch based on the value defined here;
- *watch_port* and *watch_group*: indicates the port and/or group whose liveness controls whether the bucket is a candidate for forwarding. Required for fast failover groups only;
- *action_header*: has the same scope of the action field for Packet-In and Flow Mod messages, which is to refine the packets on which decisions are to be taken, based on packet headers and context.

4.3.2 Stages of the Sampling Process

After studying the messages that enable sending packets upon express request of the controller, the solution can be put into practice.

In this case study, with the defined objective of collecting network details at packet level, and thus implement packet-level sampling techniques, all existing and active flows in the network are valid for being queried, aiming at obtaining a diverse and reliable measurement of the state of the network.

The main interactions can be represented by two scenarios (see Figure 4.8).

Figure 4.8: Stages of interaction

Once the controller and switch acknowledge each other, the controller waits to detect activity in the network. This signal is triggered by the switch when it receives a packet for which it has no actions defined in its flow table (table-miss), and therefore sends it encapsulated in a Packet-In message to the controller.

In turn, the controller parses the packet header in order to identify a possible destination in the network for the packet, based on pre-established rules. If a valid destination is found, a Flow Mod to add the new flow is sent to the switch, so that from then on the packets with the same destination can be directly forwarded, without incurring the penalty of awaiting the controller's decision. So far, this is the expected procedure for a [Media Access Control \(MAC\)](#) learning switch.

What happens while this process unfolds is what is depicted in Figure 4.8a: when the first Packet-In message arrives at the controller, it will take advantage of this moment to send a Group Mod message to insert into the switch called *the sampling group*, essentially containing a bucket with two actions : to continue the standard packet forwarding action for new packets with the same characteristics; to create a copy of each packet that should be sent to the controller, which will take care of saving it without triggering further actions.

Not least important is the use of Flow Mod messages in this prototype, responsible for the communication to the switch, present at the network layer, the set of rules to be followed for packets with those characteristics. For the Group Mod message to be correctly parsed, the actions field of the Flow Mod message includes a group with the action bucket to be fulfilled, either simultaneously for all rules, if the sampling mechanism is enabled, or simply to forward the packets to their destination, when the mechanism is off.

In a second moment, portrayed in Figure 4.8b, the controller decides to stop receiving all the packets from the switch. For this purpose, it communicates the intention by sending a Group Mod message, with the requirement to modify the previously created group so that the switch only routes the packets to their destination.

In the scenario illustrated in Figure 4.9, the controller injects an action bucket into the switch to start getting all the packets indiscriminately, including payload, previously specified with an OFP_NO_BUFFER instruction.

Figure 4.9: Group Mod triggered to start the sampling mechanism

This goal is achieved after pushing the Group Mod message into the switch. In turn, the switch sets up the group rule that requires sending Packet-In messages following the specified selection criteria. The scenario exemplified in Figure 4.9 corresponds to the scenario illustrated in Figure 4.8a.

The message sequence is intended to illustrate the moment of interruption and reactivation of the sampling mechanism:

Figure 4.10: Group Mod sent twice to stop and restart the sampling mechanism

The example shown in Figure 4.10 corresponds to the set of messages Figure 4.8b visually describes.

Joining both stages of the sampling process, the sequence of messages exchanged over time, directly related to sampling instructions and sampled packets, results from the continuous execution cycle of two scenarios Figure 4.8 represents, and may be found in detail in Appendix B. Flow Mod messages were not represented in that figure by the fact they do not directly intervene in the applied strategy, although they allow implementing the group responsible for the sampling process at an early stage.

To better understand all dynamics involving the communication process between switch and controller entities in this architecture, a sequence diagram is available in Figure 4.11. The sampling process is triggered by the arrival of data at the switch, which is in an initial state without configured flow tables. For this reason, a Packet-In message containing a packet whose destination is unknown is sent to the controller. Then, after detecting a Packet-In event, the controller will analyse the packet header and send two messages, a Group Mod and a Flow Mod, establishing the sampling group and flow routing rule to which the packet belongs. From that moment on, a thread is started to run the selected sampling mechanism. A sleep time is used to represent the sample size (the sampling mechanism is on when the thread is started), defined by the sampling technique. After this period, a new Group Mod message is sent to stop the sampling action, using a sleep time in the meantime equal to the

interval between samples, at the end of which a new Group Mod message will activate the sampling process, and so on and so forth.

Figure 4.11: Sequence Diagram representing messages exchanged between switch and controller

Despite the versatility offered by Group Mod messages in defining and changing sampling actions on the switch, it is possible to realize such strategy cannot be applied to all types of sampling techniques. Requiring to send packets over a long period of time, even if they are duplicated by the switch (implying they do not need to be returned in a Packet-Out message), has a negative impact, especially in periods of heavy congestion [10]. It is clear that from the techniques presented in Chapter 3, count-based and random techniques have to be discarded when thinking of model efficiency. Those sampling algorithms require continuous knowledge of all packets in the network. Time-based sampling techniques, on the other hand, can be controlled by timers set by the controller, which means they fit perfectly into the desired communication process. Therefore, the prototype implementation will focus on three distinct time-based techniques: a systematic technique - [Systematic Time-based \(SystT\)](#); and two adaptive techniques - [Linear Prediction \(LP\)](#) and [Multiadaptive Sampling Technique \(MuST\)](#) [39], to be analysed and compared in terms of network overhead.

An early strategy was designed to explore hard timeouts to trigger flow removal after a defined time value (specified in seconds) for a specific flow [14], instead of changing the group actions. This proposal intended to avoid sending a new Group Mod message whenever the controller intended to stop receiving all packets. The approach was discarded since the hard

timeout minimum value is 1 second [14], proved to be insufficient for the minimum values stipulated in time-based sampling techniques (millisecond precision required).

4.4 IMPLEMENTATION TOOLS

For the implementation phase, it is necessary to identify the available tools to demonstrate the development of sampling techniques in a SDN system and the protocols able to be used to create a solution nearly OOTB.

Below, the necessary elements working together for the creation of the prototype are presented, as well as the justification for their use.

4.4.1 Mininet

Mininet as a network virtualization tool opens the possibility to create realistic virtual networks from any personal computer or Virtual Machine (VM) within seconds [35].

Figure 4.12: Mininet diagram [35]

The Mininet CLI and API, alongside with the customization capability to create parametrized topologies, simplify the testing of network solutions before deploying them on real hardware [35]. Figure 4.12 displays the command to create switch(es), hosts, and the connection to the controller(s).

It also provides interfaces to experiment with OpenFlow and SDN systems, remote OpenFlow controllers and different switch software solutions [35].

All these characteristics plus those evidenced by Mininet in Section 2.5 make it the ideal platform to implement and test the sampling techniques and the OpenFlow protocol.

4.4.2 OpenFlow

OpenFlow is described by ONF as the first standard communications protocol between the control and data layers of a SDN architecture. Its open interfaces enable remote controlling of forwarding tables in network switches, routers and access points.

This work proposes a new approach to provide sampling measurements at the controller level, ready for use by monitoring applications, exploring the messages OpenFlow offers. Validate options working directly from OpenFlow's specified messages and that did not imply changes in protocol became a goal throughout the decision-making process, keeping the switch specification as defined by ONF. Therefore, the implementation can be easily config-

ured in existing equipment, seen as an additional feature to be included in existing controller solutions rather than an alternative approach.

4.4.3 Choosing the Controller

At an early stage, the use of POX controller was taken into account to serve as the [NOS](#) in the system architecture. POX is written in Python and supports OpenFlow. It includes an high-level [SDN API](#), including a queryable topology graph and support for virtualization [57]. It has the advantages of using a high-level programming language object-oriented and the fact of having an [API](#) allows to write code that is standardized, modular and fast to develop.

However, as described on the project page on GitHub, despite the special support for Open vSwitch/Nicira extensions, POX currently is only able to communicate with OpenFlow 1.0 switches [20]. This key aspect was relevant in the decision process, because important changes were made in later versions of the protocol. Using POX would result in an outdated solution due to the limited number of OpenFlow messages available to deploy sampling techniques in the controller-switch interaction.

Since POX's support is limited to OpenFlow 1.0 switches, a more in-depth study of existing alternatives that met the design needs was made.

Minimum requirement were then established to select the controller:

- Support recent versions of the OpenFlow protocol;
- Framework based on a programming language with good concurrency mechanisms and support for multithreading;
- Intuitive and properly documented [API](#).

From the list shown in Section 2.4, Ryu is the controller framework that fulfills all the requirements set. The project [API](#) is up-to-date and supports up to version 1.5 of the OpenFlow protocol [19]. The sample examples are intuitive and within the project scope.

The code that serves as basis for the implementation of the sampling techniques and interaction with the switches is developed from the *simple_switch_13.py* layer 2 switch, using OpenFlow 1.3 [58].

4.4.4 Switch Software

Choosing the switch software required a research focused on the efficiency of virtualization software and support for different versions of OpenFlow, so that it would not be limiting in the solution to be tested. Open vSwitch stood out, namely for the constant updating and support of several versions of the OpenFlow protocol (up to OpenFlow 1.5) and the concern for evolutive maintenance and speed improvement in the packet forwarding pipeline [59]. Among the projects compared in Section 2.5, Open vSwitch is the most active in terms of evolution.

Stating that the main focus of the work is not an overly detailed analysis of this particular component and Open vSwitch is capable of providing fast packet processing and OpenFlow versions versatility, the integration of Open vSwitch as the software used for the virtual switches to be integrated in Mininet seemed to be a sensible choice.

4.4.5 Other System Settings to Consider

There are also two extra settings that were considered for performance tuning in the implementation and testing, and must be mentioned:

- *OVS built with Data Plane Development Kit (DPDK)* - OVS was compiled to work with a DPDK datapath. That way, Open vSwitch can use the DPDK library to operate entirely in userspace [60]. The DPDK is a set of data plane libraries and network interface controller drivers, managed by the Linux Foundation as an open-source project that enables faster development of high speed data packet networking applications [61];
- *Linux preemptive kernel* - Linux itself is not real-time capable. The PREEMPT_RT patch provides real-time capabilities to the Linux kernel [62]. This change confers greater time precision on tasks that are time sensitive. A custom version with this feature was compiled to run the sampling tests.

4.4.6 Components Integration

Once chosen, the components have to be integrated and a parametrized topology defined in Mininet, composed of one OVS switch, two hosts and a remote controller. The Ryu controller works as a remote entity connected to the switch via OpenFlow. Figure 4.13 demonstrates the position of each component within the architecture.

Figure 4.13: Architectural scheme

4.5 SUMMARY

In this chapter, the inclusion of controller-level sampling mechanisms in a [SDN](#) architecture is proposed, which represents the basis for monitoring in various applications and can be a step forward by its implementation speed.

Design goals were first established while designing the solution to be achieved and a proposal about the architecture based on these principles was outlined.

Next, the set of important messages OpenFlow provided from scratch were identified and implementation decisions were made, taking into account the technical requirements of various sampling techniques.

Finally, the implementation description was made up of two well-defined phases delimiting the periods of activity and pause in the network sampling.

The intended solution may be feasible only for time-based techniques, since their mechanisms do not require specific information to be known by the controller, such as packet counters, that are essential for measurements obtained through count-based techniques. Considering the current version of the OpenFlow protocol, there is no effective way to get this information from the switches with the necessary precision and efficiency.

The chapter ends with the presentation of used software frameworks, their role and integration in the system architecture.

TEST SCENARIOS AND RESULTS

The set of experimental tests executed in Mininet is described in this chapter. The purpose of these tests is to assess the proposal's applicability in a virtual environment and that can be tested in the future in a real network environment. To do this, the controller uses a set of traffic sampling techniques widely discussed in the literature and used in production networks. In particular, it is possible to run three time-based sampling mechanisms - [SystT](#), [LP](#) or [MuST](#) - in [SDN](#) environments. After running the tests, a set of collected parameters are compared, analyzing the overhead created by OpenFlow to handle the process and the end results' reliability. This is possible by looking at the throughput and [Relative Mean Error \(RME\)](#) for all tested techniques, also crosschecking some of these metrics against results described in [\[39\]](#), where the same sampling techniques were applied in a real network environment.

5.1 EXPERIMENT SETUP

To carry out the experiment, different trace files were selected with packets transmitted in a real network. The traces were downloaded from different public datasets containing anonymized passive traffic.

[Center for Applied Internet Data Analysis \(CAIDA\)](#)'s OC-48 trace was taken at a west coast OC-48 peering link in the United States for a large ISP in 2003 [\[63\]](#). The OC-192 trace was captured in 2015 on a particular high-speed Internet backbone link [\[64\]](#), also in the USA. [CAIDA](#) removed the packets' payload included in the trace files for anonymity and security reasons. Files' characteristics are briefly described in [Table 5.1](#).

Table 5.1: Traces used for testing

Traffic Label	Characteristics	Duration	Source
OC-48	Anonymized passive trace taken at an US west coast OC-48 peering link for a large ISP in 2003.	5 minutes	CAIDA [63]
OC-192	Trace contains anonymized passive traffic collected at CAIDA's equinix-chicago high-speed monitor on a commercial backbone in 2015.	1 minute	CAIDA [64]

Even though there is some general knowledge about the mean throughput and total number of packets the traces contain, there is no in-depth knowledge of the network traffic pattern. The result of each sampling technique depends entirely on the parameters stipulated initially and the systematic or adaptive properties of the technique being used. In the case of **SystT**, due to the static characteristics of the technique, only the initial parameters are relevant to the final result.

In regards to time-based adaptive techniques, although the initial parameters are also important, the sampling intervals are modified over time by the techniques themselves in order to adapt the sampling efficiently to the characteristics of the traffic circulating in the network.

In this test environment, the **SystT** sample size and the time between samples parameters are set to 100ms and 500ms, respectively. These are the values also proposed in [39] and [65]. For the **LP** technique, the sample size is set to 100ms and the interval between samples is set to 200ms. The order of prediction (N), *i.e.*, the quantity of previous samples whose values are considered in the next sampling forecast, is set to 2, according to the original specification [65]. In **MuST**, the sample size and interval between samples are set to 200ms and 300ms, respectively. The order of prediction is set to 3 for this technique. These values were set according to their default values set in [39].

The testbed network topology is created with Mininet. Open vSwitch is used to provide a multilayer switch. The topology consists of: one controller $c0$; one switch $s1$; and two hosts - $h1$ and $h2$ - as shown in Figure 5.1:

Figure 5.1: Topology overview on MiniEdit graphical user interface

Figure 5.2: Network interfaces

Traffic source and destination in the CAIDA traces are overwritten to depict the topology represented in Figure 5.1. The header of each packet has to be changed to contain the MAC addresses of *h1* and *h2* (Figure 5.2) as source and destination, respectively.

For this purpose, the following command was used (example for OC-48 trace):

```
tcprewrite --dlt=enet --enet-dmac=00:00:00:00:00:02 --enet-smac=00:00:00:00:00:01
--infile=oc48-mfn.dirA.20030424-075000.UTC.anon.pcap --outfile=oc48_new.pcap
```

The test occurs as follows: the traffic is injected into the network using *tcpreplay* - a command line application that replays capture files made from any libpcap-based application. *Tcpreplay* is executed on *h1*, which in turn sends the traffic to the switch. The switch will forward the packets to the corresponding destination if it is already known. Otherwise, the unknown packets are sent to the controller for analysis, through Packet-In messages. Subsequently, the controller communicates with the switch through a Group Mod encapsulated inside a Flow Mod message, defining the action to be performed for a particular type of packet. From that moment on, it is only necessary for the controller to change group settings according to the sampling method, using the sample size and interval between samples parameters to define the amount of time before sending the next group modification.

Sampling results are stored in a path defined in the controller configuration in the form of a pcap file.

The commands used to define the topology, initiate the required services and the simulation can be found in Appendix C.

To perform the tests, the environment reported in Table 5.2 was used.

Table 5.2: System Information

Operating System	Linux Mint 18.1 Cinnamon 64-bit
Linux Kernel	4.4.86-r99 (PREEMPT_RT custom build)
Processor	Intel®Core™ i7-6500U CPU @ 2.50GHz x 2
Memory	8192 MB, DDR 3 (1866 MHz)
Hard Drive	256GB SSD (SATA, M.2 2280)
Mininet version	2.3.0d1
Open vSwitch version	2.8.90

5.2 RESULTS

A set of results herewith described was obtained after ten tryouts executed for each sampling technique, applying the average to the results collected.

The decision to collect the data in this way is due to the fact the test values are not constant because of OVS's variable latency and throughput limitation when using an emulated virtual network environment such as Mininet [28].

To calculate the overhead created by each sampling technique and OpenFlow, the parameters collected in the simulation must be examined. The study variables are the number of packets collected, number of Packet-In messages handled by the SDN controller, data volume and number of samples taken during execution. They represent objective values to do a comparative analysis of the obtained results [39].

In total, 60 tests (30 per trace file, 10 per sampling technique) were performed and the mean values are available below.

Table 5.3: Metrics collected from all sampling techniques

Traffic / Parameter	Total	SystT	LP	MuST
OC-48				
Number of Packets	6550395	539228	209322	289925
Number of Packet-In Messages		1254978	493600	604550
Data Volume (MBytes)	372.52	30.67	11.91	16.49
Number of Samples		493	193	42
OC-192				
Number of Packets	14990493	80965	38377	22318
Number of Packet-In Messages		683719	401047	246583
Data Volume (MBytes)	945.87	5.13	2.44	1.43
Number of Samples		82	43	16

The collection of the total number of Packet-In messages aims to assess the overhead introduced by sampling techniques applied over the proposed architecture in the communication from the switch to the controller, and compare it to a SDN architecture without sampling. To effectively do so, the overall number of Packet-In messages is collected at the controller. The ratio between the number of packets and this value indicates the overhead introduced along with sampling techniques.

Other chosen parameters meet the parameters mentioned in [39] and can be defined as:

- *Number of Packets* - amount of packets collected during the period of activity, set by the technique's sample size;
- *Data Volume* - size of the result file (in megabytes), containing the amount of packets collected. It is calculated from the sum of every packet's total length value, within the Internet Protocol (IP) frame;
- *Number of Samples* - indicates the number of times the packet capture process was triggered on the MP. In this case, the MP is the switch.

For a better understanding of the data in Table 5.3, comparative charts of the previously stipulated parameters were created among the three sampling techniques tested (Figure 5.3).

Figure 5.3: Comparison of SysT, LP and MuST sampling parameters in SDN

Figure 5.4 shows a visual representation of the original network throughput for the OC-48 traffic and its counterparts (estimated throughput) when applying each one of the time-based sampling techniques.

Figure 5.4: Throughput for OC-48 traffic

The throughput and sampled packets were collected in 1 second time intervals. Since traffic sampling implies selecting only part of the packet stream in the switch, for reliable es-

timation of the throughput it is necessary to also consider the packets that were not selected. This calculation can be done in several ways, the most common being a statistical extrapolation of the number of unsampled packets, from the sample value gathered in the defined time interval [5].

Following previous assumptions, Figure 5.5 is a graphical representation of the instantaneous throughput for the OC-192 traffic:

Figure 5.5: Throughput for OC-192 traffic

For quality purposes, some extra parameters of the original traffic and each sample have to be analysed. These parameters aim to quantify the reliability and the margin of error of the acquired sample.

The parameters in which this work focuses are the following [39]:

- *Throughput* - indicates the actual or estimated transfer rate (kbps) in each unit of time, depending on whether it is the original or sampled traffic. With the use of statistical extrapolation, allows a correlation between the expected average value and the value obtained with sampling measurements;
- *RME* - allows to infer about the mean error of the estimated average throughput compared to the average throughput of the total traffic;
- *Peak-to-average ratio* - parameter calculated from the ratio between the peak and average throughputs for the same sample. Helps to measure the maximum packet variance in regards to the calculated average.

The parameters calculated from the tests performed are listed in table 5.4.

Table 5.4: Throughput, error estimation and peak-to-average ratio

Traffic / Parameter	Total	SystT	LP	MuST
OC-48				
Throughput	9934.22	9934.90	9903.53	9936.72
RME		0.00007	0.00309	0.00025
Peak-to-average ratio	1.13	1.13	1.14	1.11
OC-192				
Throughput	154429.34	156406.07	162038.02	156623.15
RME		0.01280	0.04927	0.01415
Peak-to-average ratio	1.06	1.11	1.03	1.04

5.3 DISCUSSION

Table 5.5: OC-48 trace sampling results in [39]

Parameter	Total	SystT	LP	MuST
Number of Packets	6550395	1094002	935066	310427
Data Volume (MBytes)	3189.67	533.39	454.92	151.70
Number of Samples		500	431	54

Comparing the results presented in Table 5.5 against those obtained in this test scenario (see Table 5.3), after sampling the same OC-48 trace file, it is possible to attest the similarity of the values obtained with respect to the parameters defined in both test scenarios.

For instance, the percentage of data volume in the MuST technique, is 4.42% using SDN against 4.76% in [39]. The number of packets captured in this technique also follows this trend. The most notorious differences occur in the SystT and LP techniques, with the latter being the technique with the lowest volume of data and packets collected, while in [39], the MuST technique was the one that had the lowest values for all parameters under study. In the SystT technique, there is a decrease in the number of packets from 16.70% to 8.23%, having a similar percentage decrease of collected data compared to the original trace file.

Looking at the charts, the number of Packet-In messages has the peculiarity of resembling the graphical representation of the number of packets and data volume, as well. This is an interesting fact, having in mind many Packet-in messages were not originated by the sampling action. Effectively, Packet-In messages sent by the switch while OC-48 traffic was being forwarded, represent 42.97%, 42.41% and 47.96% of all Packet-In messages received by the controller, in SystT, LP and MuST sampling techniques, respectively. With regard to OC-192 traffic, Packet-In messages resulting directly from a sampling action represent 11.84%, 9.57% and 9.05% of all Packet-In messages in SystT, LP and MuST sampling techniques, respectively.

The visual differences between charts related to the two traces in Figure 5.3, are justified by their duration. As noted in Table 5.1, the OC-48 trace lasts for 5 minutes, while the OC-192

trace has only one minute of traffic captured in the network. Hence, the OC-192 sampling results are expected to be quantitatively below the OC-48 data sampled values. However, the difference is smaller for the number of Packet-In messages, which is explained by the large amount of information in the OC-192 trace sent in a short time that has to be properly communicated by the switch to the controller.

To corroborate the sampling accuracy in all time-based sampling techniques, the throughput, *RME*, and peak-to-average ratio parameters were collected. The values in Table 5.4 demonstrate that the sampling results present fairly acceptable values for both test scenarios, for all techniques, having an average accuracy and error level within the values observed in [39], for OC-48 traffic.

Despite having a larger margin of error, the OC-192 sampled traffic corresponds, statistically, to the total traffic (unsampled), as Figure 5.5 shows. Table 5.4 corroborates this assertion, with values of *RME* tendentially close to zero, in addition to values of throughput and peak-to-average ratio near to the values calculated on the total traffic.

Notwithstanding the good results obtained, there are some aspects relevant to the performance of Open vSwitch that should be mentioned. Open vSwitch log analysis showed that in times of heavy load on the virtual switch, *e.g.*, for OC-192 traffic, packet routing is not done for all packets, resulting in packet loss. The behaviour was not observable for the OC-48 trace, regardless of the sampling technique.

Example of log messages demonstrating this behaviour:

```
2017-07-24T19:16:41.218Z|00001|dpif_netlink(handler39)|WARN|
system@ovs-system: lost packet on port channel 2 of handler 1

2017-07-24T19:16:44.643Z|00002|dpif_netlink(handler39)|WARN|
system@ovs-system: lost packet on port channel 2 of handler 1

2017-07-24T19:16:44.883Z|00003|dpif_netlink(handler39)|WARN|
system@ovs-system: lost packet on port channel 2 of handler 1

2017-07-24T19:16:45.410Z|00460|connmgr|INFO|s1<->tcp:
127.0.0.1:6653: dropping packet-in due to queue overflow

2017-07-24T19:16:45.412Z|00461|connmgr|INFO|s1<->tcp:
127.0.0.1:6653: dropping packet-in due to queue overflow
```

The same pattern of loss happens with the implementation of `simple_switch_13.py` [58], which does not run sampling techniques. This is caused by empty routing tables on the switch in the first milliseconds of execution, forcing it to send the whole traffic to the controller (default action of OpenFlow), ultimately leading to buffer overflow. When this happens and the link cannot support the additional load, the packets are lost [66]. In the suggested architecture, there are short periods of packet loss and dropped Packet-In messages during the sampling of packets on the switch, throughout the execution.

These occurrences of packet loss have been foreseen in [10] for periods of heavy loading when all packets are being copied over to the controller and it is still valid for the latest version of Open vSwitch. This is substantially visible due to OC-192 traffic's characteristics (almost 15 million packets forwarded over one minute) and when only one switch is available to forward the burst of traffic imposed by the trace when there is no load balancing mechanism.

Another relevant fact is that latency is also affected by the load of the switch. This effect is especially visible when OVS is running in the Linux kernel due to its interruption techniques, leading to varying latency as the load changes [28]. The testbed created tried to minimize this with the use of a preemptive kernel, as previously explained in Section 4.4.5.

Virtual machines display higher latencies when under high load [28]. This is relevant because this work is using virtualization to emulate the topology (Mininet) and also in the switches' software (Open vSwitch). Packet throughput is also affected by virtualization: latency also increases. Nonetheless, moving packet processing systems or virtual switches and routers into VMs is problematic due to the high overhead created per packet when moving from VM to host and vice versa, and due to VM's latency-sensitive nature [28].

Factors such as latency and packet loss on the switch may require a rework, for example increasing the switch's buffer size or improving its design. Nonetheless, time sensitive applications will be subjected to high latency and jitter in the network if the buffer gets too big. The buffer size must be carefully planned depending on the network requirements [66].

The SDN architecture promotes multilayer programming flexibility. However, there are problems (such as the latency evidenced in the tests performed) that arise from the separation of layers that SDNs advocate and that good planning not always overtakes.

5.4 SUMMARY

A statistical and objective analysis of the results obtained was done in this chapter, by applying different sampling techniques to previously captured traffic and comparing some stipulated parameters to determine if the techniques correctly replicate the traffic behaviour. The study also intended to analyse the overhead created by OpenFlow messages used for this purpose and conclude if such an architecture can incorporate a productive environment.

The analysis corroborates the results presented in other studies, proving that it is possible to estimate the network throughput with an acceptable margin of error with a smaller volume of data collected by the controller.

There is strong evidence that the chosen architecture may overflow the switch and cause forwarding limitations in specific overload scenarios, such as for OC-192 traffic, inherent to the system architecture and the role the switch is required to perform in the periodic routing of data for sampling purposes. A limited number of switches and SDN's architecture may represent a bottleneck in this situation.

CONCLUSIONS

Some final considerations are included in this chapter about the methodology identified and tested in this work, to implement systematic sampling techniques at the controller level in a [SDN](#) environment. Several suggestions have also been elaborated from the extrapolated data and system limitations identified that may be relevant for future work on the topic.

6.1 CONCLUSIONS

[SDN](#)'s architecture has opened new possibilities for the creation of tools and network applications capable of serving the most varied purposes in a very practical way.

The decoupling of control and data planes with high programmability of the system changed the way network monitoring tools are being built. Following this trend, a set of functionalities to be included in the control plane has been proposed with the objective of performing sampling data measurements remotely in the data plane and verify if it could be done in a sustainable way, considering a network environment as close as possible to the real world. Moreover, it was also pertinent to see whether this implementation would bring some performance benefit over sampling models applied in legacy network environments.

From the data obtained in Chapter 5, it is possible to conclude that the implementation of sampling techniques at the controller level is a real possibility. The calculated evaluation parameters attest it is possible to determine the state of the network with a high degree of accuracy from the sampled measurements that the controller provides, with a very small [RME](#).

It was also possible to verify that there are elements inherent to the [SDN](#) architecture and test environment, such as virtual switching and delay in the transmission of packets that cause discrepancies in the final results of singular executions with the same settings. As showcased in [28], virtual switching is limited by the number of packets, not the overall throughput. It implies that a large number of small packets will be more difficult for a virtual switch to forward than a bulk data transfer. Virtual ports that are overloaded on the switch can lead to packet loss before reaching the maximum throughput [28].

On the other hand, it was expected to have some overhead caused by the OpenFlow messages exchanged between controller and switch during the application of the sampling techniques. Effectively, the number of OpenFlow messages used to implement the sampling mechanisms represent in the worst case (OC-48 traffic, [SystT](#) sampling technique) 8.25% of

all packets (including Packet-In and Group Mod messages) and did not have a negative effect on the packet forwarding, with no losses recorded. Moreover, the overhead created by these messages had in fact little impact on the final metrics (Group Mod messages alone represent 0.18% of messages compared to the sampled traffic).

For packet count-based and random sampling techniques, it is not possible to test them correctly at this level based on the original OpenFlow specification, which would imply the need to make changes to the protocol, create a new rule to be inserted directly into the switches as suggested in previous monitoring solutions [67], or use a solution at the controller level with questionable performance.

6.2 PROSPECT FOR FUTURE WORK

In terms of prospect for future work, several modifications can be done to improve the stability of final results by focusing on reducing the latency and eventual packet loss caused by virtual switching.

The first proposed scenario is to divide the traffic through more switches to reduce bottlenecks and packet forwarding limitations of the virtual network interfaces observed and produce more stable results. The sampling techniques tested are time sensitive and depend heavily on the overall delay.

Packet throughput and latency are adversely affected by virtualization. Conducting the tests with physical [Network Interface Controller \(NIC\)](#) can also contribute to greater data reliability, being less susceptible to the problems aforementioned [12].

The programmability of the data plane may present new ways of exploiting the implementation of all existing sampling techniques, requiring only the control plane to stipulate the parameters to be executed through a single controller-switch message. [Deeply Programmable Network \(DPN\)](#) is a recent trend to cover the programmability of both the control and data planes [7]. Programmers are able to increase the switch abilities and, in this case, directly implement sampling techniques on the switch.

Another suggestion of future work is to perform tests with a set of more diversified traffic and even in a real network environment, in order to more comprehensively perceive the applicability of the proposed approach in different network environments with different challenges.

BIBLIOGRAPHY

- [1] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for SDN? Implementation challenges for software-defined networks," *IEEE Communications Magazine*, vol. 51, pp. 36–43, Jul. 2013.
- [2] D. Tammaro, S. Valenti, D. Rossi, and A. Pescapé, "Exploiting packet-sampling measurements for traffic characterization and classification," *International Journal of Network Management*, vol. 22, pp. 451–476, Nov. 2012.
- [3] J. M. C. Silva, P. Carvalho, and S. R. Lima, "A Modular Traffic Sampling Architecture: Bringing Versatility and Efficiency to Massive Traffic Analysis," *Journal of Network and Systems Management*, vol. 25, no. 3, pp. 643–668, 2017.
- [4] M. Moshref, M. Yu, and R. Govindan, "Resource/accuracy tradeoffs in software-defined measurement," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13*, (New York, New York, USA), p. 73, ACM Press, Aug. 2013.
- [5] J. M. C. Silva, "A Modular Traffic Sampling Architecture for Flexible Network Measurements," *Doctoral thesis, Universidade do Minho, Braga, Portugal*, 2015.
- [6] Open Networking Foundation, "SDN Architecture Overview 1.0," May 2013.
- [7] H. Farhady, H. Lee, and A. Nakao, "Software-Defined Networking : A survey," *Computer Networks*, vol. 81, pp. 79–95, Apr. 2015.
- [8] D. Kreutz, F. M. V. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," pp. 1–61, 2014.
- [9] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: taking control of the enterprise," *Sigcomm '07*, pp. 1–12, 2007.
- [10] 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 Issue 2, pp. 69–74, April 2008.
- [11] Open Networking Foundation, "Software-Defined Networking: The New Norm for Networks," *ONF White Paper*, vol. 2, 2012.
- [12] A. Lara, A. Kolasani, and B. Ramamurthy, "Network Innovation using OpenFlow: A Survey," *IEEE Communications Surveys & Tutorials*, vol. PP, no. 99, pp. 1–20, 2013.
- [13] B. Pfaff *et al.*, "OpenFlow Switch Specification v1.1.0." <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.1.0.pdf>, 2011. Retrieved: 2017-10-20.

- [14] Open Networking Foundation, "OpenFlow Switch Specification v1.3.0." <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf>, 2012. Retrieved: 2017-10-20.
- [15] Open Networking Foundation, "OpenFlow Switch Specification v1.5.1." <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf>, 2015. Retrieved: 2017-10-20.
- [16] B. Oliver, "Pica8: First to Adopt OpenFlow 1.4; Why Isn't Anyone Else." <http://www.tomsitpro.com/articles/pica8-openflow-1.4-sdn-switches,1-1927.html>. Retrieved: 2017-10-22.
- [17] Open Networking Foundation, "OpenFlow Switch Specification v1.5.0." <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf>, 2014. Retrieved: 2017-10-20.
- [18] R. Khondoker, A. Zaalouk, R. Marx, and K. Bayarou, "Feature-based Comparison and Selection of Software Defined Networking (SDN) Controllers," in *2014 World Congress on Computer Applications and Information Systems (WCCAIS)*, Jan. 2014.
- [19] Ryu Project Team, "Ryu component-based software defined networking framework." <https://github.com/osrg/ryu>. Retrieved: 2017-10-08.
- [20] Nicira Networks, "The POX Controller." <https://github.com/noxrepo/pox/>. Retrieved: 2017-04-16.
- [21] Y. Takamiya *et al.*, "Trema." <http://trema.github.io/trema/>. Retrieved: 2017-10-22.
- [22] Big Switch Networks, "FloodLight." <http://www.projectfloodlight.org/floodlight/>. Retrieved: 2017-10-22.
- [23] The Linux Foundation, "OpenDayLight." <https://www.opendaylight.org/>. Retrieved: 2017-10-22.
- [24] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky, "Advanced study of SDN/OpenFlow controllers," *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia on - CEE-SECR '13*, no. October, pp. 1–6, 2013.
- [25] Y. Takamiya *et al.*, "Trema." <https://github.com/trema/trema-edge>. Retrieved: 2017-10-22.
- [26] M. B. Al-Somaidai, "Survey of Software Components to Emulate OpenFlow Protocol as an SDN Implementation," *American Journal of Software Engineering and Applications*, vol. 3, no. 6, p. 74, 2014.

- [27] The Linux Foundation, "Open vSwitch." <http://openvswitch.org/>. Retrieved: 2017-10-22.
- [28] P. Emmerich, D. Raumer, S. Gallenmüller, F. Wohlfart, and G. Carle, "Throughput and Latency of Virtual Switching with Open vSwitch: A Quantitative Analysis," *Journal of Network and Systems Management*, pp. 1–25, 2017.
- [29] B. Pfaff *et al.*, "The Design and Implementation of Open vSwitch," *Nsdi*, pp. 117–130, 2015.
- [30] Z. L. Kis *et al.*, "Open vSwitch." <https://github.com/CPqD/ofsoftswitch13>. Retrieved: 2017-10-22.
- [31] Big Switch Networks, "LINC - OpenFlow software switch." <https://github.com/FlowForwarding/LINC-Switch>. Retrieved: 2017-10-22.
- [32] Big Switch Networks, "Indigo Virtual Switch." <http://www.projectfloodlight.org/indigo-virtual-switch/>. Retrieved: 2017-10-22.
- [33] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," *Proceedings of the 8th international conference on Emerging networking experiments and technologies - CoNEXT '12*, p. 253, 2012.
- [34] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop," *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks - Hotnets '10*, pp. 1–6, 2010.
- [35] Mininet Team, "Mininet." <http://mininet.org/>. Retrieved: 2017-10-08.
- [36] The National Science Foundation and Planète group, "ns-3." <http://www.nsnam.org>. Retrieved: 2017-10-22.
- [37] EstiNet Technologies Inc, "EstiNet." <http://www.estinet.com/>. Retrieved: 2017-10-22.
- [38] J. M. C. Silva, P. Carvalho, and S. R. Lima, "Inside packet sampling techniques: exploring modularity to enhance network measurements," *International Journal of Communication Systems*, vol. 30, no. 6, pp. e3135–n/a, 2017. e3135 dac.3135.
- [39] J. M. C. Silva, P. Carvalho, and S. Rito Lima, "A multiadaptive sampling technique for cost-effective network measurements," *Computer Networks*, vol. 57, no. 17, pp. 3357–3369, 2013.
- [40] P. Phaál, S. Panchen, and N. McKee, "InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks rfc 3176 (informational)." <http://www.ietf.org/rfc/rfc3176.txt>, 2001. Retrieved: 2017-10-17.
- [41] P. Phaál, "sFlow blog - Software defined networking." <http://blog.sflow.com/2012/05/software-defined-networking.html>. Retrieved: 2017-10-16.

- [42] K. Giotis, C. Argyropoulos, G. Androulidakis, D. Kalogeras, and V. Maglaris, "Combining OpenFlow and sFlow for an effective and scalable anomaly detection and mitigation mechanism on SDN environments," *Computer Networks*, vol. 62, no. April 2014, pp. 122–136, 2014.
- [43] P. Phaal, "sFlow blog - OpenFlow and sFlow." <http://blog.sflow.com/2011/05/openflow-and-sflow.html>. Retrieved: 2017-10-16.
- [44] J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, a. R. Curtis, and S. Banerjee, "Devoflow: Cost-effective flow management for high performance enterprise networks," *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, p. 1, 2010.
- [45] P. Phaal and M. Lavine, "sFlow Version 5." http://www.sflow.org/sflow_version_5.txt, 2004. Retrieved: 2017-10-17.
- [46] S. Shirali-Shahreza and Y. Ganjali, "Efficient implementation of security applications in OpenFlow controller with FleXam," *Proceedings - IEEE 21st Annual Symposium on High-Performance Interconnects, HOTI 2013*, vol. 2013, pp. 49–54, 2013.
- [47] S. Shirali-shahreza and Y. Ganjali, "FleXam : Flexible Sampling Extension for Monitoring and Security Applications in OpenFlow," *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13*, pp. 167–168, 2013.
- [48] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "A roadmap for traffic engineering in SDN-OpenFlow networks," *Computer Networks*, vol. 71, pp. 1–30, Oct. 2014.
- [49] S. Chowdhury, M. Bari, and R. Ahmed, "PayLess: A Low Cost Network Monitoring Framework for Software Defined Networks," in *14th IEEE/IFIP Network Operations and Management Symposium*, 05 2014.
- [50] J. Casey *et al.*, "Flowgrammable - PacketIn, OpenFlow 1.3.0." http://flowgrammable.org/sdn/openflow/message-layer/packetin/#tab_ofp_1_3. Retrieved: 2017-10-20.
- [51] Ryu Project Team, "OpenFlow Switches." <https://osrg.github.io/ryu/certification.html>, 2014. Retrieved: 2017-10-20.
- [52] IEEE Computer Society, "IEEE Standard for Ethernet, 802.3-2012 - section one," p. 53, 2012.
- [53] Ryu Project Team, "Ryu SDN Framework - Using OpenFlow 1.3," 2014. Retrieved: 2017-10-20.
- [54] J. Casey *et al.*, "Flowgrammable - FlowMod, OpenFlow 1.3.0." http://flowgrammable.org/sdn/openflow/message-layer/flowmod/#tab_ofp_1_3. Retrieved: 2017-10-20.
- [55] J. Casey *et al.*, "Flowgrammable - OpenFlow Message Layer." <http://flowgrammable.org/sdn/openflow/message-layer>. Retrieved: 2017-10-20.
- [56] J. Casey *et al.*, "Flowgrammable - GroupMod, OpenFlow 1.3.0." http://flowgrammable.org/sdn/openflow/message-layer/groupmod/#tab_ofp_1_3. Retrieved: 2017-10-20.

- [57] M. Casado, "List of OpenFlow Software Projects." <http://yuba.stanford.edu/~casado/of-sw.html>. Retrieved: 2017-04-16.
- [58] Ryu Project Team, "Ryu component-based software defined networking framework." https://github.com/osrg/ryu/blob/master/ryu/app/simple_switch_13.py. Retrieved: 2017-10-08.
- [59] S. Finucane, "Why Open vSwitch?." <https://github.com/openvswitch/ovs/blob/master/Documentation/intro/why-ovs.rst>, 2017. Retrieved: 2017-10-19.
- [60] The Linux Foundation, "Open vSwitch with DPDK." <http://docs.openvswitch.org/en/latest/intro/install/dpdk/>. Retrieved: 2017-10-22.
- [61] The Linux Foundation, "DPDK: Data Plane Development Kit – What it is." <http://dpdk.org/>. Retrieved: 2017-10-22.
- [62] The Linux Foundation, "Linux Foundation Wiki - HOWTO setup Linux with PRE-EMPT_RT properly." https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/preemptrt_setup. Retrieved: 2017-10-22.
- [63] C. Shannon, E. Aben, K. C. Claffy, D. Andersen, and N. Brownlee, "The CAIDA UCSD Anonymized Passive OC48 Internet Traces Dataset - 20030424-005000-UTC-anon." http://www.caida.org/data/passive/passive_oc48_dataset.xml.
- [64] C. Shannon, E. Aben, K. C. Claffy, D. Andersen, and N. Brownlee, "The CAIDA UCSD Anonymized Internet Traces 2015 - 20150917-125911-UTC-anon." http://www.caida.org/data/passive/passive_2015_dataset.xml.
- [65] E. Hernandez, M. Chidester, and A. George, "Adaptive sampling for network management," *Journal of Network and Systems Management*, vol. 9, no. 4, pp. 409–434, 2001.
- [66] K. Phemius and M. Bouet, "OpenFlow: Why latency does matter," *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM2013)*, pp. 680–683, 2013.
- [67] S. Shirali-shahreza and Y. Ganjali, "FlexAm : Flexible Sampling Extension for Monitoring and Security Applications in OpenFlow," *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13*, pp. 167–168, 2013.
- [68] B. Pfaff *et al.*, "OpenFlow Switch Specification v1.0.0." <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf>, 2009. Retrieved: 2017-10-20.
- [69] Open Networking Foundation, "OpenFlow Switch Specification v1.2.0." <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.2.pdf>, 2011. Retrieved: 2017-10-20.
- [70] Open Networking Foundation, "OpenFlow Switch Specification v1.4.0." <https://www.opennetworking.org/images/stories/downloads/sdn-resources/>

[onf-specifications/openflow/openflow-spec-v1.4.0.pdf](#), 2013. Retrieved: 2017-10-20.

SDN SAMPLING CONTROLLER

The program presented in this appendix requires Python ≥ 2.7 , the RYU development framework, Open vSwitch, Mininet and OpenFlow 1.3 to be able to run.

The standalone file needed to run the simulation is *Sampling_controller.py*.

A.1 SAMPLING_CONTROLLER.PY

The controller performs sampling measurements once a Mininet topology is set and a valid Packet-In message is received. It was implemented using the existing L2 Learning Switch Controller implementation provided by RYU [58], designed to continuously implement sampling rules and collect samples by using either *SystT*, Adaptive *LP* or *MuST* techniques, and assuming there will be a non-negligible latency in the communication between the measuring point and the program, due to the characteristics of the Mininet environment.

Both adaptive techniques *LP* and *MuST* call the method *predictor* to calculate the next sample estimation based on the *N* previous values.

```
# Copyright (C) 2011 Nippon Telegraph and Telephone Corporation.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
```

```

from ryu.controller.handler import HANDSHAKE_DISPATCHER, DEAD_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib import pcaplib
from ryu import utils
from ryu.lib import hub

class SamplingController(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    MISS_SEND_LENGTH = 1500 # set to match maximum ethernet payload
    DEFAULT_TABLE = 0
    GROUP_ID = 1
    PRIORITY = 50

    def __init__(self, *args, **kwargs):
        super(SamplingController, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.datapaths = {}
        self.group_enabled = False
        self.sampling_enabled = False
        self.pcap_writer = pcaplib.Writer(open('result.pcap', 'wb'))
        self.sample_size = 200.0 # ST (0.1s) ; LP (0.1s); MuST (0.2s)
        self.interval_between_samples = 300.0 # ST (0.5s) ; LP (0.2s); MuST (0.3s)
        self.N = 3 # LP (2); MuST (3)
        self.min_interval_between_samples = 100.0 # LP (0.2s); MuST (0.1s)
        self.max_interval_between_samples = 8000.0 # LP (10s); MuST (8s)
        self.sampling_options = {0: self.systematic_time_based,
                                1: self.linear_prediction,
                                2: self.must}
        self.sampling_technique = 2 # sampling technique selection
        self.min_sample_size = 100.0
        self.max_sample_size = 2000.0
        self.m_min = 0.9, self.m_max = 1.1
        self.nr_packet_in_sampling = 0
        self.nr_packet_in_total = 0
        self.throughput = 0
        self.m_factor = 1.0
        self.nr_samples = 0
        self.k = 0.15
        self.logfile = open('logfile.txt', 'w')

```

```

def add_flow(self, datapath, hard_timeout, priority, match, actions, buffer_id=None,
             table_id=None):
    ofp = datapath.ofproto
    ofp_parser = datapath.ofproto_parser

    inst = [ofp_parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]

    if buffer_id and table_id:
        mod = ofp_parser.OFPFlowMod(datapath=datapath,buffer_id=buffer_id,
                                     table_id=table_id,
                                     priority=priority, hard_timeout=hard_timeout,
                                     match=match, instructions=inst,
                                     flags=ofp.OFPFF_SEND_FLOW_REM)

    elif table_id:
        mod = ofp_parser.OFPFlowMod(datapath=datapath, table_id=table_id,
                                     priority=priority, hard_timeout=hard_timeout,
                                     match=match, instructions=inst,
                                     flags=ofp.OFPFF_SEND_FLOW_REM)

    else:
        mod = ofp_parser.OFPFlowMod(datapath=datapath,priority=priority,
                                     hard_timeout=hard_timeout,match=match,
                                     instructions=inst,
                                     flags=ofp.OFPFF_SEND_FLOW_REM)

    datapath.send_msg(mod)

def del_flow(self, datapath, priority, match, instructions, table_id):
    ofp = datapath.ofproto
    ofp_parser = datapath.ofproto_parser

    mod=ofp_parser.OFPFlowMod(datapath, 0, 0,
                              table_id,ofp.OFPFC_DELETE,0, 0,
                              priority,ofp.OFPCML_NO_BUFFER,ofp.OFPP_ANY,
                              ofp.OFPG_ANY, 0,match, instructions)

    datapath.send_msg(mod)

```

```

def _build_packet_out(self, datapath, buffer_id, src_port, dst_port, data):
    actions = []
    if dst_port:
        actions.append(datapath.ofproto_parser.OFPACTIONOutput(dst_port))

    msg_data = None
    if buffer_id == datapath.ofproto.OFP_NO_BUFFER:
        if data is None:
            return None
        msg_data = data

    out = datapath.ofproto_parser.OFPPacketOut(datapath=datapath,
                                                buffer_id=buffer_id,
                                                data=msg_data, in_port=src_port,
                                                actions=actions)

    return out

def send_packet_out(self, datapath, buffer_id, data, src_port, dst_port=None):
    out = self._build_packet_out(datapath, buffer_id,
                                src_port, dst_port, data)

    if out:
        datapath.send_msg(out)

def flood(self, msg):
    datapath = msg.datapath
    ofp = datapath.ofproto
    ofp_parser = datapath.ofproto_parser
    out = self._build_packet_out(datapath, ofp.OFP_NO_BUFFER,
                                ofp.OFPP_CONTROLLER,
                                ofp.OFPP_FLOOD, msg.data)

    datapath.send_msg(out)
    self.logger.debug("Flooding packet")

def send_group_mod(self, datapath, group_id, action):
    ofp = datapath.ofproto
    ofp_parser = datapath.ofproto_parser

    watch_port = ofproto_v1_3.OFPP_ANY
    watch_group = ofproto_v1_3.OFPQ_ALL

```

```

normal_actions = [ofp_parser.OFPActionOutput(ofp.OFPP_NORMAL, 0)]
sampling_actions = [ofp_parser.OFPActionOutput(ofp.OFPP_CONTROLLER,
                                                ofp.OFPCML_NO_BUFFER)]

if action == 'add':
    buckets = [ofp_parser.OFPBucket(watch_port=watch_port, \
                                    watch_group=watch_group, \
                                    actions=normal_actions),
               ofp_parser.OFPBucket(watch_port=watch_port, \
                                    watch_group=watch_group, \
                                    actions=sampling_actions)]

    req = ofp_parser.OFPGroupMod(datapath=datapath,
                                command=ofp.OFPFC_ADD,
                                type_=ofp.OFPGT_ALL,
                                group_id=group_id,
                                buckets=buckets)

if action == 'delete':
    req = ofp_parser.OFPGroupMod(datapath=datapath,
                                command=ofp.OFPFC_DELETE,
                                type_=ofp.OFPGT_ALL,
                                group_id=group_id)

if action == 'sampling_on':
    buckets = [ofp_parser.OFPBucket(watch_port=watch_port, \
                                    watch_group=watch_group, \
                                    actions=normal_actions),
               ofp_parser.OFPBucket(watch_port=watch_port, \
                                    watch_group=watch_group, \
                                    actions=sampling_actions)]

    req = ofp_parser.OFPGroupMod(datapath=datapath,
                                command=ofp.OFPFC_MODIFY,
                                type_=ofp.OFPGT_ALL,
                                group_id=group_id,
                                buckets=buckets)

if action == 'sampling_off':
    buckets = [ofp_parser.OFPBucket(watch_port=watch_port, \

```



```
        watch_group=watch_group, \
        actions=normal_actions)]

    req = ofp_parser.OFPGroupMod(datapath=datapath,
                                  command=ofp.OFPFC_MODIFY,
                                  type_=ofp.OFPGT_ALL,
                                  group_id=group_id,
                                  buckets=buckets)

    datapath.send_msg(req)
```

```
### Sampling related methods
```

```
def new_sample(self, sample_size):
    # enable sampling
    for dp in self.datapaths.values():
        self.send_group_mod(dp, self.GROUP_ID, 'sampling_on')
    self.sampling_enabled = True

    hub.sleep(sample_size * 0.001)
    self.nr_samples += 1

def new_sampling_interval(self, interval_between_samples):
    # disable sampling
    for dp in self.datapaths.values():
        self.send_group_mod(dp, self.GROUP_ID, 'sampling_off')
    self.sampling_enabled = False
    hub.sleep(interval_between_samples * 0.001)

def predictor(self, vector_x, vector_t):
    i = 1
    n = self.N
    sum = 0.0

    while i < n:
        sum += abs((vector_x[i] - vector_x[i - 1]) / vector_t[i - 1])
        i += 1

    forecast = vector_x[n - 1] + (vector_t[n - 1] / (n - 1)) * sum
    return forecast

def systematic_time_based(self):
    sample_size = self.sample_size
    interval_between_samples = self.interval_between_samples

    while 1:
        throughput_before = self.throughput
        self.new_sample(sample_size)
        throughput_after = self.throughput
        self.new_sampling_interval(interval_between_samples)

        if throughput_before == throughput_after:
            self.log_sampling()
```

```

def linear_prediction(self):
    i = 0
    n = self.N
    vector_x = [0] * n
    vector_t = [0] * n
    sample_size = self.sample_size
    interval_between_samples = self.interval_between_samples
    while i < n:
        throughput_before = self.throughput
        self.new_sample(sample_size)
        throughput_after = self.throughput
        vector_x[i] = throughput_after - throughput_before
        vector_t[i] = interval_between_samples * 0.001
        i += 1
        self.new_sampling_interval(interval_between_samples)

    while 1:
        xp = self.predictor(vector_x, vector_t)
        throughput_before = self.throughput
        self.new_sample(sample_size)
        throughput_after = self.throughput
        s = throughput_after - throughput_before

        m_factor = 1.0
        if s - vector_x[n - 1] != 0:
            m_factor = abs((xp - vector_x[n - 1]) / (s - vector_x[n - 1]))
            m_factor = round(m_factor, 2)
        else:
            interval_between_samples *= 2

        if m_factor < self.m_min:
            interval_between_samples *= m_factor

        elif m_factor > self.m_max:
            interval_between_samples += 1000.0

        if interval_between_samples < self.min_interval_between_samples:
            interval_between_samples = self.min_interval_between_samples
        elif interval_between_samples > self.max_interval_between_samples:
            interval_between_samples = self.max_interval_between_samples

```

```

        i = 0
        j = 1
        while j < n:
            vector_x[i] = vector_x[j]
            vector_t[i] = vector_t[j]
            i += 1
            j += 1
        vector_x[i] = s
        vector_t[i] = interval_between_samples * 0.001

        self.interval_between_samples = interval_between_samples
        self.m_factor = m_factor

        self.new_sampling_interval(interval_between_samples)

        if throughput_before == throughput_after:
            self.log_sampling()

def must(self):
    i = 0
    n = self.N
    vector_x = [0] * n
    vector_t = [0] * n
    sample_size = self.sample_size
    interval_between_samples = self.interval_between_samples

    while i < n:
        throughput_before = self.throughput
        self.new_sample(sample_size)
        throughput_after = self.throughput
        vector_x[i] = throughput_after - throughput_before
        vector_t[i] = interval_between_samples * 0.001
        i += 1

        self.new_sampling_interval(interval_between_samples)

    while 1:
        xp = self.predictor(vector_x, vector_t)
        throughput_before = self.throughput
        self.new_sample(sample_size)
        throughput_after = self.throughput
        s = throughput_after - throughput_before

```

```

m_factor = 1.0
if s:
    m_factor = xp / s
    m_factor = round(m_factor, 2)

if m_factor < self.m_min:
    interval_between_samples *= m_factor
    sample_size *= m_factor

elif m_factor > self.m_max:
    interval_between_samples *= 2
    sample_size *= (1 + self.k)

if interval_between_samples < self.min_interval_between_samples:
    interval_between_samples = self.min_interval_between_samples
elif interval_between_samples > self.max_interval_between_samples:
    interval_between_samples = self.max_interval_between_samples

if sample_size < self.min_sample_size:
    sample_size = self.min_sample_size
elif sample_size > self.max_sample_size:
    sample_size = self.max_sample_size

if s:
    i = 0
    j = 1
    while j < n:
        vector_x[i] = vector_x[j]
        vector_t[i] = vector_t[j]
        i += 1
        j += 1
    vector_x[i] = s
    vector_t[i] = interval_between_samples * 0.001

self.interval_between_samples = round(interval_between_samples,0)
self.sample_size = round(sample_size,0)
self.m_factor = m_factor

self.new_sampling_interval(interval_between_samples)
if throughput_before == throughput_after:
    self.log_sampling()

```

```

def log_sampling(self):
    if self.sampling_technique == 0:
        print >> self.logfile, self.nr_samples, self.nr_packet_in_sampling, \
            self.nr_packet_in_total, self.sample_size, \
            self.interval_between_samples, self.throughput

        self.logger.debug('nr_samples: %d\n'
            'nr_packet_in_sampling: %d, nr_packet_in_total: %d\n'
            'sample_size: %.2f, interval_between_samples: %.2f\n'
            'throughput: %d\n',
            self.nr_samples,
            self.nr_packet_in_sampling, self.nr_packet_in_total,
            self.sample_size, self.interval_between_samples,
            self.throughput)

    else:
        print >> self.logfile, self.m_factor, self.nr_samples, self.nr_packet_in_samp
            self.nr_packet_in_total, self.sample_size, \
            self.interval_between_samples, self.throughput

        self.logger.debug('nr_samples: %d, m_factor: %.2f \n'
            'nr_packet_in_sampling: %d, nr_packet_in_total: %d\n'
            'sample_size: %.2f, interval_between_samples: %.2f\n'
            'throughput: %d\n',
            self.nr_samples, self.m_factor,
            self.nr_packet_in_sampling, self.nr_packet_in_total,
            self.sample_size, self.interval_between_samples,
            self.throughput)

```

```
### Events handlers
```

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    ofproto = msg.datapath.ofproto
    reason = msg.reason

    if msg.msg_len < msg.total_len:
        self.logger.info("packet truncated: only %s of %s bytes", \
                        ev.msg.msg_len, ev.msg.total_len)

    self.nr_packet_in_total += 1

    if self.sampling_enabled:
        self.pcap_writer.write_pkt(msg.data)
        self.throughput += msg.msg_len
        self.nr_packet_in_sampling += 1

    if reason != ofproto.OFPR_ACTION:
        datapath = msg.datapath
        ofp_parser = datapath.ofproto_parser

        in_port = msg.match['in_port']
        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocols(ethernet.ethernet)[0]
        dst = eth.dst
        src = eth.src
        dpid = datapath.id
        self.mac_to_port.setdefault(dpid, {})

        self.logger.debug('packet in %s %s %s %s %s', dpid, src, dst, \
                        in_port, reason)
        self.logger.debug('OFPPacketIn received: '
                        'buffer_id=%x total_len=%d'
                        'table_id=%d match=%s data=%s',
                        msg.buffer_id, msg.total_len,
                        msg.table_id, msg.match,
                        utils.hex_array(msg.data))

    # learn a mac address to avoid FLOOD next time.
    self.mac_to_port[dpid][src] = in_port
```

```

    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD

    if out_port != ofproto.OFPP_FLOOD:
        if self.group_enabled is False:
            # inject group into the switch
            self.send_group_mod(datapath, self.GROUP_ID, 'add')
            self.logger.info('Group %s injected', self.GROUP_ID)

            # start sampling thread
            self.sampling_thread = \
                hub.spawn(self.sampling_options[self.sampling_technique])
            self.group_enabled = True

        actions = [ofp_parser.OFPActionGroup(group_id=self.GROUP_ID)]
        match = ofp_parser.OFPMatch(in_port=in_port, eth_dst=dst)

        if msg.buffer_id != ofproto.OFP_NO_BUFFER:
            self.add_flow(datapath, 0, self.PRIORITY, match, actions, \
                           msg.buffer_id, self.DEFAULT_TABLE)
        else:
            self.add_flow(datapath, 0, self.PRIORITY, match, actions, \
                           self.DEFAULT_TABLE)

        self.send_packet_out(datapath, msg.buffer_id, msg.data, in_port)
    else:
        self.flood(msg) # flood the packet

@set_ev_cls(ofp_event.EventOFPErrormsg, [HANDSHAKE_DISPATCHER, CONFIG_DISPATCHER, \
                                           MAIN_DISPATCHER])
def error_msg_handler(self, ev):
    msg = ev.msg
    self.logger.debug('OFPErrormsg received: type=0x%02x code=0x%02x '
                      'message=%s', msg.type, msg.code,
                      utils.hex_array(msg.data))

```



```

@set_ev_cls(ofp_event.EventOFPStateChange,
            [MAIN_DISPATCHER, DEAD_DISPATCHER])
def _state_change_handler(self, ev):
    datapath = ev.datapath
    if ev.state == MAIN_DISPATCHER:
        if not datapath.id in self.datapaths:
            self.logger.debug('register datapath: %016x', datapath.id)
            self.datapaths[datapath.id] = datapath
    elif ev.state == DEAD_DISPATCHER:
        if datapath.id in self.datapaths:
            self.logger.debug('unregister datapath: %016x', datapath.id)
            del self.datapaths[datapath.id]

@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
    datapath = ev.msg.datapath
    self.datapath = datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    # install table-miss flow entry
    match = parser.OFPMatch()
    actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER, \
                                      ofproto.OFPCML_NO_BUFFER)]

    # Configure datapath with custom MISS_SEND_LENGTH
    self.send_set_config(datapath)

    self.add_flow(datapath, 0, 0, match, actions, table_id=self.DEFAULT_TABLE)

@set_ev_cls(ofp_event.EventOFPGetConfigReply, MAIN_DISPATCHER)
def get_config_reply_handler(self, ev):
    msg = ev.msg
    dp = msg.datapath
    ofp = dp.ofproto
    flags = []

    if msg.flags & ofp.OFPC_FRAG_NORMAL:
        flags.append('NORMAL')
    if msg.flags & ofp.OFPC_FRAG_DROP:

```

```

        flags.append('DROP')
    if msg.flags & ofp.OFPC_FRAG_REASM:
        flags.append('REASM')
    self.logger.debug('OFPPGetConfigReply received: '
                      'flags=%s miss_send_len=%d',
                      ', '.join(flags), msg.miss_send_len)

@set_ev_cls(ofp_event.EventOFPPFlowRemoved, MAIN_DISPATCHER)
def flow_removed_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofp = datapath.ofproto
    ofp_parser = datapath.ofproto_parser

    if msg.reason == ofp.OFPRR_IDLE_TIMEOUT:
        reason = 'IDLE TIMEOUT'
    elif msg.reason == ofp.OFPRR_HARD_TIMEOUT:
        reason = 'HARD TIMEOUT'
    elif msg.reason == ofp.OFPRR_DELETE:
        reason = 'DELETE'
    elif msg.reason == ofp.OFPRR_GROUP_DELETE:
        reason = 'GROUP DELETE'
    else:
        reason = 'unknown'

    field = []
    for f in msg.match.fields:
        field.append('%s: value=%s' % (f.__class__.__name__, f.value))

    self.logger.debug('OFPPFlowRemoved received: '
                      'cookie=%d priority=%d reason=%s table_id=%d '
                      'duration_sec=%d duration_nsec=%d '
                      'idle_timeout=%d hard_timeout=%d '
                      'packet_count=%d byte_count=%d match.fields=%s',
                      msg.cookie, msg.priority, reason, msg.table_id,
                      msg.duration_sec, msg.duration_nsec,
                      msg.idle_timeout, msg.hard_timeout,
                      msg.packet_count, msg.byte_count, field)

```

```

@set_ev_cls(ofp_event.EventOFPPortStatus, MAIN_DISPATCHER)
def port_status_handler(self, ev):
    msg = ev.msg
    dp = msg.datapath
    ofp = dp.ofproto

    if msg.reason == ofp.OFPPR_ADD:
        reason = 'ADD'
    elif msg.reason == ofp.OFPPR_DELETE:
        reason = 'DELETE'
    elif msg.reason == ofp.OFPPR_MODIFY:
        reason = 'MODIFY'
    else:
        reason = 'unknown'
        self.logger.debug('OFPPortStatus received: reason=%s desc=%s', reason, msg.de

### Other methods

def send_get_config_request(self, datapath):
    ofp_parser = datapath.ofproto_parser
    req = ofp_parser.OFPGetConfigRequest(datapath)
    datapath.send_msg(req)

def send_set_config(self, datapath):
    ofp = datapath.ofproto
    ofp_parser = datapath.ofproto_parser
    req = ofp_parser.OFPSetConfig(datapath, ofp.OFPC_FRAG_NORMAL, \
                                   self.MISS_SEND_LENGTH)
    datapath.send_msg(req)

def send_port_desc_stats_request(self, datapath):
    ofp_parser = datapath.ofproto_parser
    req = ofp_parser.OFPPortDescStatsRequest(datapath, 0)
    datapath.send_msg(req)

```

MESSAGE FLOW TO IMPLEMENT SAMPLING MECHANISMS

Example of communication in the suggested architecture, in terms of OpenFlow messages triggered exclusively for the sampling mechanism. The first Packet-In message is triggered by a table miss [14]. The following packets are requested by the controller through an explicit action.

Figure B.1: Group Mod and Packet-In messages exchanged between controller and switch

COMMANDS TO TEST THE ARCHITECTURE

Commands for testing the proposed architecture are listed here.

Start Open vSwitch with DPKD:

```
#!/bin/bash
export PATH=$PATH:/usr/local/share/openvswitch/scripts
export DB_SOCK=/usr/local/var/run/openvswitch/db.sock

ovsdb-server --remote=punix:/usr/local/var/run/openvswitch/db.sock \
              --remote=db:Open_vSwitch,Open_vSwitch,manager_options \
              --private-key=db:Open_vSwitch,SSL,private_key \
              --certificate=db:Open_vSwitch,SSL,certificate \
              --bootstrap-ca-cert=db:Open_vSwitch,SSL,ca_cert \
              --pidfile --detach
ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-init=true
ovs-ctl --no-ovsdb-server --db-sock="$DB_SOCK" start
```

Start Ryu controller:

```
#!/bin/bash
ryu-manager ryu/app/sampling_controller.py #--verbose
```

Define the topology in Mininet:

```
#!/bin/bash
sudo mn --topo single,2 --mac --controller remote --switch ovs
```

Extra settings to define maximum disk and CPU access priority:

```
ionice -c 2 -n 0 nice -n -20 ./sampling_controller.sh
ionice -c 2 -n 0 nice -n -20 ./topology.sh
```

Send the traffic of host h1 through interface h1-eth0:

```
mininet> h1 tcpreplay -i h1-eth0 oc48_new.pcap
```

OPENFLOW VERSION CHANGELOG

The notes here described about the features available in different versions of the OpenFlow protocol were made available by [ONF](#). They are transcripts of these changes.

OpenFlow version 0.2.0 [\[15\]](#)

Release date: 28 March, 2008

Wire Protocol: 1

OpenFlow version 0.2.1 [\[15\]](#)

Release date: 28 March, 2008

Wire Protocol: 1

No changes were made in this version.

OpenFlow version 0.8.0 [\[15\]](#)

Release date: 5 May, 2008

Wire Protocol: 0x83

- OpenFlow message types were reorganised;
- OFPP_TABLE virtual port added providing ability to send Packet-Out packets to the tables;
- Global flag OFPC_SEND_FLOW_EXP added to configure flow expired messages generation;
- Flow priority added in this version;
- Flow Group-ID, for experimental QoS support, removed;
- Error messages added;
- More generic stat request and stat reply messages, with a generic header and stat specific body;

- Fragmentation strategy changed for stats reply, by using explicit flag OFPSF_REPLY_MORE instead of empty packet;
- Table stats and port stats messages added.

OpenFlow version 0.8.1 [15]

Release date: 20 May, 2008

Wire Protocol: 0x83

No changes were made in this version.

OpenFlow version 0.8.2 [15]

Release date: 17 October, 2008

Wire Protocol: 0x85

- Echo Request and Echo Reply messages added;
- All messages are 64 bits aligned.

OpenFlow version 0.8.9 [15]

Release date: 2 December, 2008

Wire Protocol: 0x97

- Flow entries may contain IP subnet masks;
- New physical port statistics - ofp_port_stats message returns more statistical information;
- Behaviour of sending out the incoming port is now better defined, being forbidden unless if the output port is explicitly set to OFPP_IN_PORT;
- Port and link status and configuration changed - the switch must inform the controller of changes to port and link status;
- Echo Request/Reply messages have been created so the controller and the switch can verify the connectivity between each other;
- Vendor extensions added with the purpose of allowing the creation of extensions compliant with the OpenFlow protocol. This can be achieved using the new OFPT_VENDOR message type;

- The switch is now able to communicate to the controller whether it is able to handle IP fragments;
- OpenFlow can now configure and view results of on-switch 802.1D Spanning Tree Protocol implementations;
- Commands to support modifying the actions of existing entries have been added;
- Table description is more flexible with `n_tables`, which describes the lists of tables in the switch;
- Lookup count in tables return the number of packets that have been looked up in the table, whether they hit or not;
- Mask field added containing a bit set for each flag that should be changed on the switch, when modifying characteristics of a switch's ports, with the Port-Mod message;
- New Packet-Out message format. The packet always has a list of actions, followed by an additional variable-length array with the contents of the packet if `buffer_id` is not set;
- Hard time added for flow entries. If set, then the entry must expire in the specified number of seconds regardless of whether or not packets are hitting the flow entry;
- Initial handshake reworked to support backwards compatibility, by including version negotiation capability in a `OFPT_HELLO` message, that includes the higher version supported by the sender;
- Description of switch statistics added, such as hardware and software running on the switch;
- Vendor-defined actions added to OpenFlow. Actions switched from fixed-length to variable, to provide more versatility;
- VLAN Action changed. Possibility to set the priority field in VLAN tags and stripping VLAN tags became a separate action;
- The maximum number of supported ports increased to 65280, to support large vendor switches. Previous value was 256;
- The switch now sends an error message when a flow is not added because all the tables are full;
- Common behaviour defined when the switches lose connection to the controller;
- Matchable ICMP type and code Fields to allow matching ICMP traffic based on type or code;
- Support added for listing and deleting entries based on an output port.

OpenFlow version 0.9.0 [15]

Release date: 20 July, 2009

Wire Protocol: 0x98

- Simple failover mechanism included;
- Protocol extended to allow insertion and management of emergency flow entries;
- Barrier Command added - a mechanism to get notified when an OpenFlow message has finished executing on the switch;
- Match on VLAN Priority Bits - optional new feature for matching on priority VLAN fields ;
- Per-flow expiration messages can now be requested;
- Flow Mod now has a CHECK_OVERLAP flag to check for conflicts during flow insertion or modification;
- Flow expiration duration - value returned is the time in which the flow was active and not include any timeout period;
- If the notification bit is set when a controller deletes a flow, it now receives a notification;
- Flow action added to rewrite the DiffServ CodePoint bits part of the IP ToS field in the IP header, for basic QoS support;
- Port numbers now start at 1.

OpenFlow version 1.0 [68]

Release date: 31 December, 2009

Wire Protocol: 0x01

- OpenFlow now supports multiple queues per output port;
- Flows have been extended to include an opaque identifier, referred to as a cookie;
- The switch description reply now includes a datapath description field;
- The reference implementation can now match on IP fields inside ARP packets;
- Supports for matching on the IP ToS/DSCP bits;
- Port stat request messages include a port_no field to allow stats for individual ports to be queried;
- Flow durations in stats and expiry messages were improved to be expressed with nanosecond resolution.

OpenFlow version 1.1 [13]

Release date: 28 February, 2011

Wire Protocol: 0x02

- OpenFlow pipeline support for multiple tables added;
- New group abstraction to represent a set of ports as a single entity for forwarding packets;
- New tagging support - explicit actions to add, modify and remove VLAN tags. Support for multiple levels of VLAN tagging;
- Support for virtual ports, that can represent complex forwarding abstractions such as LAGs or tunnels;
- Two modes to deal connectivity issues with the controller added. In fail secure mode, the switch continues to operate in OpenFlow mode, until it reconnects to a controller. In fail standalone mode, the switch reverts to using normal processing (Ethernet switching).

OpenFlow version 1.2 [69]

Release date: 5 December, 2011

Wire Protocol: 0x03

- Extensible match support has been added - The ofp_match has been changed to a [TLV](#) structure, called [OXM](#), to increase flexibility;
- Extensible set_field action reuse of the [OXM](#) encoding defined for matches. It permits the rewriting of any header field in a single action;
- [OXM](#) encoding is used to carry packet context in Packet-In messages;
- Experimenter error code has been added, enabling experimenter functionality to generate custom error messages;
- Basic support for IPv6 match and header rewrite has been added;
- Flow-mod requests simplified behaviour;
- Packet parsing specification was removed;
- Controller role change mechanism added - simple mechanism to support multiple controllers for failover.

OpenFlow version 1.3 [14]

Release date: 13 April, 2012

Wire Protocol: 0x04

- Improved description of table capabilities, moved out of the table statistics structure in its own request/reply message, and encoded using a flexible [TLV](#) format;
- Flexible table miss support - table-miss flow entry uses standard OpenFlow instructions and actions to process table-miss packets. OpenFlow's full flexibility in processing those packets is achieved this way;
- Ability to match the presence of common IPv6 extension headers, and some anomalous conditions in IPv6 extension headers;
- Per-flow meters support added;
- Enabled a condition to filter connection events;
- A switch can now create auxiliary connections to supplement the main connection between switch and controller;
- A new OXM field has been added to match the [Bottom of Stack bit \(BOS\)](#) from the [Multiprotocol Label Switching \(MPLS\)](#) header;
- Support added for tagging packets using [Provider Backbone Bridging \(PBB\)](#) encapsulation;
- Tag order reworked - the final order of tags in a packet is dictated by the order of the tagging operations;
- Tunnel-ID metadata included - support for a wide variety of encapsulations;
- Cookie field added to the Packet-In message;
- Duration field included to most statistics;
- Flow Mod flags have been added to disable packet and byte counters on a per-flow basis.

OpenFlow version 1.4 [\[70\]](#)

Release date: 5 August, 2013

Wire Protocol: 0x05

- Wire protocol flexibility - The introduction of the [OXM](#) in OpenFlow 1.2 added extensibility in the OpenFlow classifier. In version 1.4, other parts of the protocol were changed to [TLV](#) structures for improved extensibility;
- Packet-In message contains more descriptive reasons;
- New set of port properties to add support for optical ports;
- Flow-removed reason added when a meter is deleted;

- Flow monitoring framework - a controller may now define a number of monitors to select subsets of flows;
- Role status message that enables the switch to inform about changes to its role to a controller;
- Eviction mechanism - the switch can automatically eliminate entries of lower importance to make space for newer entries;
- Vacancy events - mechanism enabling the controller to receive an early warning based on a defined capacity threshold;
- Bundle mechanism - apply a group of OpenFlow message as a single operation;
- Synchronised tables - feature allowing to represent several lookups as a set of two tables whose data is synchronised;
- Group and Meter change notifications - allow a controller to monitor in real-time the modifications to the group table or meter table done by other controllers;
- Several new error codes introduced - to indicate bad priority, an error during Set-async-config, duplicate instructions or multipart timeout;
- New [OXM](#) field added to match the “use customer address” header field from the [PBB](#) header;
- TCP default port changed to 6653.

OpenFlow version 1.5 [\[17\]](#)

Release date: December, 2014

Wire Protocol: `oxo6`

- Egress Tables - feature that enables the processing in the context of the output port;
- Packet aware pipeline, enabling processing for other types of packets, such as [IP](#) or [Point-to-Point Protocol \(PPP\)](#) packets;
- [OpenFlow eXtensible Statistics \(OXS\)](#) added to encode any arbitrary flow entry statistics;
- Creation of flow entry statistics trigger - mechanism that enables statistics to be automatically sent to the controller based on various statistics thresholds;
- Packet Registers - a set of pipeline fields designed to be used as generic scratch space for the Copy-Field action;
- [OXM](#) field added to match the flag bits in the TCP header;
- Two new group commands added to enable the insertion or removal of specific group buckets without causing impact in the other group buckets in the group;

- Set-field action allowed to set metadata field (register value used to carry information from one table to the next) and associate wildcards;
- Bundle features extended - schedules bundles and bundle features request added;
- Controller Connection Status - the controller can know the status of all connections;
- Replaced meter instruction by a meter action;
- Support setting all pipeline fields of the Packet-in and Packet-Out messages;
- Port properties for pipeline fields and recirculation added;
- Barrier message improvements - controls message ordering, notifies of completed processing and commits states to the datapath;
- Port status message is sent to the controllers when port configuration changes, including those done by one OpenFlow controller;
- Standard for all experimenter [OXM](#)-IDs set to 64 bits;
- Use of a common structure for group, port and queue multipart requests.