Study of the cost of measuring virtualized networks

Karyna Gogunska

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Etude du coût de mesure des réseaux virtualisés

Karyna GOGUNSKA

Inria Sophia Antipolis

Présentée en vue de l'obtention du grade de docteur en INFORMATIQUE d'Université Côte d'Azur
Dirigée par : Chadi Barakat / Guillaume Urvoy-Keller
Soutenue le : 18/12/2019

Devant le jury, composé de :
Hossam AFIFI, Professeur, Télécom SudParis
André-Luc BEYLOT, Professeur, IRIT/ENSEEIHT
Pietro Michiardi, Professeur, Eurecom
Chadi Barakat, Directeur de Recherche (DR), Inria
Guillaume Urvoy-Keller, Professeur, I3S/CNRS

Inria

Etude du coût de mesure des réseaux virtualisés

Jury :

Rapporteurs

Hossam AFIFI, Professeur, Télécom SudParis
André-Luc BEYLOT, Professeur, IRIT/ENSEEIHT

Examinateur

Pietro Michiardi, Professeur, Eurecom

Directeur

Chadi Barakat, Directeur de Recherche, Inria

Co-directeur

Guillaume Urvoy-Keller, Professeur, I3S/CNRS

Resume

La tendance actuelle dans le développement et le déploiement d'applications consiste à les embarquer dans des machines virtuelles ou des conteneurs. Il en résulte une combinaison de configurations de réseaux physiques et virtuels combinant des commutateurs virtuels et physiques avec des protocoles spécifiques pour créer des réseaux virtuels s'étendant sur plusieurs serveurs. Ce nouvel environnement constitue un défi lorsqu'il s'agit de mesurer et de debuguer les problèmes liés aux performances.

Dans cette thèse, nous examinons le problème de la mesure du trafic dans un environnement virtuel et nous nous concentrons sur un scénario typique : des machines virtuelles interconnectées par un commutateur virtuel. Nous avons étudié le coût de la mesure en continu du trafic réseau des machines. Plus précisément, nous avons évalué le cout du partage du substrat physique entre la tâche de mesure et l'application. Nous avons dans un premier confirmé l'existence d'une corrélation négative entre la mesure et le trafic applicatif.

Dans une seconde partie de la thèse, nous avons orienté notre travail vers une minimisation de l'impact des mesures en utilisant des techniques d'apprentissage automatiques en temps réel. Nous avons proposé une solution basée sur les données, capable de fournir des paramètres de surveillance optimaux pour les mesures de réseau virtuel avec un minimum d'interférence pour le trafic applicatif.

Mot-clé: Réseau virtualisé, Mesure, Surveillance, Open vSwitch, Centre de Données, Apprentissage Machine, sFlow, IPFIX

Abstract

The current trend in application development and deployment is to package applications within containers or virtual machines. This results in a blend of virtual and physical resources with complex network setups mixing virtual and physical switches along with specific protocols to build virtual networks spanning over several servers. While this complexity is often hidden by cloud management solutions, such new environment constitutes a challenge when it comes to monitor and debug performance-related issues. In this thesis, we consider the problem of measuring traffic in a virtualized environment and focus on one typical scenario: virtual machines interconnected with a virtual switch. We assess the cost of continuously measuring the network traffic of the machines. Specifically, we seek to estimate the competition that exists to access the resources (e.g., CPU) of the physical substrate between the measurement task and the application. We confirm the negative correlation of measurement within such setup and propose actions towards its minimization. Concluding on the measurement interference with virtual network, we then turn our work towards minimizing its presence in the network. We assess the capability of machine learning techniques to predict the measurement impact on the ongoing traffic between virtual machines. We propose a data-driven solution that is able to provide optimal monitoring parameters for virtual network measurements with minimum traffic interference.

Keywords: Virtual Network, Measurement, Monitoring, Open vSwitch, Data Center, Machine Learning, sFlow, IPFIX

Acknowledgements

This thesis summarizes my research results during my PhD study at the research team DIANA, Inria, France from 2016 to 2019.

First of all, I express my gratitude to my advisors Prof. Guillaume Urvoy-Keller and Dr. Chadi Barakat, who walked me through these three years and contributed to my development as a young scientist. Thank you for putting your effort, time and patience in me, so that I end up at this point.

I sincerely thank all my colleagues and friends in SigNet and DIANA team. Being a part of Inria and I3S let me meet great people from all over the world, who brightened up my days in France, especially Dr. Dimitra Politaki, Dr. Osama Arouk, Dr. Mathijs Wintraeken, Lyes Khacef, Dimitra Tsigkari, Dr. Tingting Yuan, Dr. Zeineb Guizani, Dr. Olexandra Kulankhina, Thierry Spetebroot, Dr. Naoufal Mahfoudi, Dr. Hardik Soni.

I thank Christine Foggia for her great competence and immense help in a variety of subjects.

I thank Prof. Deep Medhi for showing his interest and questioning my research. It is amazing, how meeting him once per year was enough to boost my interest in science.

Special thank goes to Dr. Damien Saucez. Thank you for your timely support and help, conversations, lunches, hikings, experiences and knowledge.

Finally, I dedicate this thesis to Dr. Vitalii Poliakov, my favorite person in the world, my inspiration and my role model. Thank you for believing in me, for lifting up my morale level for so many times, for rising up my confidence to pursue the objective of becoming a PhD and for being at my side since 2012.

Table of contents

List of figures

List of tables

Chapter 1

Overview

1.1 Network softwarization

The evolution of the Internet and its ubiquitous adoption in nearly every level of our society have opened a whole new domain for communication providers. Their interest in profit motivates the invention of new technologies and services for individual users just as well as for the enterprises. Internet by its very definition is a large-scale, global network. While it was arguably easier to organize such a network in its early days, when the subscribers were not-so-numerous and rather predictable in their behaviour, the newly-arrived Internet Service Providers (ISPs) started to struggle scaling up their facilities for huge crowds of users when the Internet stepped into its late stages of adoption. At a certain point, such networks became subject to scalability constraints and problematic maintenance of numerous protocols within purpose-built networking devices, that have to be configured one by one, requiring physical presence at the site. Enterprises networks started organizing into logically separated networks within their facilities given by the Infrastructure Provider (InPs). Meanwhile, Service Providers (SPs) were aggregating resources from multiple InPs and offering end-to-end services. For the rapidly developing IT-market, there appeared a need to share physical resources in an efficient manner while ensuring isolation. This was the reasoning invention of virtualization, hypervisors and containerization systems. Additionally, the aforementioned issues with legacy networks lead in a part to the appearance of network softwarization and virtualization, which aimed at breaking the chains of fixed-role network architectures and allow more flexibility in network design.

In general, virtualization is used to transform available hardware into a software environment. It first appeared to serve the purpose of shared computer resources among a large group of users and brought multiple advantages since then to the computer science domain. Virtualization has an important role in cloud computing technologies. Virtualization is a kind of abstraction to make software object or resource to behave and look like a hardware-implemented one, but with significant advantages in flexibility, cost, efficiency, scalability and wide range of applications and general capabilities. With the advent of virtualization technologies and subsequent discovery of their advantages, Internet research and industries have ventured to study and experiment with virtualizing every component of a computing system – from the computers as a whole (Virtual Machine, VM) to its individual subsystems (storage, networking, and so on).

A virtual machine is a software imitation of a computer system with its imitated hardware (processor memory, hard disk, etc.), which is based on another operating system built upon a hardware entity. One could think of it as a kind of a software duplicate of a real machine [1]. Virtual machines are not aware of the underlying system where they are deployed, and they can only use as much physical resources as dedicated by the underlying host operating system, which ensures isolation of resources and prevents any interference of co-hosted VMs and native system. The functionality of VMs is equal to the functionality of its operating system. It can implement certain applications, functions or services, and act as a programming emulator or simulator. Cloud providers are direct beneficiaries of virtualization, as they deliver hosted applications and services over the Internet. Such applications and services can be accessed from across the globe thanks to the cooperation of virtualization and networking. Virtualization also helps cloud service providers in achieving isolation and resource multiplexing.

Network virtualization combines resources of physical servers and networks into software networks between virtual machines. These software networks share the same underlying physical infrastructure and become an efficient tool for new application deployment, accessibility, and scalability of resources and services deployed in such virtual environments, as well as automation and energy saving. Networking functions and networking equipment can be softwarized as well. Network function virtualization (NFV) and software switches took the functionality of physical devices and now enable Internet traffic processing by software programs in order to accelerate deployment of network services and cope with demanding and rapidly changing needs from network users and administrators. Network under virtual-

ization becomes a sophisticated system where high-level applications are deployed inside virtualized operating systems (VMs), which can be organized into a virtual network with software switches and software network functions. All these virtual components are based on a host operating system (hypervisors and cloud management systems like OpenStack [2]), that handles this virtual domain plus the whole operation of a hardware server, which is only a single node in a typically large network alongside with other servers and network equipment. This is what a typical data center would look like nowadays.

In the quest of searching for simple and flexible network administration, the concept of Software-Defined Networking attracted the attention of the Internet and cloud actors. Software-Defined Networking (SDN) is a paradigm of decoupling network control and management from traffic forwarding. In a network that is implemented using the SDN paradigm, an administrator can now issue network administration and configuration from a centralized controller, which has access to all network devices. Compared to the traditional, individual administration approach, the required device configurations are applied, necessary policies are set, the forwarding rules are issued in a more transparent, efficient and less time-consuming manner. In such a way, administration operations are communicated to the devices not directly but using an application programming interface (API), such as OpenFlow [3] and Cisco OpFlex [4], which provides better visibility, efficiency, and flexibility to the network management.

Virtualization, SDN, and clouds are uniting towards the goal of IT softwarization, which relies on software programs to satisfy the demands of network services. The interoperation of such solutions inevitably leads to increasing the overall complexity of the entire system. On the other hand, the complexity of such a system is justified by the level of technological progress achieved and the profit obtained.

This thesis will present our findings as to how to ensure proper monitoring of one of the crucial components of this united system: the network subsystem.

1.2 Software network monitoring

Almost any kind of a sizeable business nowadays makes an extensive use of networking technologies. Running such a business is often associated with ensuring the integrity and security of the commercial data, as well as guaranteeing timely delivery of the product. If

we apply this observation to an Internet services-related business, the product becomes the service itself. In such a case it becomes crucial to have extended visibility into the operation of the internal network, as this is what defines the client agreement compliance (Service Level Agreement, SLA) and other important business goals.

Depending on the size and complexity of the network, achieving such visibility can become difficult already due to the amount of incoming information. When we start considering softwarization and virtualization in the networks, management, and monitoring gain a whole new dimension of complexity. As a brief example, let us consider a virtual network client and the way its network traffic can make before reaching the destination: after being issued from its network socket, traffic will have to traverse software network equipment (e.g., a software switch on the host operating system), then exit virtual networking interface, entering physical equipment and interfaces, isolated by means of VXLAN (virtual extensible local access network) tunneling, leaving the local network and travel throughout the physical network – which can make use of NFV and, therefore, involve other virtualized segments. Our virtual client's traffic monitoring now includes monitoring of virtual network (equipment and traffic) and virtual hosts, plus the monitoring of the aforementioned software and hardware parts of the virtualization platform, which will include monitoring of the underlying network as well. With the increasing use of virtualization, the number of measurement endpoints can grow quickly.

Monitoring the virtualization platform is vital for providers to assure a decent level of services, yet it is complex and challenging, as now it includes both monitoring the software component (that enables virtualization), and physical component where it is deployed. Twofold amount of problems has to be detected and handled: software faults may be results of improper configurations just as well as hardware issues, or an insufficient amount of system resources can lead to obscured failures of virtualization (and not only) applications. Great effort has to be devoted to monitoring, observing all aspects of the virtual environment, preventing, detecting, localizing and resolving problems. Aiming to propose the best monitoring tools, the state-of-the-art solutions for network monitoring often contain investigations of the amount of resources they consume (network- and/or systems-wise) [5], [6], [7], [8], [9]. Even though it seems difficult to monitor an infrastructure seamlessly and without a trace, modern research continues to excel at minimizing monitoring overheads. Researches consider different kinds of the effect of their monitoring frameworks within an evaluation setup (delay-. report-, bandwidth-, memory-wise, etc.). Yet, the evaluation phase may not

reveal all pitfalls of such effects. To obtain a full picture, the monitoring tool would have to be checked in operation with all its possible configurations within all possible setups. It is problematic to anticipate under which setup (number of VMs, amount of traffic, the criticality of application SLAs) the tool will be used. Also, it may be quite challenging to perform tests within all the configuration scenarios. Nonetheless, it would be beneficial for end-user, DevOps and network/system administrators to obtain a general idea of monitoring tool behaviour in the setup with respect to its parameters, its application, virtual network components operation (VMs, their application performance, software switch), and hardware components (server and network performance). Throughout the state-of-the-art multiple proposals of monitoring techniques and methods exist, that we have reviewed in Chapter 2. Yet, there was no explicit exploration of the scope of the footprint of monitoring tool parameters at work in the virtual network setup.

1.3 Problem statement

Driven by the aforementioned two-fold kind of problems and shortage of its study, we have ventured to shed more light on the monitoring processes in virtual networks and their effect onto operation of the virtual network themselves, as well as their physical hardware – in terms of system resource consumption. More specifically, the main question is whether and to which extent certain monitoring solutions, that run over a virtual network, affect the performance in terms of system and, in its turn, network resources.

Unfortunately, so far there exists no universal all-purpose monitoring framework, that could be applied to all virtualization platforms (or, at least, a broad range of them), while comprising traditional and virtual networks, devices and applications, and covering all kinds of needs for users and operators at the same time. That is why numerous monitoring solutions are developed for specific needs of certain platforms, often poorly applicable in different setups: some works are developed for use with the most popular platforms, however, their claim to be cross-platform compatible and deployable is rarely performed and verified, e.g., [10]. Such a shortage of cross-platform monitoring tools makes it difficult for cloud providers and users to efficiently manage their monitoring tasks for virtualized networks. Having a better insight on the effect of monitoring onto physical resources of the hosting hardware has

the potential to help them better estimate and adapt the existing monitoring tools for their specific needs.

Understanding the monitoring footprint in this scenario helps to achieve more efficient resource management, thus improving reliability and decreasing operational costs of the system. By knowing the nature and extent of the correlation between monitoring and the monitored system in a virtualized environment, one could gain an opportunity to better control the monitoring effects and overhead and take measures towards its minimization. We aim to achieve this goal in the presented thesis.

Studying the effects and overhead of monitoring benefits all the network actors. Operators, providers, and administrators could understand better the monitoring tool capabilities, and thus be able to better profit from the advantages the tool is able to provide – while also being aware of its weak sides. Possessing such information can help to anticipate resource consumption, advancing capacity planning, obtaining facilitated network management, and delivering better service for the end-user. From a user/tenant point of view, being aware of monitoring footprint can help to maintain their applications, machines, and networks, analyzing the nature of problems, whether they appear due to their action or to those inherent to the service itself.

On the way to revealing how system resources are shared among monitoring and virtual network, both implemented within the same hardware, we follow an experimental approach. We install an experimental testbed representing the study object: the virtual network exposed to the traffic monitoring, all deployed on top of the hardware server.

For the purpose of our study we chose monitoring tools widely used in both virtual networks and traditional networks, supported in virtualized and hardware networking environments:

- sFlow [11] – a packet-level monitoring tool for high-speed switched networks. It is implemented in Open vSwitch and provides monitoring by sampling the ongoing traffic and composing the statistics at remote node. More details in Section 2.3.2;
- IPFIX [12] – a monitoring framework for flow-level information collection. In virtual environment is implemented in Open vSwitch as well. More details in Section 2.3.2.

Depending on the monitoring needs and network setup, these tools are able to provide monitoring for traffic between virtual nodes as well as physical nodes. These monitoring

tools were initially developed for hardware switches and routers to monitor traffic at the packet and flow level. They enable monitoring for high speed switched networks and fulfill well the required monitoring objectives (network performance, billing, troubleshooting, and others). With the appearance of a software switch, such tools and their functionalities started to be used in virtual networking equipment.

Additionally, the functionality of sFlow and IPFIX, namely sampling, allows to control the amount of measurements to be performed. Sampling can greatly reduce the cost of measurement in terms of secondary produced data, networking resources for its transporting and datacenter facilities for its storage. It is useful especially depending on the size of the datacenter network and number of cloud tenants and their traffic. As an example, Facebook continuously monitors its data centers servers at a rate of 1 out of 30,000 packets with a tool akin to sFlow [13]. Such rates of monitoring may not be suitable for smaller networking player. Thus the necessity to calibrate the measurement arises and brings additional challenges to network monitoring.

The ability of these tools to serve numerous monitoring objectives and to perform multiple monitoring functions, as well as their integration in the mostly used virtual switch, motivated the research work in this thesis.

1.3.1 Contributions

This thesis tackles the issue of system resource consumption of virtual network monitoring and proposes a solution to tune monitoring parameters so as to control its interference with application traffic. Our contributions are listed below:

- We provide an overview of virtual networking and describe the state-of-the-art solutions for monitoring of various aspects of virtual networks: from infrastructure-level to network performance monitoring in clouds and SDN-based environments;
- We explore the system resources consumption of typical monitoring processes run in virtual switches in terms of network-related and server-related resources;
- We demonstrate the existence of a negative correlation between the monitoring tasks and the operational traffic in the network under flow- and packet-level monitoring;

- We analyze this effect with regard to system resource usage and show that such an effect is not caused by a lack of server resources;
- We provide an overview of scientific works, proposing to leverage machine learning algorithms for benefits in the area of networking and its monitoring;
- In order to minimize the overhead of virtual network monitoring, we propose a solution based on Machine Learning to *(i)* identify a potential drop in throughput due to traffic measurement and *(ii)* automatically tune monitoring parameters so as to limit the measurement and traffic interference in the virtual environment.

1.4 Thesis Outline

In this Chapter 1 we described the notion of network softwarization and the need for its monitoring. We revealed the rationale behind our work in this domain and explained how important and challenging monitoring becomes in the scope of virtual networks, especially with relation to resource management. We briefly summarized the issues that we consider in this thesis and listed our main contributions. The rest of this manuscript is structured as follows:

In Chapter 2 we explore network measurements performed in a virtualized environment and what virtual network represents by itself. We briefly present different virtualization solutions and platforms. We explore the peculiarities of network measurement in virtual networks, clouds, and SDN. We provide a categorized list of the existing contributions of the research community in these areas. Furthermore, we discuss how machine learning techniques rush into virtual networking and networking measurement.

In Chapter 3 we first describe our experimental testbed, next we present the results of our experimentation with two measurement tools deployed into virtual network and the effect of virtual network measurement process on system resources.

In Chapter 4 following our discovery of measurement interference with a virtual network, we head towards avoidance of such interference by means of machine learning. In this chapter, we first describe how we collect a dataset of measurements to use it further to build machine learning models and then obtain a solution that can propose such monitoring parameters to control the level of measurement footprint in the network.

1.5 List of Publications

- International conferences
  - K.Gogunska, C.Barakat, G.Urvoy-Keller, and Dino Lopez-Pacheco, "On the cost of measuring traffic in a virtualized environment", in *IEEE CloudNet*, 2018.
  - K.Gogunska, C.Barakat, G.Urvoy-Keller, "Tuning optimal traffic measurement parameters in virtual networks with machine learning", in *IEEE CloudNet*, 2019.
- Poster
  - K.Gogunska, C.Barakat, G.Urvoy-Keller, and Dino Lopez-Pacheco, "Empowering virtualized networks with measurement as a service", in *RESCOM*, 2017.

1.6 Awards and distinctions

Best Student Paper Award at IEEE CloudNet 2018 for the paper "On the cost of measuring traffic in a virtualized environment".

Chapter 2

Monitoring Virtualized Networks: State-of-the-art

2.1 Generalities on network virtualization and monitoring

2.1.1 System virtualization

The idea of system virtualization (in its modern interpretation) is to share hardware resources of a computer (which is called ”a host“) among multiple securely isolated operating systems (called ”guests“), by so establishing a pool of independent ”virtual“ computers within just one physical computing system instance. The advantages are plentiful:

- expenditures reduced: virtualization allows to exploit resources of hardware server to the full in a dynamic and efficient manner. For datacenter it means purchasing fewer servers and reducing their energy costs;
- better resource utilization: virtualization provides a natural way of co-locating multiple processing services, which greatly improves utilization ”density“ of the computing resources while maintaining their isolation and scalability;
- thin provisioning: virtualized components can be deployed at will on the host systems, in reasonably little time. Such a property improves the resource utilization efficiency of the host hardware, allowing to dynamically dimension the number of virtual components according to their current demand.

System virtualization can be achieved in different ways. Nowadays, two main families of virtualization technologies can be identified: hardware virtualization, and operating system (OS)-level virtualization.

Hardware virtualization is implemented in the form of hypervisors. The first mentioning of hypervisor was in [1]. It was called a virtual machine monitor (VMM) and was defined as a piece of software, having three main characteristics:

1. ”the VMM provides an environment for programs which is essentially identical with the original machine;
2. programs running in this environment show at worst only minor decreases in speed;
3. the VMM is in complete control of system resources“.

Nowadays different virtualization players propose their own definition of the hypervisor. The authors of [14] considered several definitions of hypervisors and propose to define a hypervisor as: ”a thin software layer that provides an abstraction of hardware to the operating system by allowing multiple operating systems or multiple instances of the same operating system, termed as guests, to run on a host computer“. A considerable number of successful hypervisors are available today, and their differences are significant enough so as keep them in quite different niches:

- KVM/QEMU [15]. KVM/QEMU stands for the kernel-based virtual machine – open-source virtualization platform, which allows Linux kernel to perform functions of a hypervisor.
- VMware (VMware vSphere) [16]. VMware is a commercial virtualization project, which offers bare-metal virtualization and hosted virtualization. Desktop hypervisor software runs over Microsoft Windows, Linux, and macOS, whereas hypervisors for servers do not require an underlying operating system.
- Xen Project [17]. Xen is an open-source hypervisor. Except for hardware virtualization (XenServer/XCP[18]), it is mostly known for its paravirtualization mode, meaning that multiple operating systems can be executed on the same hardware at the same time.

- Hyper-V [19]. Hyper-V is a Microsoft hypervisor. It runs over Windows-based operating systems and has different architectural principles than those which derive from Linux.
- PowerVM [20]. PowerVM is an IBM server virtualization product. Commercial solutions provide numerous virtualization features regarding resource management, aggregation, migration, real-time information about virtualized workloads and many more.
- Vx32 [21]. Vx32 is a user-level sandbox for code execution. Vx32 runs on several popular operating systems, does not require kernel extensions or special privileges.

OS-level virtualization is a paradigm that implies the deployment of multiple isolated processes that can perform various functions within an operating system. One of its examples is containerization.

Container-based virtualization can be considered as a lightweight version of hypervisor virtualization: it performs isolation of the *processes* inside the operating systems – as opposed to the isolation of the entire OS in the hardware virtualization. A container in its simple form is a mere process with a restricted filesystem exposure, running in the host OS user space. The implementation of the mentioned filesystem exposure restriction is important, as it, in fact, implements the basic isolation. For instance, by presenting an ordinary directory as a root point to the container, programs that run within it would falsely consider having access to the entire filesystem. In comparison to usual filesystem permission, such a mechanism gives an opportunity to implement more extensive isolation of a process, while giving the contained programs full freedom to access their fictitious filesystem with no impact outside of it. While modern containers rely on more advanced mechanisms than the one presented above, root directory changing has been used quite extensively for smaller tasks (often within applications themselves) in UNIX-like systems since almost the inception of the UNIX, under the names of “chroot” and “jails”.

Being a process in the host user space, the container’s internal activity is competing for resources in the same way as other non-container processes. Being a process also gives containers the access level to the specific host hardware as for the host OS. These two properties free the containers from a need for a per-container guest operating system (and hence hypervisors), as well as from specific hardware pass-through mechanisms that often

pose a significant architectural challenge in hypervisors (for example, VirtualBox having limited support for graphical processing units). One of the main disadvantages of such an approach is that resource allocation and reservation becomes less trivial than for hypervisors, though technically possible to a certain extent. The lack of a guest OS environment to serve specific system calls to the containers also requires them to be compatible with the host OS: it is, therefore, not normally possible to host, for example, Linux-based containers in Windows host environment.

Nowadays, containerization is represented largely by the following utilities and softwares: Docker [22], LXC [23] , OpenVZ [24], chroot, FreeBSD Jails [25].

The goal of hypervisors and containerization remains the same, however, the underlying implementation differs, thus affecting different features from performance to the overhead of the platform. The comparison of hypervisors, their strength and weaknesses are explored in several works, e.g.:

- performance comparison of container-based and hypervisor virtualization: KVM, LXC, Docker and OSv [26];
- performance analysis of four virtualization platforms: Hyper-V, KVM, vSphere and XEN with regard to CPU, memory, disk and network resources [27].

Many cloud computing software solutions are developed on the basis of hypervisors - Apache CloudStack [28] (for creating, managing and deploying cloud infrastructure services), OpenStack [2] (for virtualization of servers and other resources for customer use), RHV [29] and oVirt [30] virtualization management platform.

2.1.2 Network virtualization

Network virtualization works as an abstraction that hides complex underlying network resources, enables resource isolation and encourages resource sharing for multiple network and application services [31]. This is usually achieved by virtualizing active network equipment so that their resources could be shared between multiple isolated overlays. This way of operating the network provides many benefits, such as portability, flexibility, and scalability – when compared to traditional, physical network environments. A typical virtual network

architecture is presented in Fig. 2.1. As presented in Fig. 2.1, physical servers interconnected by means of traditional networking devices organize a traditional network. Each of those servers by means of virtualization can be virtualized into several virtual machines interconnected by the virtual switch, uniting into virtual networks among available hardware servers.

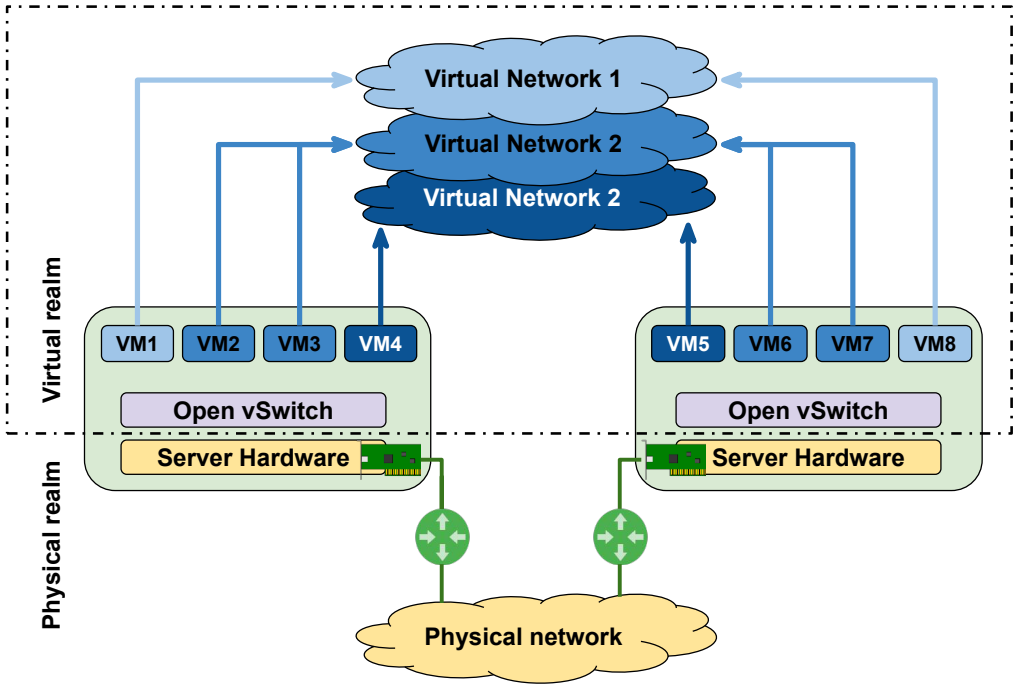


Fig. 2.1 Virtual network: physical and virtual view.

One of the most prominent examples of such performance optimization is the issue of network throughput in virtualized network devices. Traditionally, virtualization and sharing of extension host devices (such as ones attached via PCI) between different VMs had to be implemented by means of the hypervisors themselves. This entailed compatibility and support difficulties, as well as (often) a decrease in the performance of those devices when accessed from a VM. Such an issue was especially problematic for virtualized network devices because their cornerstone component – Network Interface Cards (NICs) – are typically implemented as PCI/PCIe extension cards, and hypervisors were not always capable of providing their VMs with a line-rate NIC performance at high data rates. A substantial improvement in

this regard has been achieved by the introduction of Single-Root Input/Output Virtualization (SR-IOV).

Within SR-IOV specification, a PCIe device itself supports isolation of its resources for direct VM access, in a form of physical and virtual functions. A physical function (PF) presents itself as a full-featured PCIe device, supporting input/output as well as device configuration. Within a physical function, virtual functions (VF) can be defined, which provide virtualized devices supporting input/output only within their parent physical function configuration. When VMs perform I/O operations within these functions (that appear as PCIe devices), hypervisors move the data to and from the physical PCIe hardware directly, bypassing their own internal network abstractions (whenever such functionality is supported, of course). Such a standard specification with a resource sharing and isolation in mind has greatly promoted high-bandwidth I/O for network virtualization.

An important example of network virtualization (which is also heavily used in this thesis) is a software switch - an application to perform packet forwarding and other switch capabilities between virtual NICs (vNIC, that can be attached to VMs) and physical NICs of physical equipment. The most famous player in the domain is Open vSwitch [32], nevertheless, there are other non-hardware implementations of switches, e.g., snabb [33], BESS [34], PISCES [35], FastClick [36], Fr.io VPP [37], netmap VALE [38].

Open vSwitch was created as a *simple flow based switch with a central controller* and was released in 2009. It is a multilayer virtual switch implemented in software. It operates like a physical switch, e.g., OvS handles information about connected devices with MAC addresses, instances tap interfaces are connected to OvS bridge ports. It uses overlay (GRE/VXLAN) networking providing multi-tenancy in cloud environments.

Due to OvS architectural principles, switching is performed in the kernel module, the equivalent of the fast-path of physical switches/routers. Indeed, the kernel can directly copy frames from one interface to another without any context switching, which is the fastest option in an off-the-shelf computer [1]. However, such an operation is not possible for every packet but only for those for which an entry exists in the associate array used by the kernel mode [39]. Other packets, typically the first packet of any new connection must be handled by ovs-switchd, which is executed in user space (hence slower, as some context switching will occur while processing this packet). Ovs-switchd will perform a set of hash lookups in

[1] note that OvS supports some kernel by-pass techniques like DPDK, which is out of the scope of this thesis.

its rule tables to find the rule matching the packet. The rule will next be installed in the kernel module path to speed up the processing of the next packets form the connection. Fig.2.2 vividly demonstrates the packet handling in OvS in kernel and user spaces.

With virtualization, a VM (or a container) has its own TCP/IP stack and virtual interface that is connected to a physical interface of the server. This complexity affects performance, especially for servers with high bandwidth requirements. This is typically the case of NVF [40] in the data centers of the telco ISPs, which are in general close to the mobile base stations. While telcos want to benefit from the flexibility of virtualization, by virtualizing some of their network operations into so-called VNFs, they do not want to sacrifice performance. For these cases, hypervisor bypass solutions like DPDK have been invented [41, 42]. In such cases, a direct link is established between the user space in the VM and controller of the physical interface. This enables to achieve high performance, at the expense of a loss in terms of flexibility as these DPDK-enabled VNFs cannot be easily migrated from one server to another.

This type of solution is also offered in public/private clouds for users/applications with specific network requirements. In this thesis, we do not consider these niche cases but focus on the mainline case where the network traffic of the VM will be handled by the hypervisor.

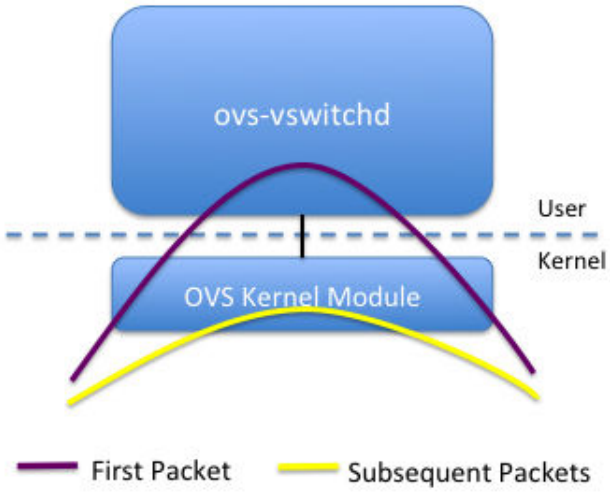


Fig. 2.2 OvS data path[2]

[2]Source of the figure https://blog.sflow.com/2015/01/open-vswitch-performance-monitoring.html

The comparison and evaluation of 6 state-of-the-art software switches were conducted in [43]. The authors proposed a benchmarking methodology for understanding the switches' performance baseline. A methodology implies 4 test scenarios to evaluate the performance of software switches. In this work OvS-DPDK, snabb, BESS, FastClick VPP and netmap VALE switches were compared and evaluated. The results of such comparison did not bring striking insights: no single switch prevails others. The performance depends on the switch implementation and investigated scenario, e.g., BESS prevails over others in the scenario where packets are forwarded from physical NIC to VNF, whereas snabb wins in the scenario of VNF to VNF. Performance characteristics of virtual switching were also described in [44] and performance modeling in [45].

2.1.3 Monitoring the virtual network

The complexity and blend of software and hardware in data centers raise the difficulty to monitor and debug performance issues in a virtualized environment. A computer network is a complex system, and a failure of a single component can impact the system as a whole and network users' experience: from incorrect network configurations or virtual machine settings to underlying servers and network devices problems. Monitoring the network health can help to identify such issues, and so to troubleshoot network appropriately in case of failures.

Generally, monitoring can be defined as a process of data collection and processing. More specifically, monitoring in the domain of networking implies several subprocesses [46]:

1. *Data collection*: there exist two ways of data collection: passive and active. The active measurement simulates users behaviour by injecting test traffic into the network for monitoring purposes (e.g., SDN rule checking [47]), whereas passive measurement implies observation of ongoing users traffic (e.g., sampling techniques) and its analysis over some period of time. Data collection does not exclude a combination of both. Detailed analysis of the pros and cons of these two methods is presented in section 2.5.

2. *Data preprocessing*: previously collected data is prepared for further stages (itemizing, labeling, packaging, etc.)

3. *Data transmission*: prepared data is being sent to the analysis entity.

4. *Data analysis*: analysis phase, when received data is investigated and all kinds of statistics are built.

5. *Data presentation*: the results of data analysis are presented or visualized for further perception, providing a view on the network.

Apart from trivial assistance in network debugging, monitoring becomes a crucial process for service providers. In such a case, it serves several more complex objectives:

1. Quality of service delivery: there exists a certain level of services, that are delivered to a customer by the provider, which is specified in the Service Level Agreement (SLA). SLA obliges the provider to monitor and keep services parameters to a required level and assure proper operation of the services, that customers pay for.

2. Billing and accounting: network monitoring allows to collect necessary statistics of computer resources and services usage, as in the cloud computing paradigm pay-for-use, when a customer pays to a provider in accordance with the amount of utilized system resources.

3. Efficient resource usage: monitoring of system resources (memory, CPU cycles, I/O, etc.) helps to anticipate failures due to resource limitations.

4. Performance monitoring: monitoring of network performance implies collecting network-related metrics, such as bandwidth, throughput, latency, jitter, etc.

5. Security/privacy assurance: constant monitoring helps to detect numerous kinds of security attacks (intrusion, DDoS).

6. Fault management/troubleshooting: it is important to keep track of network state during its operation. Constant network monitoring helps to detect, diagnose, and prevent network faults/frauds, which in turn assures decent service delivery to customers.

The monitoring of a virtual network implies the same stages, however considering the architectural principles of clouds, SDN and virtual networks, it includes monitoring of underlying physical components (hardware servers and network) and the platform that enables virtualization, and monitoring of virtual components themselves (virtual hosts, virtual network, applications health).

2.2 Monitoring in the cloud-based environment

Being the basis for the virtual infrastructure, the underlying physical network is, perhaps, the most important component in a virtual network. It is not uncommon to "overbook" the physical resources for virtual overlays, so a single physical failure (such as network connectivity issue, lack of server resources) may quickly translate into a massive outage in a virtual domain. To overcome such issues multiple resource allocation algorithms and VM migration techniques were proposed [48], [49], [50], [51], [52]. It is also worth noting that virtual networks are supposed to be isolated from their physical substrate by means of hypervisors. This gives one an opportunity to create such a monitoring scheme that would help to identify the exact failure domain and, by so narrow down the problem scope for faster issue resolution.

However, as it has been noted in the previous section, modern networks are not only composed of physical devices but also of a wide range of virtual network functions deployed on top of computing platforms – for instance, virtual switches, load balancers, firewalls, application gateways and so on.

Literature specifies at least three classes of performance problems of these virtual platform elements [53]: *(i)* miss-allocation of resources for element placement and functioning, *(ii)* contention for resources between elements, and *(iii)* implementation bugs. PerfSight aims at diagnosing such problems [53]. PerfSight tries to identify the location and reveal the nature of performance degradation in the virtual domain. PerfSight relies on a controller, that manages agents and diagnostic applications, analyzing collected data on the way to retrieve the root cause of the observed problem. A collecting agent running at each server collects the necessary data (packet counter, byte counter, I/O time) of a certain network component (all VMs placed on the server, only VMs of a single tenant). The obtained statistic is enough to pinpoint the problematic element according to authors. The diagnosis is based on a rule book, which is constructed offline and aids to map the symptom to a resource contention or bottleneck problem, i.e., packet drop location (tunnel between virtual switch and vNIC, server NIC driver, etc.) has a match to a certain resource in shortage (CPU, memory space, incoming bandwidth, VM CPU, etc.). The overhead of such metric collection and analysis is studied.

It is evident that virtualized network functions neighboring in the same physical server compete for its resources. Interestingly, different VNFs behave (in terms of resource con-

sumption) in a different manner when placed together. In [54] the authors provided insights on how various types of VNFs, co-located in the same server, tolerate each other and what degree of server resource degradation can be expected from such neighbourship. The authors also give some recommendations on VNF placement for efficient resource usage.

NFV networks – networks that are composed of virtual network functions and physical devices, require different measurement solutions than traditional networks as state the authors of [55]. They also propose their solution for such networks: a measurement system, which enables passive and active measurements in Linux containers, called Virtual Network Measurement Function.

2.2.1 Cloud infrastructure monitoring

Even if the physical substrate health is assured, problems may appear from the virtualization enabler, such as hypervisors and cloud management stacks. Solutions offering cloud or virtualization deployment are complex systems with distributed architecture, that tend to incur faults related to operation or performance, provisioning compute and storage workloads, incorrect component configuration, third party dependencies, API latencies, etc. That is why some research work has focused on monitoring the health of the cloud infrastructure aspect, which refers to hardware and software components that support cloud services [56], [57, 58], [59]. As an example, OpenStack is a popular and convenient framework to develop cloud on-premises. It has modular architecture - different interrelated components support certain functions of the cloud: Nova - cloud computing controller, Neutron assures networking management, Cinder - storage system, Swift - object storage system, Horizon - provides a dashboard for platform administration, etc. Platform users can manage their cloud through graphical interface (dashboard), command-line or RESTful web-services. Hansel and Gretel [57, 58] focus on troubleshooting the infrastructure of OpenStack and leverage the REST calls exchanged between the OpenStack modules to reveal components failures. Hansel [57] focuses on fault detection, whereas Gretel [58] performs fault diagnosis by constructing a precise sequence of messages exchanged between system nodes corresponding to a task or operation (operational fingerprint). A data-driven approach is followed by these tools to pinpoint the origin of system errors. In terms of system overhead Hansel is using 100MB of memory and 4.5% CPU, and its enhanced version Gretel uses 123 MB memory and 4.26% CPU on typical modern servers, which can be considered fairly small footprint.

CloudSight [59] is a cloud infrastructure monitoring framework revealing the state of cloud resources to tenants to increase the visibility of tenants' problems, caused by underlying cloud resource abstraction. The tool logs the states of cloud components, keeps the resources change history, generates resource graphs containing information regarding resources involved in tenant's instances operation, which is available for interpretation for cloud tenant. CloudSight is supposed to run on top of any cloud platform and was prototyped and evaluated in OpenStack.

CloudHealth [60] offers a model-based monitoring approach for configuration, deployment, and operation of monitoring infrastructure in the cloud. CloudHealth is designed for cloud operators. Monitoring objectives are translated into the necessary set of metrics in accordance with two ISO standards (ISO/IEC 25010:2011 and ISO/IEC TS 25011:2017), which provide specifications of quality models for IT services quality evaluation. As an example, responsiveness – a level at which the service promptly and timely responds to requests and provides the required functionality, – can be understood by paying attention to waiting time to accept requests and waiting time to receive a response.

2.2.2 Network monitoring in cloud

Network performance diagnostic in the cloud is a hot topic and over past years numerous solutions emerged: for cloud operators use or for tenants control, platform-specific or of general application, etc. Resource sharing in the cloud may lead to performance problems, which can be the result of either problem in providers' infrastructure or the tenants' VMs. RINC (Real-time Interference-based Network diagnosis in the Cloud) [9] is a framework for diagnosing the performance problems in the cloud for cloud operators. It provides monitoring of tenants' connections within the cloud, leveraging the idea of multiple phase measurement. The main idea of multiple phase measurement implies collecting light-weight data to detect problematic connections and heavier-weight to discover root-causes of troubling connections, e.g., harvest simple statistics (e.g., throughput) to detect problematic connections, then on such connections enable deeper-level measurement (congestion window, maximum segment size, round-trip time, etc.) for the purpose of root-cause analysis. RINC measurement agent runs in the hypervisor and relies on TCP statistics (SYN, FIN, CWND, sstresh, sending rate, throughput, packet, and byte counters) by inspecting packets passing through the hypervisor (implementing active measurement approach and sampling for certain cases is

also considered). The measurement agent communicates with a global coordinator module, which aggregates data received from the agents. The global coordinator module also provides an interface between the cloud operator and the monitoring framework. Using this module the cloud operator specifies the statistics needed to be collected. The cloud operator can employ the query interface provided by RINC to write diagnosis applications. Examples of diagnosis applications include detecting long-lived connections, heavy hitters and super spreaders, root cause analysis of slow connections. In terms of overheads, the worst case (querying all the connections) memory consumption is 3.5MB and 0.01% of networking overhead. The CPU overhead is considered with respect to the number of concurrent flows: 0-10000 flows lead to a CPU consumption in the range 0-27 %, and a network footprint (to convey measurement metrics) of 0-10 Mbps corresponding to 0-2% of the CPU consumed.

Virtual network diagnosis as a service (VND) [61] is also a query-based virtual network diagnosis framework. However, in contrast to RINC, it provides network troubleshooting tool for cloud tenants and their virtual network. It is composed of a control server, which translates tenants' queries into diagnosis policy, and several servers that collect network traces, perform data parsing and storing. It provides troubleshooting by collecting flow traces, mirroring the problematic flows and extracting traffic metrics (RTT, throughput, delays, etc.) to identify network related issues, such as packet loss, high delay, network congestion, heavy heaters, etc. Due to the trace collection method used, i.e., traffic mirroring, the overhead in terms of memory throughput is 59MB/s for every 1 Gbps of traffic, plus storage required to keep SQL-database. Even though VND can help to reveal certain networking problems using queries on data from packet headers, sender or receiver side problems remain undetected [9].

A platform-independent monitoring framework for cloud service instantiations is CloudView [8]. It allows cloud tenants to perform active and passive measurements between virtual nodes after their instantiation, utilizing arbitrary measurement tools according to their needs. Third-party measurement tools can be integrated into CloudView and provide tool-agnostic flexibility to pick a tool for a particular metric and particular measurement task. The authors also integrated CloudView in OpenStack and evaluated it on two different cloud infrastructures. Two main components are sensor pod (the framework which hosts measurement sensors, e.g., ping, pathChirp, tulip, etc.) and the information manager (application for user interaction with platform). They observed that sensor components play a major role in monitoring resource consumption: overhead of 0.5 % of CPU and memory beyond the raw invocation of the sensors.

Dapper [62] is a TCP-related performance diagnostic tool for the cloud implemented based on the P4 programming language for packet processing. It distinguishes performance problems according to their location: sender side, network side or receiver side. Slow application data generation due to resource constraints, and long application reaction time are sender-side problems; congestion, packet loss, and path latency are network-related problems; performance degradation can also occur due to receiver limitations regarding delayed ACKs or small receive buffer.

NetWatch [7] is another cloud performance monitoring tool suitable for both client users and cloud providers' needs. It provides an API interface to query measurement tasks on demand. It has three components delivering measurement in the cloud: a controller, an agent, and a probe (as it exploits active measurement). The controller is a communication node between the service customer and the measurement framework. It issues control messages for agent and probe to realize specified measurement tasks. The agent is parsing controller instructions, executing them, managing probes creation, configuration, and deletion, and obtaining reports on measurement results. The probe module is the software component executing the actual network probing. NetWatch was prototyped in OpenStack. Measuring network performance depends on the number of measurement tasks to be performed. For example, NetWatch introduces reasonable overhead in terms of CPU - 0.7% per measurement task and negligible influence in terms of memory and 0.01% of the bandwidth of gigabit NIC is consumed by executing five tasks.

ConMon [5] is a network performance monitoring solution for container-based virtual networks. In ConMon a monitoring controller resides in every physical server and communicates with monitoring containers, which execute monitoring functions to monitor different network performance metrics (packet loss, delay, jitter, path capacity). During their evaluation of ConMon, the authors conducted network tests and reported accumulated CPU usage of 100-200 % in total. The latency between two application containers expressed in RTT is 4.5 microseconds and 130 microseconds correspondingly for cases when containers reside in the same and different servers. No packet loss or impact on background traffic was observed.

2.3 Monitoring in SDN-based environment

In traditional networks traffic is forwarded between independent and autonomous devices. These networking devices require individual configurations. Network operators experience difficulties maintaining, debugging and managing various networks built upon such networking equipment. Traditional networks having static, decentralized and complex architecture do not offer the flexibility required for modern environments such as cloud networks with multiple tenants.

The SDN paradigm aims to resolve the challenges of network devices control and management by separating the control plane (network management and configuration) from the data plane (network traffic) of software and hardware components of networking equipment. Decoupling the control and forwarding functions transforms the network into a programmable environment, where all the configuration of forwarding devices is (logically) centralized and these devices only follow the rules issued from the intelligent controller. Within such networks the traffic passing through network devices is considered in the form of flows - a set of packets with common properties (e.g., source, destination, protocol). Rules installed by the controller are used at each switch to determine the operation to perform on flows.

Due to their architectural differences, monitoring in traditional networks and SDN-based networks differs. Such a difference in monitoring legacy networks and SDN is considered in [46]. Briefly, SDN facilitates and accelerates monitoring phases. Data collection requests can be issued from a controller with the necessary frequency to be adjusted anytime from any device. Whereas, adjusting collection parameters in legacy network devices (equipped with collecting function) would require to manage the configuration of each device individually. Furthermore, the transmission via legacy networks is done with the help of transmission protocols, whereas SDN offers APIs and SDN protocols. With their help, the interactive interface for further measurement operations within SDN can be deployed. Moreover, they can offer functional measurement data representation and visualization.

2.3.1 SDN control plane monitoring

Before monitoring network anomalies in SDN, it is necessary to make sure that the system is working consistently: the controller properly manages networking nodes and networking nodes follow controller instructions. Different kinds of software faults (e.g., switch software

bugs, outdated versions of switch software and OpenFlow [63]) and hardware (e.g., bit flips, non-responding line cards) may trigger packet forwarding which does not match control plane instructions [6]. To assure that such networking problems do not happen due to imperfect communication between controller and network, VeriDP [6] follows a passive approach to track control-data planes consistency, meanwhile Monocle [47] follows the active approach for the same task. Some solutions to check SDN configuration correctness appeared earlier [64], [65], [66], [67].

VeriDP [6] tool consists of a server and a pipeline. The pipeline deals with tagging and reporting packet headers to controllers of different types of switches in the network: entry, exit, and internal switches. The server is put alongside the controller and intercepts controller communication with switches to construct a path table. Reported packets are to be verified according to the constructed path table. If the packet fails the verification, it means that configuration inconsistency is present and the server tries to localize the faulty switch. The reported overhead of VeriDP is expressed in terms of processing delay of its pipeline and native OpenFlow pipeline in hardware SDN switch. Tagging delays equal to 0.27 microseconds, and sampling delays of 0.15 microseconds are reported in [6].

Monocle [68] is another solution to verify whether the network view configured by the controller corresponds to actual devices behaviour. Monocle is inserted as a proxy into the network between the controller and the switches. Firstly, it inserts catching rules in the network and then issues test packets to check the specified rules, when received at a switch with installed catching rule, are forwarded back to Monocle, where the conclusion on rule-behaviour correctness is made. The authors of Monocle argue that injecting packet probes into the network does not overload switches and consumes small switch resources space.

2.3.2 SDN data plane monitoring

SDN (Software Defined Networking) is gaining momentum in modern data centers [69]. As such, a significant amount of works has focused on measuring SDN networks, their performance and resources management.

SDN's intelligent centralization paradigm is not a cure-all solution. It also has its issues and may become a bottleneck with an increasing amount of tasks in the network [70]. Such

issues with centralized controller bottlenecks are addressed in DIFANE [71], DevoFlow [72], software-defined counters [73], Onix [74], Kandoo [75], HyperFlow [76] and Maestro [77], which try to reduce the measurement overhead of the control plane by sharing some tasks with forwarding devices or trying to improve controller performance by means of buffering, pipelining, multithreading and parallelism.

Data center exploits large scale, high speed and high utilization networks, that are complex to monitor. Tremendous effort was put into developing tools that would allow network operators to manage and troubleshoot their datacenter networks [78], [79], [80], [81], [82], [83], [66], [84], [85]. Some works [66], [84], [61] require entire data plane snapshot and may only be able to track specific events at coarse-grained time-scale [85]. Sampling technologies were introduced in order to alleviate overheads of such solutions [86], [11], [87], [88], [89], [90], [79], [91], [6], [92], [10]. Traffic sampling implies extraction of a certain portion of traffic for the purpose of network analysis. SFlow [93] is a famous sampling-based packet monitoring solution, that was adopted from traditional network devices and is now implemented in software switch (OvS) too. Full or partial extraction of traffic in SDN-based cloud environments is offered by TREX [94]. CeMon [10] develops three sampling algorithms for their framework in order to eliminate measurement overhead. UMON [95] modifies the way OvS handles SDN flow tables to decouple monitoring from traffic forwarding, hence building specific monitoring tables. UMON has a low CPU footprint: monitoring activities of 150 hosts takes 9.9% CPU of a core. A considerably small packet loss is present during monitoring with UMON of 26 packets per second. Installing monitoring capabilities at the network edge is proposed by Felix [96]. Predicates are used to answer monitoring queries of interest. Applying declarative query languages for network measurement was first proposed by GigaScope [97]. It allows to execute numerous tasks for network monitoring (traffic analysis, intrusion detection, router configuration analysis, network research, network monitoring, and performance monitoring and debugging) and since then high-level language programming and programming itself was leveraged in a variety of approaches in the context of SDN [98], [90], [81], [68], [99], [100], [101].

Measurement tasks are often executed on the commodity switches in SDN, which rely on TCAM (Ternary Content-Addressable Memory) - a very efficient, however greedy in terms of resource consumption type of memory. High energy consumption of switch memory makes it the most expensive component of the device. It is the number one priority for TCAM to execute networking functionality, and introducing the burden of measurement functions into

TCAM has to be done with great care. That is why DREAM [102] proposes to dynamically distribute the measurement tasks over several SDN switches depending on the utilization of the flow rule table and the targeted accuracy. DREAM considers an overhead in terms of task allocation delays, which the authors estimate to be negligible compared to other delays. SDN-Mon [103] also sends some tasks to switches in the network to be implemented by switches processing powers.

SDN traffic is represented by flows – a sequence of IP packets of common properties, sent between sending and receiving nodes in the network. The term *'flow'* was firstly mentioned in RFC 1272 [1] published in 1991 by the IETF Internet Accounting Working Group with the aim to introduce Internet traffic accounting. In 1995 packet aggregation in flows was formulated by [104] and first efforts towards flow monitoring were started. Meanwhile, without a clear purpose of flow monitoring or traffic accounting, Cisco devices were handling packets using a flow level approach: to accelerate packet switching procedure, only the first packet of the flow was considered when selecting a forwarding decision. This fact, alongside the need for flow monitoring, inspired the development of NetFlow technology. NetFlow is a monitoring protocol initially proposed by Cisco System for traffic accounting inside their networks and equipment. It, however, got widespread among networking devices of other vendors as well. NetFlow operates by means of three components interacting with each other: a flow exporter (which performs packet aggregation into flows and provides flow statistics to collectors), flow collector (which receives, stores and processes statistics from flow exporter) and analysis application (which analyses flow data for the necessary monitoring purpose and provides human-readable reports). Later on, it was integrated into virtual environments, e.g., Open vSwitch.

OpenFlow being a de-facto standard for SDN networks can reveal a lot of network information with its functionality (port statistics, flow-counters, etc.). The appearance of a centralized controller facilitated the polling of the networking equipments and provided an uncharted territory for the development of networking monitoring frameworks.

Numerous works have been utilizing OpenFlow to achieve different monitoring goals [105], [106], [107], [108], [95], [10], [109]. Baatdaat [105] proposes to insert the measurement into the SDN network to use it to disclose the topology-wide network utilization map. To schedule traffic flows Baatdaat uses OpenFlow to adapt to traffic bursts and average link load. The eventual results show reduced network-wide link optimization by up to 18% with ECMP and improved flow completion time by 41%-95%. FlowSense [106] offers

performance monitoring of flow-based networks using OpenFlow messages sent by switches to the controller. It utilizes passive observation of exchanged messages thus allowing to compute link utilization and performance changes and avoid additional overhead. PayLess [110] is a network statistics collection framework also based on OpenFlow controllers north-bound API. CeMon [10] is also built around OpenFlow. It is a generic SDN monitoring system, which can be implemented alongside other SDN monitoring frameworks to achieve lower-cost measurements, as the main CeMon implementation intent was to reduce the cost and overhead of flow statistics fetching. OpenNetMon [109] is a flow monitoring module working on OpenFlow controller POX. The monitoring is performed combining adaptive polling (throughput and packet loss) and active probing (delay) in order to verify whether the QoS parameters are met. OpenNetMon also uses adaptive rate of measurements to minimize the overhead in the network and on the switch, while paying attention to the measurement accuracy.

OpenWatch [111] offers anomaly detection by adjusting the granularity of collected flow statistics: finer-grained measurement data when a network attack is suspected and coarser-grained during normal operation. Such an approach is supposed to unload traffic measurement when unnecessary and identify network anomalies faster when they happen. OpenWatch overhead is considered in terms of the additional number of reports with comparison to other approaches of flow statistic collection: *no aggr* - no aggregation method for IP address counting and *static* aggregating with fixed mask length (whereas OpenWatch enables adaptive aggregation of flows). *No aggr* method introducing the biggest overhead in terms of reports, meanwhile OpenWatch introduces additional load up to 10% as compared to the *static* approach.

In Table 2.1 we summarize considered monitoring tool. We present them in several perspectives:

- Tool'year implies a tool's name and year of publication.
- Monitoring object describes the domain to which a monitoring solution contributes: SDN management (control plane monitoring), or SDN performance (data plane monitoring).
- Monitoring actor for which player of the virtual network the measurement solution is proposed: SDN administrator, cloud tenant or operator.

- Measurement approach: passive, active or none (non-applicable).
- Overhead considered: whether the authors consider the overheads of the proposed monitoring tool in any display: network-, system-wise, etc.
- Monitoring goals is a set of monitoring tasks that a monitoring tool can serve.

2.4 SFlow and IPFIX overview

Two versions of NetFlow became popular – 5 and 9. On the basis of the latter, the IPFIX – an open standard for flow monitoring was engineered. IPFIX [12] is also a popular monitoring solution for virtual networks, also functioning with Open vSwitch. IPFIX stands for "IP Flow Information eXport" and its development was driven by vendors to step away from Cisco standards. Indeed even though NetFlow featured interesting monitoring capabilities, additional monitoring demands appearing during network administration could not be implemented, as the protocol is proprietary. In contrast, IPFIX as an extended version of NetFlow v9 supports fields of variable length (e.g., HTTP hostname) and also enterprise-defined fields. Following the idea of flow monitoring of Cisco's NetFlow other vendors also created their versions, such as J-Flow by Juniper [112], Cflowd by Alcatel-Lucent [113], sFlow by HP. The latter has caught our interest as a technology nowadays supported in SDN and virtual networks.

SFlow has emerged in 2001. Its main difference from the previously mentioned monitoring solutions is that it does not perform packet aggregation into flows, but collects individual packets.It is also different from traffic mirroring, as mirroring implies copying all the packets towards another place in the network. SFlow uses sampling to select packets for export and to reduce the amount of exported monitoring data. Additionally, sFlow is configurable in terms of the amount of packet bytes to be captured. SFlow also collects port statistics and counters, which is not the case with mirroring. SFlow has some advantages compared to flow export technology. Indeed, flow-based tools export flow records based on active/inactive flow timers and may take up to 30 minutes.

The appearance of network virtualization and widespread usage of flow monitoring in traditional networks, promoted sFlow adaptation to virtual networks, especially in Open

[3]Source of the figure [93]

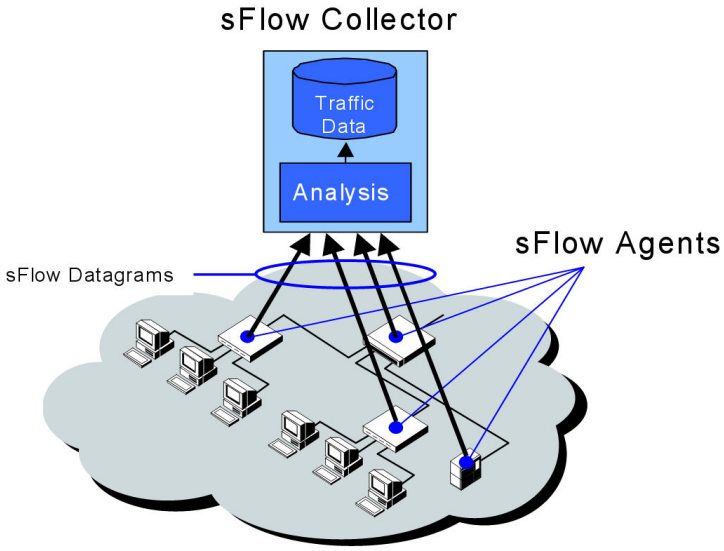


Fig. 2.3 sFlow architecture[3]

vSwitch. An sFlow agent and an sFlow collector are the two components of sFlow. Fig. 2.4 portrays how sFlow works. SFlow operates in user space of OvS. It is implemented by ovs-switchd process, see Fig. 2.5. SFlow performs monitoring using a sampling mechanism (1 packet out of n), which in the case of traffic measurement means exporting a packet header of every n-th packet over the network to a remote collector, which further builds statistics on the monitored network traffic. The sFlow agent is responsible for traffic capture on the device, where sFLow is enabled. Collected sampled packet headers and port statistics are encapsulated into sFlow datagrams, which are forwarded to the sFlow collector for analysis. With sFlow, no computation is made at the switch as compared to NetFlow/IPFIX, which should limit its CPU consumption.

Sampled NetFlow overhead was studied in [114]. The authors conclude that this tool does not incur dramatic overhead during their experimentation. However, since the overhead is proportional to the number of flows recorded, care has to be taken, when using sampled NetFlow for network monitoring. The overhead was considered in terms of usage of router resource and the amount of export and experimentation was performed on an operation Cisco router.

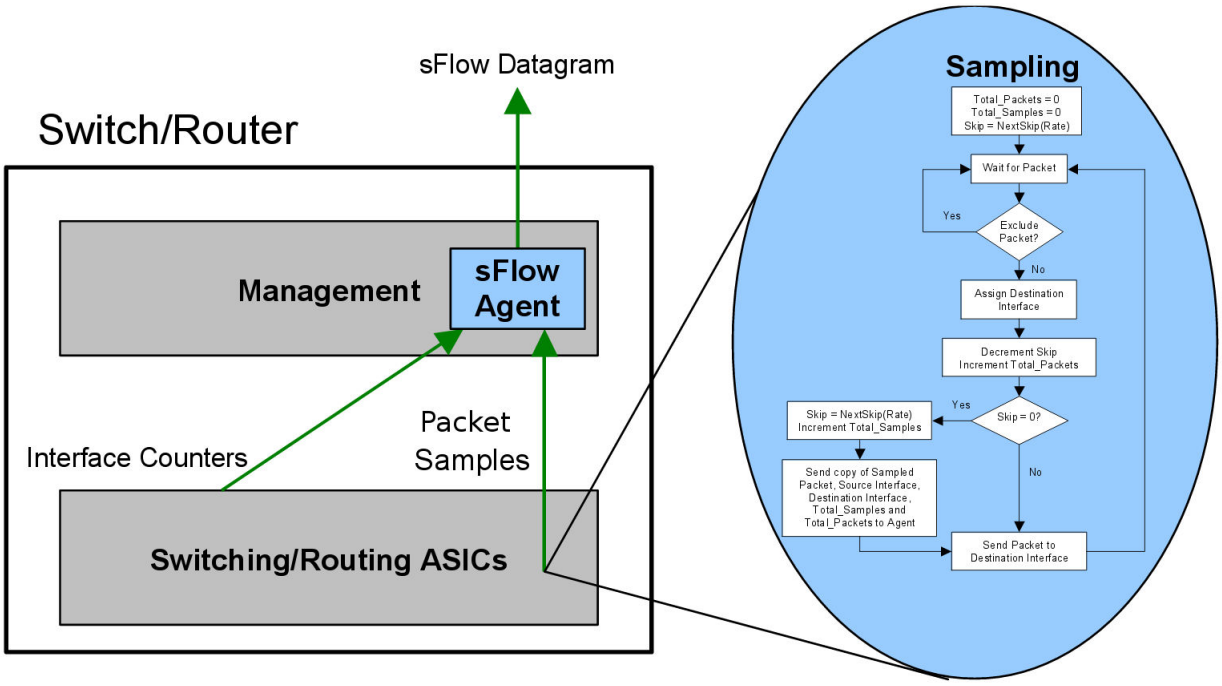


Fig. 2.4 sFlow agent embedded in hardware switch[3]

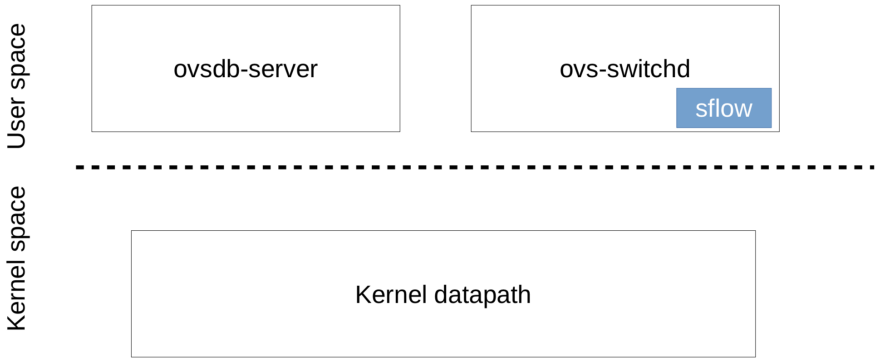


Fig. 2.5 sFlow measurement process in virtual switch

Table 2.1 Monitoring tools

Tool'year	**Monitoring object**	**Monitoring actor**	**Measurement approach**	**Overhead considered**	**Monitoring goals**
DREAM'14	SDN management	SDN administrator	none	considered	resource allocation for multiple measurement tasks: hh, hhh, change detection
Monocle'15	SDN management	SDN administrator	active	considered	control-data plane consistency
VeriDP'16	SDN management	SDN administrator	passive	considered	control-data plane consistency
Flowsense'13	SDN performance	SDN administrator	passive	considered	link utilization, traffic change
OpenWatch'13	SDN traffic	SDN administrator	passive	considered	anomaly detection
FlowCover'14	SDN traffic	SDN administrator	passive	considered	flow monitoring
OpenSample'14	SDN traffic	SDN administrator	passive	considered	elephant flows, congested links
Payless'14	SDN traffic	SDN administrator	passive	considered	flow monitoring

Continued on next page

Table 2.1 – continued from previous page

Tool'year	Monitoring object	Monitoring actor	Measurement approach	Overhead considered	Monitoring goals
CeMon'15	SDN traffic	SDN administrator	passive	considered	flow monitoring: link utilization, traffic matrix, anomaly detection
OpenNetMon'16	SDN traffic	SDN administrator	active and passive	considered	flow monitoring
SDN-Mon'17	SDN traffic	SDN administrator	passive	considered	traffic monitoring
PerfSight'15	cloud management	operator	passive	considered	resource contention faults
HANSEL'15	cloud management	operator	none	considered;	operational faults in OpenStack
GRETEL'15	cloud management	operator	none	considered	operational faults in OpenStack
CloudSight'17	cloud management	tenant	active	not considered	cloud components and resource faults

Continued on next page

Table 2.1 – continued from previous page

Tool'year	Monitoring object	Monitoring actor	Measurement approach	Overhead considered	Monitoring goals
CloudView'13	network performance in cloud	tenant	passive and active	considered	network topology monitoring, performance information for job placement, cloud application monitoring
VND'13	network performance in cloud	tenant	active	considered	filtering, bottleneck middlebox detection, flow trajectory, different kinds of traffic related problems
RINC'15	network performance in cloud, programmable for operators	operator	active and passive	considered	heavy hitters, traffic counter, slow connections, super spreaders
Dapper'16	network performance in cloud	operator	passive	considered	network congestions, resource bottleneck, routing changes, limited bandwidth, etc.

Continued on next page

Table 2.1 – continued from previous page

Tool'year	Monitoring object	Monitoring actor	Measurement approach	Overhead considered	Monitoring goals
NetWatch'16	network performance in cloud	operator and tenant	active	considered	end-to-end network performance: delay, packet loss, congestion detection and localization
ConMon'17	network performance	operator and tenant	active and passive	considered	network performance issues
CloudHealth'18	cloud management	operator	active	not considered	cloud services monitoring: network performance, resource utilization, availability, reactiveness, capacity
TREX'17	cloud traffic measurement and management	tenant	passive	considered	application specific real time anomaly detection, packet loss
UMON'15	cloud traffic measurement and management	tenant	passive	considered	port scan attacks, anomaly detection

Continued on next page

Table 2.1 – continued from previous page

Tool'year	Monitoring object	Monitoring actor	Measurement approach	Overhead considered	Monitoring goals
AC/DC'16	cloud traffic measurement and management	SDN administrator	passive	considered	tcp congestion control
OpenSketch'13	programmable SDN measurement	SDN administrator	passive	considered	traffic anomaly
SNAP'16	programmable SDN measurement	SDN administrator	passive	looks like not considered, check again later	network events, flood detection, elephant flows detection, heavy hitter detection
Felix'16	programmable SDN measurement	SDN administrator	passive	not considered	network stats related queries
sFlow	traffic measurement	network administrator	passive	not considered	packet monitoring
IPFIX	traffic measurement	network administrator	passive	not considered	flow monitoring

2.5 Virtual networks monitoring challenges

Private and public cloud providers that have to deploy networking services face numerous challenges stemming from the combination of platforms that compose their networks. As such, a multitude of monitoring tools exists and new tools continue to emerge to serve any kind of problems met in virtualized environments.

Researchers have to pay attention to several important aspects when proposing a virtual network solution:

- *Preservation of abstraction*: a virtual network is an abstraction of a network. Tenants virtual networks are logically isolated and the hidden physical infrastructure has to remain hidden, as well as the physical points of measurement for tenants' network. Developing their virtual network diagnosis authors of [61] emphasize this point.

- *Scalability*: virtual network monitoring tool is obliged to adapt to constantly changing the size of the virtual network otherwise it will not be able to serve its purpose. Often virtual machines are being deployed to perform shot-time jobs and once the job is done, virtual machines are getting suspended and new machines will be created for new needs. VMs migration is an ordinary behaviour within a virtual environment: machines are being transferred to different servers with spare resources. Also, machines may scale in both directions: horizontal (adding more machines into setup) and vertical (adding more resource powers to machines). In such a dynamically scaling environment a monitoring tool has to bear the scalability challenge, i.e., to adapt to a rapidly changing traffic load.

- *Accurate measurement providing real-time statistics*: certain monitoring goals require measurement to be done over some time, e.g., accounting and billing: necessary data for billing will be available after resources or service were used. At the same time, other monitoring goals demand a timeless reaction from the administrator, e.g., congestion control, heavy heaters detection, intrusion detection, prevention of attacks, flooding, etc.

- *Proper choice of measurement point*: when collecting measurement in an environment composed of multiple components, several operating systems, interconnections of software and hardware networking interfaces, tunneling, encapsulation, one faces the

problem of where such measurement has to be inserted. In addition, one has to consider the fact that several VMs from different owners may sit alongside in the same server, thus their traffic would pass through the same virtual switch and exit the same NIC of the server. TREX [94] offers a solution for a tenant to define measurement templates, including measurement extraction point.

- *Traffic security*: when it comes to traffic measurement, the security of user data has high priority. The packet from users' applications may carry private information of the user, that even if encrypted, has not to be trapped into the hands of wrong network actors. A technique to assure data security during traffic collection is to restrict the amount of bytes to be collected, so it does not include payload, where sensitive data may be stored.

- *Low overhead*: the challenge of monitoring overhead has motivated the work described in this thesis.Throughout the literature, when proposing a new solution of monitoring, researchers investigate its possible overhead in terms of system resources, network bandwidth consumption or delays inserted, etc. In network monitoring having low-cost and high-accuracy measurement is the ideal-case scenario. The research community struggles towards that goal and proposes more and more efficient monitoring solutions.

In our work, we chose the commonly used network monitoring tools sFlow and IPFIX, known for their ability to provide multiple near real-time networking statistics. Their implementation allows to overcome scalability constraints, thus they are suitable for networks of diverse size. The measurement point is fixed for sFlow and IPFIX – it is a software switch, which forwards traffic of interest between virtual machines.

Considering the fact that these tools were designed to reduce monitoring cost by means of sampling, we were motivated to obtain a clearer picture on tools' needs in form of resources (network and server costs).

Additionally, we were driven by the fact that the footprint of virtual network monitoring solutions was presented differently every time. Some tools provide network-related overhead (link utilization, amount of reports, delays) [111], [6], [9]; others report system-side resource overhead (switch TCAM, server CPU, memory) [102], [47], [95], [5], [7], [62], etc.

In this thesis we start with an investigation of the monitoring tools under consideration, namely sFlow and IPFIX and their footprint in an experimental testbed. We noticed the

interference of the investigated tools in our setup with dependence on their configuration, as the amount of collected measurement could lead to degradation of virtual machines traffic, i.e., the user traffic. Similar throughput degradation in virtual networks because of measurement with sFlow is briefly mentioned in [115]. While proposing a network-wide monitoring service for clouds, the authors do not investigate further the phenomenon of sFlow impact on application traffic and only suggest to carefully choose its configuration parameters. In our work, we try to investigate the reasoning for this phenomenon further and propose to rely on machine learning to choose the best performing measurement parameters.

2.6 Machine learning in network measurement

Recently, various networking areas started to benefit from the usage of machine learning. A comprehensive survey [116] presents diverse machine learning techniques in various networking areas, including traffic classification and prediction, routing decisions, congestion control, resource and fault management, QoS and QoE management, and network security. Another fresh survey [117] focuses on current achievements of machine learning applied specifically to Software Defined Networking issues. Inspired by previous success of machine learning application in the domain of networking, we intend to use it towards the goals set in this thesis and provide an overview of machine learning and recent contributions concerning its usage for networking-related problems.

2.6.1 Introduction to machine learning

Machine learning (ML) is a set of methods, algorithms, and models enabling systems to exploit data and obtain knowledge. The goal of machine learning is to find and build hidden patterns in data, which will be further applied to analyze the unknown data. It is based on methods of mathematical statistics, numerical analysis, mathematical optimization, probability theory, graph theory and other techniques of digital data processing. Apart from computer science, machine learning finds its application in numerous areas: medicine, finance, linguistics, security and many more. The ubiquitous usage of information technology promotes data growth in science, production, business, transport, health services. The decision making and forecasting based on collected data become now a priority, and machine learning offers a set of directly applicable techniques to tackle those challenges.

There exist several types of machine learning techniques concerning the approach taken to solve the task in hand. The most known and used ones are:

- supervised learning: building a model by learning on the data with known inputs and desired outputs;
- unsupervised learning: building a model for data consisting only from inputs, trying to structurize data, discover patterns, groups or clusters;
- semi-supervised learning: implies modeling with the use of both supervised and unsupervised approaches, outputs only are available partially;
- reinforcement learning: one machine learning approach, where the system learns while interacting with some environment and situations and reinforces model with regard to taken decisions. Reinforcement learning may be considered as a supervised learning sub-type, however in this case "the supervisor" is environment and model itself;
- feature learning: some learning algorithms benefit from the better representation of the input data, e.g., which components and their correlation contribute the most to the model. It helps also to reduce the dimensionality of data, achieve higher accuracy, and find those features that overload the computation while being redundant or irrelevant.

As the studied phenomenon has predefined input values and experimentally obtained output, the problem considered in this thesis relates to supervised learning. It can be further grouped into regression and classification problems. A regression is used to operate with a continuous value, which has its numerical value. In this work, it will be used to provide a value of expected traffic loss under certain conditions of network and measurement. A classification task implies categorizing the data into classes. In this work we defined two classes, i.e., we will consider whether measurement impacts (or not) the virtual traffic.

Any ML model is built out of collected data. The collected data is divided into training, validation and/or test parts. A training dataset consists of data and associated labels. For the case of semi-supervised learning, a dataset can contain data with missing or incomplete labels. Once this training dataset is passed to the ML algorithm, the latter attempts to build a model to be further used with some new data and its identification. In the case of unsupervised learning, the whole dataset is unlabeled and the ML models will serve to find patterns. A

validation dataset is used to tune the parameters of the obtained model. A test dataset used to check the obtained model performance on previously unseen data.

The division in training, validation and test datasets can be performed with *holdout* or *k-fold cross validation* methods. With the *holdout* method a part of the dataset is used as training and a part remains for validation and/or test. In *k-fold* method the dataset is divided into k groups, each containing training, validation (or test) subsets. For each group, the model will be independently trained, validated and/or tested and the resulting evaluation scores will be calculated as an average.

To understand the performance of an ML model performance metrics may describe the model complexity, accuracy, and reliability. In our work to gauge the performance of our models, we will rely on the accuracy, mean absolute error (MAE), precision and recall metrics, described in Chapter 4.

Hereby, we briefly explain the idea behind learning algorithms that were used within the scope of our work:

1. k-Nearest Neighbour (kNN) consists of making a classification decision of a new sample with respect to its k nearest neighbors. More details in [118].

2. Decision Tree (DT) is a learning tree technique. It consists of nodes, branches, and leaves. Nodes represent features of the investigated data, branches are the combinations of features referencing to classification, and leaves are class labels. The classification of the new sample is performed by comparison of its features to nodes of the built decision tree. Decision Trees are known for their high accuracy of classification. More details in [119].

3. Random Forest (RF) bases its decisions on multiple decision trees, hence the name. It randomly chooses a subset of features to build multiple trees and decision about new sample is made considering the mode of the classes (for classification) or mean prediction (for regression) of individual trees. More details in [120].

4. Support Vector Machine (SVM) in brief finds the decision boundary and separates the feature space in the data. The classification is made depending on which side of this boundary the new data sample, i.e., its set of features, will appear. More details in [121].

5. Naive Bayes (NB) is a probabilistic classifier based on the Bayes theorem. New samples are classified according to the highest probabilities obtained. It can build the probabilistic model on relatively small training data. More details in [122], [123], [124].

2.6.2 Machine learning and networking

Recently, various networking areas started to benefit from the usage of machine learning. It opens new research opportunities in this domain and finds its application for networking-related problems [116]. Researches started to uncover the benefits of machine learning and its applicability in the domain of networking: the authors of [125] express their view on the future of network flow monitoring using machine learning methods. [116] also promotes machine learning as an interesting research area in networking. The authors focus on traffic engineering (traffic prediction, classification, and routing), performance optimization (congestion control QoS/QoE correlation, resource and fault management) and network security aspects of networking area.

Generally, DPI (Deep Packet Inspection) is often used when it comes to traffic classification. It has high accuracy, however, it is considered costly and it becomes more difficult to update the patterns with DPI with a constantly growing amount of applications. Additionally, by its nature, DPI is not able to classify encrypted traffic. This is where machine learning comes to play. ML appeared to be able to deal with encrypted traffic classification and as an additional advantage in terms of computational cost. [126] proposes a combination of DPI and machine learning to perform application-level classification in SDN. When a classification is required, the ML algorithms makes its decision first. If the level of certainty in this decision is higher than some predefined threshold, the traffic is classified as suggested. Otherwise, the DPI attempts to classify the traffic and if it is not able to perform classification, then the previous ML decision is applied. With this combination, method authors were able to achieve high classification speed and maintain a decent level of accuracy. Traffic classification with NetFlow data and machine learning is proposed by [127] and identification of host roles with supervised learning with sFlow in [128]. Deep learning and neural networks for the same case study are presented in [129], [130].

With the constant growth of applications in the Internet, QoS traffic characteristics may be used to divide applications into sub-classes according to QoS requirements (delay, jitter,

loss rate) on the way to clarify application identification. Leveraging QoS measurements for QoE modeling is proposed in [131]. Multiple works focus on QoS prediction [132], [133], as well as QoE prediction, e.g., a combination of DPI and semi-supervised learning is proposed in [134].

Elephant and mice flow classification is important for efficient traffic flow optimization in data centers. [135], [136] exploit machine learning to identify elephant flows for further treatment within data centers.

Following this trend, anomaly detection with machine learning catches a lot of attention recently:

- Coarse-grained intrusion detection. [137] – Hidden Markov Model (HMM) uses five flow features to identify malicious behaviour, [138] – two phase anomaly detection and classification with information theory, [139] – compare 4 machine learning algorithms to predict vulnerable hosts and connections, [140] – neural network model is used to classify traffic flows into normal and anomaly.
- Fine-grained intrusion detection. [141] – an SVM-based approach to categorize network attacks, [142] – deep learning-based intrusion detection method.
- DDoS atacks detection: [143], supervised (k-NN, Naive Bayes) and unsupervised algorithms (k-means, k-medoids) are used in [144]; neural networks in [145] and [146].
- Other security-related works: application software faults in SDN [147], firewall flow matching [148].

Anomaly detection in the cloud with machine learning is studied in [149], VNF anomaly detection in [150]. Encrypted traffic can be monitored by machine learning and [125] anticipates ML as a promising perspective for future developing in the network flow monitoring domain. [151] also recognizes the gap in Deep learning implication for computer networks and review deep learning enablers for network traffic control systems.

Authors of [152] introduce a new paradigm called Knowledge-Defined Networking, which relies on Software-Defined Networking (SDN), Network Analytics and Artificial Intelligence. They study several use-cases and show the applicability and benefits of adopting

the machine learning paradigm to the networking field, e.g., delay modeling of underlay network with machine learning for further routing improvement in overlay network or unsupervised machine learning applied to network logs to correlate events and logs and extend knowledge about the network.

The usage of the intelligence techniques applied to network-related needs is relatively new. It overcomes certain issues (heterogeneity, complexity, constant growth) faced in network management, but also brings new challenges to the domain. The exploitation of supervised machine learning remains fair until the dataset and labeling constraints enter the room. The performance of supervised learning algorithms relies on the training dataset. This is where other ML solutions can work better, e.g., semi-supervised learning could be introduced instead. It is able to learn from the smaller part of labeled data and can be useful for cases when the data collection period is limited or the prospective event requires fast reaction (e.g., fine-grained traffic classification and intrusion detection). Reinforcement learning is efficient for routing optimization and decision making. It does not need labeled data and can be adjusted according to optimization goals.

The challenge of training time and learning performance arises for the neural network (NN) models. The optimal NN architecture has to be chosen depending on the exact problem that is studied. Some problems, e.g., network security-related issues, may be sensitive to experimentation time. Timely identification of such issues is vital for network administrators to be able to prevent or react immediately to such kind of problems. To obtain the optimal model, the calibration of the neural network is important. For this purpose, it is necessary to invest some time for experimentation and train the neural network.

Nevertheless, the studies of all kinds of machine learning techniques in combination with computer networking, show that it has its advantages and perspectives. We were encouraged by the results obtained of this combination and found an inspiration to introduce machine learning as an effective approach to the problem that we faced in our work.

Chapter 3

Influence of Measurement on Virtual Network Performance

3.1 Introduction

Modern IT infrastructures heavily rely on virtualization with the so-called public or private clouds and cloud management tools such as OpenStack. The typical path taken by a packet sent from a virtual machine (VM) in a data center illustrates the complexity of such a set-up. The packet crosses the virtual network interface card (NIC) of the VM to reach a virtual switch where it is encapsulated, e.g., in a VXLAN tunnel [1], either to reach the remote tunnel endpoint (switch) before being delivered to the destination VM, or to a virtual router before leaving the virtual LAN. This complexity and blend of software and hardware equipments raise the difficulty to monitor and debug performance issues in such a virtualized environment. Monitoring and capturing traffic at the departure or arrival of its journey, i.e., at the first/last virtual switch, reduces the complexity of the task for the cloud provider or manager. It also allows to limit the impact on the physical switches that interconnect the racks. Still, it should be done carefully as the networking device (virtual switch) and the VMs share the resources (CPU, memory) of the same physical server. This key question has been overlooked in

[1] VXLAN stands for Virtual eXtended LAN and enables to have an IP network (VLAN) that spans over several segments, typically physical servers hosting VMs from different tenants. Tunnel endpoints are installed in each server and form a mesh. Each endpoint tracks the VMs it has in each VLAN, which enables at the end to map all the MAC addresses of VMs with the IP of the server hosting them.

previous studies, so that in this chapter we shed light on the interplay between measuring and delivering traffic in a physical server with VMs and a virtual switch.

We follow an experimental approach for the purpose of our study. We set up a testbed around an Open vSwitch (OvS) switch, which is arguably the most popular virtual switch nowadays, natively integrated in OpenStack and VMware, as well as in main Linux distributions. We consider the two typical levels of granularity of traffic collection tools, namely the packet level monitoring offered by sFlow [11] and the flow level monitoring offered by IPFIX [153].We aim at answering the following questions:

- What is the resource consumption (in terms of CPU) of those measurement tools as a function of the configuration parameters, e.g., sampling rate, granularity of measured data, and report generation time?
- What is the trade-off between the measurement accuracy and the system performance, i.e., the impact of measurement on the flows being measured (e.g., completion time, throughput)?

Our contribution can be summarized as follows:

- We explore the system resources consumption of a typical monitoring processes run in the virtual switches;
- We demonstrate the existence of a negative correlation between the measurement tasks and the operational traffic in the network under flow and packet level monitoring;
- We show that such an effect is not caused by a lack of system resources.

In Section 3.2 we describe our testbed. We present the results for the two use cases of sFlow and IPFIX in Sections 3.3 and 3.4, respectively. Section 4.5 concludes the chapter with a discussion and ideas for future research.

3.2 Test environment

3.2.1 Testbed

We follow an experimental approach to evaluate the impact of traffic monitoring tools on the host physical server and the measured applications. We consider a typical scenario with VMs interconnected by an OvS switch located in the same physical server, see Fig. 3.1. We consider a scenario where a virtual network is built by linking several Virtual Machines (VMs) through an OvS switch, all located inside a single physical server, as depicted in Fig. 3.1. Note that this set-up is typical of modern data centers where several VMs are placed on the same physical server. One or several virtual switches are then used to interconnect the VMs together and also to the outside by linking the physical interfaces with virtual switch.

The physical server runs a Ubuntu 16.04 Operating System, and has 8 cores with Hyper-Threading turned off at 2.13 GHz, 12 GB of RAM and 4 GigaEthernet ports. We use KVM to deploy the VMs (centOS) configured with 1 Virtual CPU (VCPU) and 1 GB of RAM each. We conduct experimentation with $\{2,4,6,8\}$ VMs to investigate how measurement tasks behave under different conditions of server CPU occupancy: from low utilization of physical resources (2 VMs with 2 cores dedicated to VMs) to a case where all eight cores are occupied (8 VMs). Half of VMS act as senders and half as destinations and the amount of traffic generated is directly related to number of VMs.

We use Open vSwitch (OvS) to interconnect the VMs due to its popularity. For traffic monitoring, we consider legacy tools natively supported by OvS, namely sFlow [11] and IPFIX [153]. The measurement collector is placed outside the server, in a remote node across the network. These tools collect information about network traffic and are presently supported by a wide range of network appliances across multiple vendors, including OvS.

sFlow is considered to be a scalable, light-weight solution for network monitoring [11]. It is a packet-level technology for monitoring traffic in data networks. In contrast to sFlow, IPFIX (stands for Internet Protocol Flow Information Export) [12] is a flow-based measurement tools, as it performs aggregation of sampled packets on flows on board of the switch, and reports statistics on flows rather than simply copying and exporting the headers of the sampled packets as in sFlow. Having said that, IPFIX can also operate at the packet level by disabling flow aggregation (by setting either flow caching or active timer to zero), thus reporting per-packet statistics. To some extent, one can see sFlow as an extreme version of

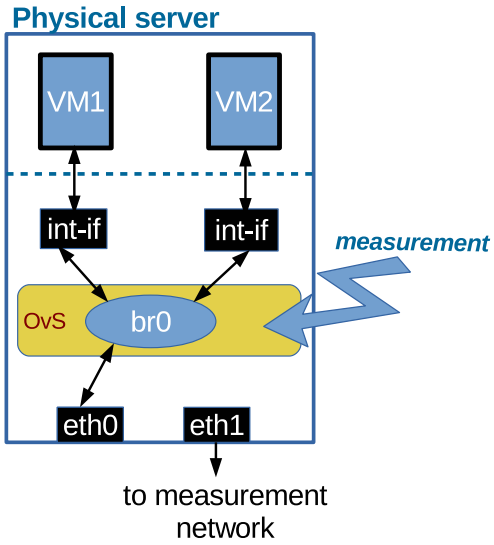


Fig. 3.1 Testbed for experimentation

IPFIX where aggregation on board is disabled. Based on that, we start by evaluating the overhead of sFlow in Section 3.3, then we move in Section 3.4 to a comparison between the two measurement approaches in terms of their load on the physical server and the impact they incur for the application data plane.

3.2.2 Traffic workload

In our testbed, traffic between the VMs is produced with two tools: *flowgrind* [154] and *iperf3* [155]. Each experiment is run 10 times to smooth the results. In general, the different runs of each experiment are producing close results, as we operate in a closed and controlled environment. We detail below the way these two benchmarking tools work and the parameters we used.

Flowgrind

Flowgrind [154] is an advanced TCP traffic generator for testing and benchmarking Linux, FreeBSD, and Mac OS X TCP/IP stacks It has distributed architecture with two components: controller and daemon. We use it for its ability to generate TCP traffic following sophisticated request/response patterns. Indeed, in flowgrind, one single long-lived TCP connection is supposed to mimic transactional Internet services such as the Web. Hence, a TCP connection

in flowgrind consists of a sequence of requests/responses separated by a time gap between the requests. The request/response traffic model in flowgrind is controlled with the following four parameters:

- Inter-request packet gap;
- Request/response size;
- Request/response size distribution;
- Number of parallel TCP connections.

We use a fixed inter-request gap equal to 10^{-4}s in our experiments. Requests and responses are sent as blocks, with request and response messages fitting in one or several IP packets. While keeping the request size constant at 512B, we vary the response size (using a constant distribution, meaning that all responses have the same size) to achieve different rates, both in packets/s and in bits/s, as presented in Table 3.1.

Table 3.1 Flowgrind parameters used for traffic generation

Response size, bytes	Throughput, Mb/s	Throughput, packets/s
1024	80	16000
2048	160	26000
3072	240	40000
4096	320	45000

iPerf3

IPerf3 is a tool to measure network bandwidth operating with TCP, UDP and SCTP transport protocols. It performs bandwidth tests between server and client hosts, establishing traffic transfers with various parameters to be tuned (timing, protocols, buffers). We use iPerf3 to generate TCP traffic at maximum achievable rate (10 Gb/s) to investigate whether monitoring with sFLow may cause any kind of impact on the workload produced between VMs. For IPFIX and sFLow comparison we also generate UDP traffic at constant and controlled rate of 100 Mb/s to minimize data plane interference. The exploited traffic pattern is depicted in Table 3.2.

Table 3.2 iPerf3 parameters used for traffic generation

Protocol	Throughput, Mb/s	Throughput, packets/s
TCP	10000	25000
UDP	100	8600

3.3 Measuring with sFlow

We focus in this section on sFlow. Its behavior is mainly driven by the following parameters:

- *Sampling rate*: Ratio of packets whose headers are captured and reported by sFlow.
- *Header bytes*: The number of bytes reported by sFlow for each sampled packet. The default value in our experiments, unless otherwise stated, is equal to 128 bytes. More than one header can be aggregated in one UDP datagram, depending on the configured header size.
- *Polling interval*: sFlow also reports aggregate port statistics (byte and packet counters) to the collector. This parameter models the frequency at which sFlow reports these statistics.

3.3.1 Initial experiment: measurement plane vs. data plane

We conducted a first experiment with a long lived TCP flow generated with iPerf3. The experiment lasts 600 seconds and the sampling rate is changed (or disabled) every 100 seconds. We plot the throughput achieved by iPerf3 in Fig. 3.2. Note that with zero packet sampled (i.e., no sFlow measurement), we can reach a throughput of up to 10 Gb/s as TCP generates jumbo packets of 65 KB. It is then clear that monitoring with sFlow can influence the application traffic and that this influence depends on the level of sampling. Finding the right sampling rate to use is a trade-off between desired monitoring accuracy and expected application performance. In the next sections, we explore this trade-off in more details and try to understand whether this result is due to a lack of resources or to the implementation of sFlow in OvS.

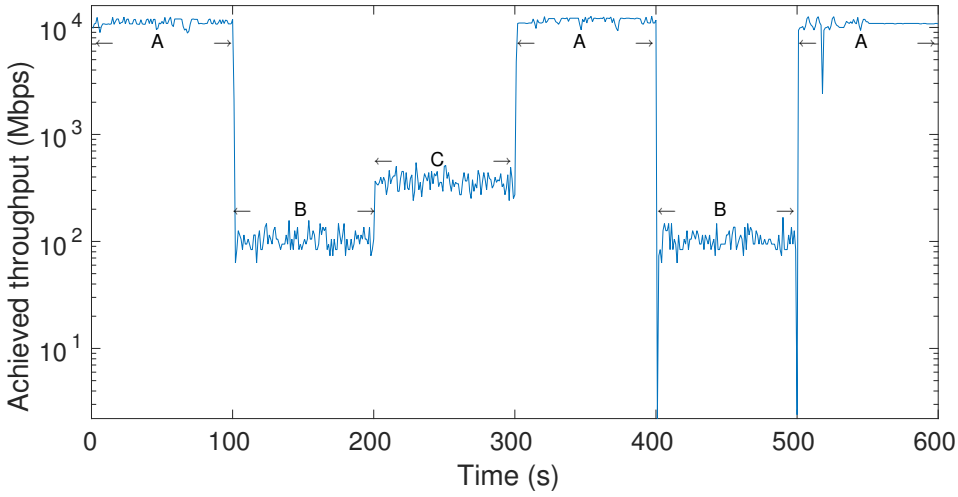


Fig. 3.2 iPerf3 throughput at different sampling rates of sFlow: no sampling (A), 100% sampling (B), 50% sampling (C).

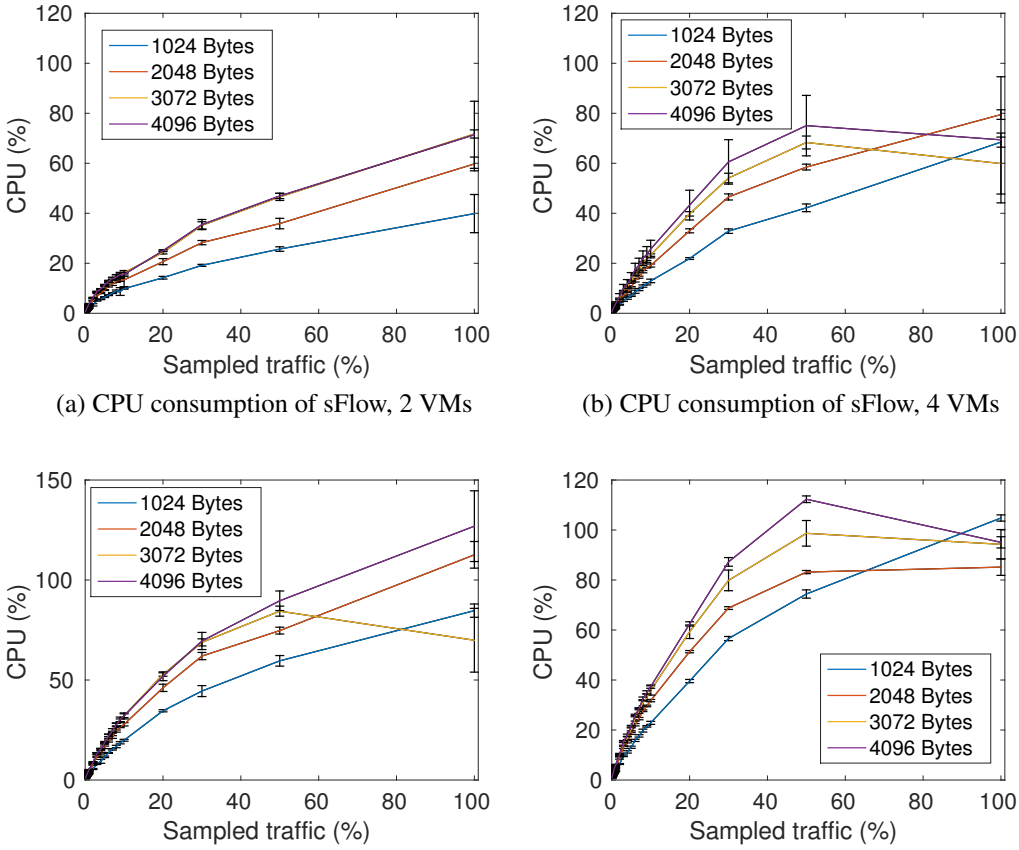
3.3.2 Resources consumption and competition

In this section, we vary the sampling rate and observe the CPU consumption of sFlow summed over all CPUs (hence potentially from 0 to 800%, as we have 8 CPU) as OvS uses multi-threading and can run over different cores. Fig. 3.3 reports the results obtained with the flowgrind workload, as described in Table 3.1. Note that the sampling rate is expressed in sFlow as the number of packets (the unsampled ones plus the sampled one) between each two consecutively sampled packets. This means that when $sampling = 1$, we sample every packet (100% rate), $sampling = 2$ we sample one packet out of two (50% rate), and so on.

We can make two observations from Fig. 3.3. First, the CPU consumption increases with both the sampling rate and the traffic rate, in line with intuition. Still, at 100% sampling, we observe a decrease of CPU consumption in some cases (note that due to the implementation of the sampling parameter in sFlow, there is no value between 50% (one packet out of two) and 100% (every packet) in our graphs). We defer the study of this phenomenon to the next section.

Second, while OvS is multi-threaded, a single thread is used to handle sFlow measurement task and the utilization of the core where sFlow operates is high in our experiments. Considering high value and sublinear increase of CPU consumed by the tool at high sampling rates, one can wonder whether interference with monitored traffic exists. This interference is indeed visible in Fig. 3.4 when plotting the number of packets generated by flowgrind for

the different sampling rates and for the different traffic profiles. We report the results for the case of 2, 4, 6 and 8 VMs. Normally, in the absence of interference, this number should stay constant whatever the sampling rate is, which is not the case in the figure, especially when the sampling rate gets close to 100%. This decrease in the number of generated packets points to a possible interference caused by the CPU consumption of sFlow. Having less data packets at high sampling rates can also explain the sublinear trend of CPU consumption with sampling rate observed in Fig. 3.3.



(a) CPU consumption of sFlow, 2 VMs

(b) CPU consumption of sFlow, 4 VMs

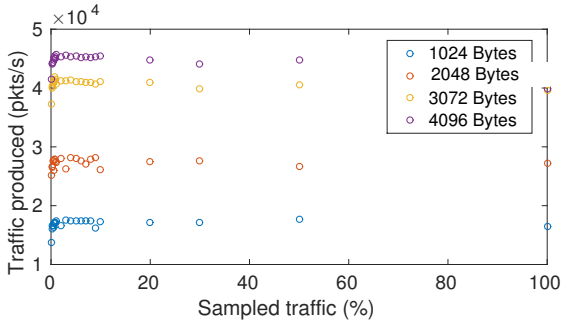
(c) CPU consumption of sFlow, 6 VMs

(d) CPU consumption of sFlow, 8 VMs

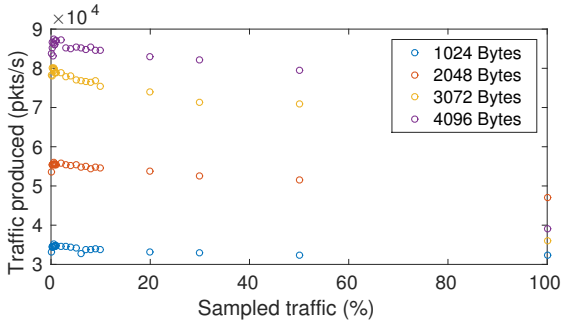
Fig. 3.3 CPU consumed by sFlow vs. sampling rate (flowgrind)

3.3.3 High sampling rate anomaly

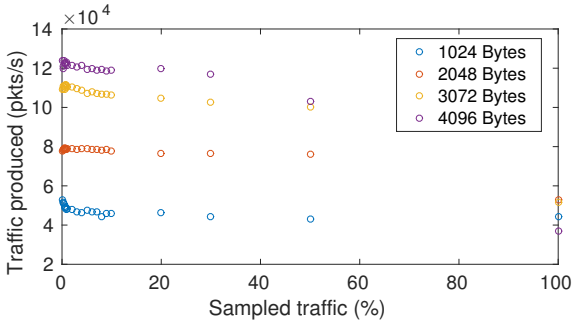
To better understand the performance anomaly aforementioned, we looked at how the OvS measurement process was utilizing the available cores in the case of 4 VMs, where there should be enough resources for both measurement process and VMs generating traffic. In an



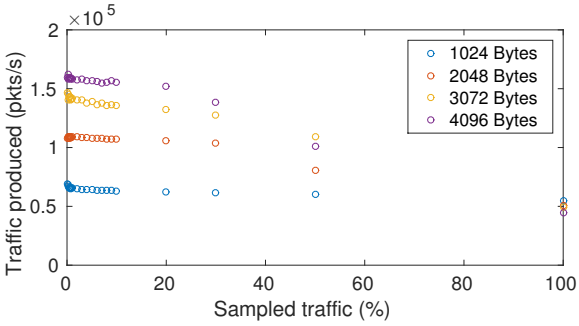
(a) Traffic generated for 2 VMs



(b) Traffic generated for 4 VMs



(c) Traffic generated for 6 VMs



(d) Traffic generated for 8 VMs

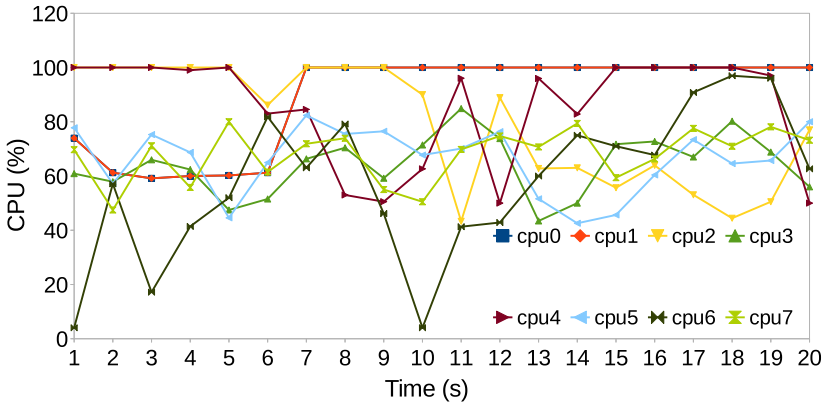
Fig. 3.4 Nb. of packets generated vs. sampling rate (flowgrind)

attempt to better understand the performance anomaly at high sampling rate where the CPU consumption of sFlow is decreasing for some traffic rates, and the user traffic is affected, we first studied cases where there should be enough resources for both the measurement process and the VMs generating traffic (less than 4 VMs). We looked at how the OvS measurement process was utilizing the available cores. We used *pidstat* to track the core utilized and its usage by sFlow.

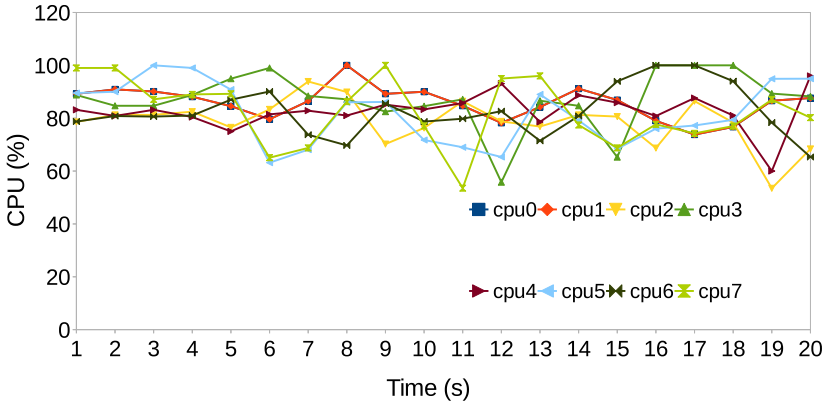
We observed that while there should be almost no competition between the iPerf3 (embedded inside single-core VMs) and OvS processes, as there are 8 cores in the server, the OvS process was regularly scheduled to a different core by the Linux scheduler. To check if this variable allocation results in suboptimal performance, we pinned each VM and the OvS process to a specific core using the *taskset* utility. Fig. 3.5a and Fig. 3.5b portray a single experiment for the case of 4 VMs (for traffic of 320 Mb/s and sampling rate of 100%), and shows a clear improvement in terms of core utilization variability. Still, the impact on user's traffic was observed to be similar. We can thus conclude that the OS scheduler is not the cause behind the performance problem we observed.

The experiments that we have carried out allowed us to make two main observations: first, when activated on an OvS switch, sFlow mainly impacts the CPU of the physical host; and second, most of the sFlow operations are done at the user-level space. As packets and segments of packets are gathered from the kernel space, all the remaining operations (e.g., data buffering and timers management) are executed by processes running at the user space. This means that monitoring the sFlow consumption is made easier as one can directly profile the OvS user side process.

Fig. 3.6 provides a high level overview of the way OvS implements sFlow. The actual switching (forwarding) of packets is done in the kernel. OvS further features two user level processes. First, *ovsdb-server*, which is in charge of storing the configuration of the OvS switches in the machine and is not of interest for us here. Second, the *ovs-switchd* daemon process, which delegates the sFlow measurement to one of its threads. Hence, to implement sFlow, each sampled packet must move from the kernel to the user space, which is likely to be a problem at high traffic and high sampling rate. A better understanding of the phenomenon we observed would require a precise profiling of the OvS code, which is out of the scope of our work. Indeed, we suspect the bottleneck to manifest at the boundary between the kernel space where forwarding is done by OvS and the user space where sFlow operates, hence slowing down the rate of traffic going through both of them.



(a) OvS is **not** assigned to a specific core



(b) OvS is assigned to a specific core

Fig. 3.5 CPU load with/without pinning OvS to specific core

3.3.4 Varying sFlow parameters

The effect of the sampling rate on the CPU consumption has been investigated in the previous section. Increasing the sampling rate was shown to increase the CPU consumption to large values, but also to impact the monitored applications themselves pushing them to reduce their throughput. In this section, we extend the study to the other parameters of sFlow to show their impact as well. We cover in particular the impact of the header length and the frequency of interface statistics reporting, which are appended to packet samples by sFlow at the desired time interval. To evaluate influence of these two parameters, we show results for the set-up with four virtual machines, as in this case the physical server is under medium utilization: four cores are assigned to four virtual machines and the remaining four cores

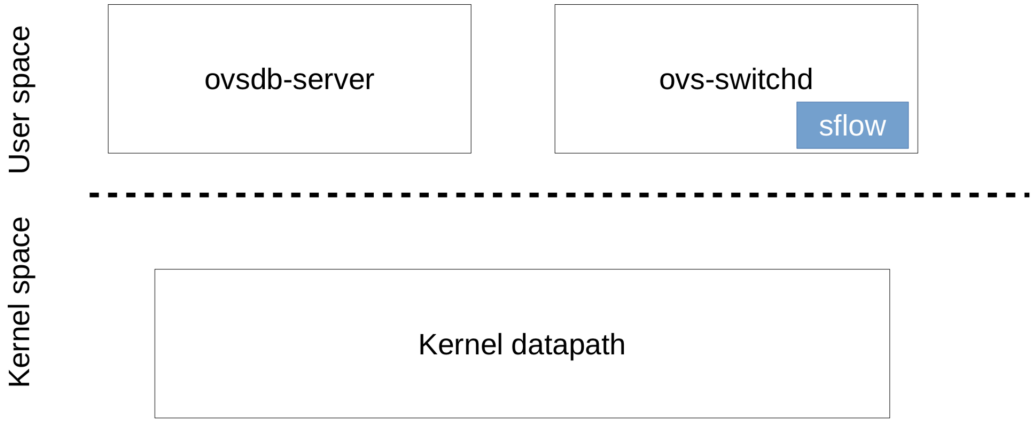


Fig. 3.6 OvS architecture with sFlow measurement process

of the physical server are left for the proper operation of the hypervisor, OvS and the other system services.

Header length

sFlow reports contain the first N bytes of sampled packets. The sFlow implementation in Open vSwitch samples by default the first 128 Bytes of each packet. Hence, if a TCP packet is sampled, assuming there is no options at the IP header, nor at the TCP header, sFlow will record: the 14 Bytes of the MAC header, the 20 Bytes of the IP header, the 20 Bytes of the TCP header and up to 74 Bytes from the application payload.

We vary the header length and assess its impact on the system performance with the help of a flowgrind workload of response size equal to 1024 Bytes in our set-up of 4 VMs.

Several reports (packet samples or statistical reports) compose an sFlow datagram. Depending on its size, more or less reports may fit into one datagram. Maintaining fewer or more reports does not matter for the software switch, as finally they are to be encapsulated into one datagram. Moreover, the process of sending datagrams to the collector does not incur a computational burden in terms of CPU. It follows, and according to what we see in Fig. 3.7, that changing the header length does not impact the CPU utilization. In this figure, we explore values of header length up to 512 Bytes, which is the recommended maximum

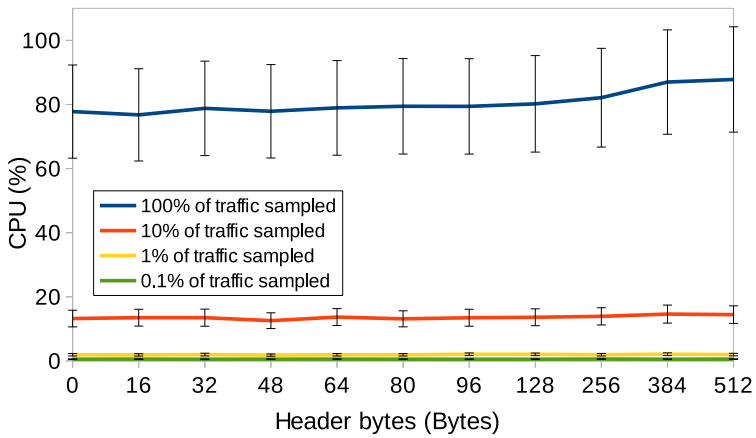


Fig. 3.7 CPU consumed by sFlow for different header lengths and different sampling rates

header length in most sFlow implementations. Next, and if not explicitly mentioned, we restrict ourselves to the default header length of 128 Bytes in our experiments.

Polling period

The polling parameter refers to the time interval (default 30s) at which sFlow appends to its reports aggregate traffic statistics (total-packets and total-samples counters) associated with the different interfaces of the virtual device, which can be either ingress or egress interfaces. These reports are usually piggybacked with sampled packet headers, but as sampling is random, and to avoid periods of no reporting, the sFlow agent can be configured to schedule polling periodically in order to maximize internal efficiency. According to sFlow developers, the polling interval should not have a big influence on CPU consumption. To confirm this statement, we performed experiments with varying polling intervals under different flowgrind workloads and sampling rates. Results were similar for the considered workloads. We report in Fig. 3.8 the case of a flowgrind workload of about 80 Mb/s. We can observe in the figure that varying the polling interval from 1 second to 30 seconds does not induce any additional CPU overhead. This is because the sFlow agent opportunistically inserts the counters into sFlow datagrams together with samples. If many packets are to be sampled, counters may not be even included as frequently as configured.

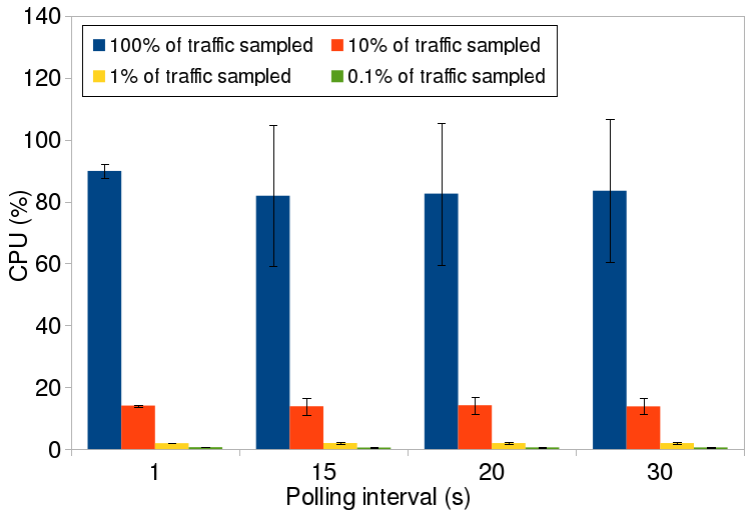


Fig. 3.8 CPU consumed by sFlow for different polling intervals

3.4 Measuring with IPFIX

As explained earlier, IPFIX and sFlow are two representatives of the two main classes of traffic monitoring approaches: the flow level approach and the packet level approach, respectively. sFlow was designed to limit the processing at the switch/router side by simply forwarding packet headers to the collector while IPFIX maintains a flow table to aggregate packets in flows, then reports on flows rather than on packets. This normally should entail a higher processing load but a smaller network footprint. Next, we compare the two tools. We configure sFlow with its default header length of 128 Bytes. The size of each IPFIX flow report is equal to 115 Bytes.

3.4.1 Running IPFIX without flow aggregation

We first configured OvS to send one IPFIX flow record per packet sample to the collector. The task of IPFIX is thus similar to the one of sFlow in this case.

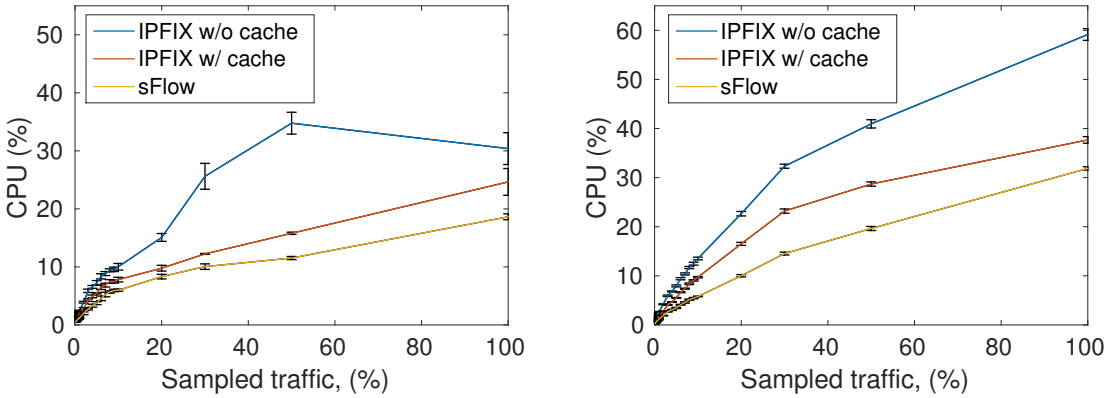
There exist two caching options for IPFIX in OvS:

- *cache active timeout* – maximum period for which IPFIX flow record is cached and aggregated before being sent;

- *cache maximum flows* – maximum amount of IPFIX flow records that can be cached at any time.

To ensure per-packet flow exporting, the caching feature of IPFIX is disabled and the active timers are set to zero.

As there is no flow aggregation in this specific experiment, we consider a scenario with one long run TCP flow produced by iPerf3 at a rate of 100 Mb/s. Results are reported in Fig. 3.9a where we compare sFlow to IPFIX for different sampling rates, both in terms of CPU consumption and total number of reports. We focus in this section only on the IPFIX w/o cache case. Clearly, and because of packet processing on-board (in the switch), IPFIX consumes more CPU than sFlow. The latter simply reports headers without on-board processing.



(a) CPU consumption of IPFIX and sFlow (iPerf3 at 100 Mb/s)

(b) CPU consumption and total number of reports of IPFIX and sFlow (iPerf3 with 1000 flows)

Fig. 3.9 IPFIX vs. sFlow resource consumption

Similarly to the case of sFlow, IPFIX also impacts the traffic forwarding function of the virtual switch, as presented in Fig. 3.10. For this, we use the same set-up as for the sFlow experiment in Fig. 3.2: 600 seconds of TCP iPerf3 traffic where sampling rate is changed (or disabled) every 100 seconds. For the same sampling rate and by comparison with Fig. 3.2, IPFIX has a more pronounced impact on the achieved rate than sFlow in this configuration, in line with its higher CPU consumption.

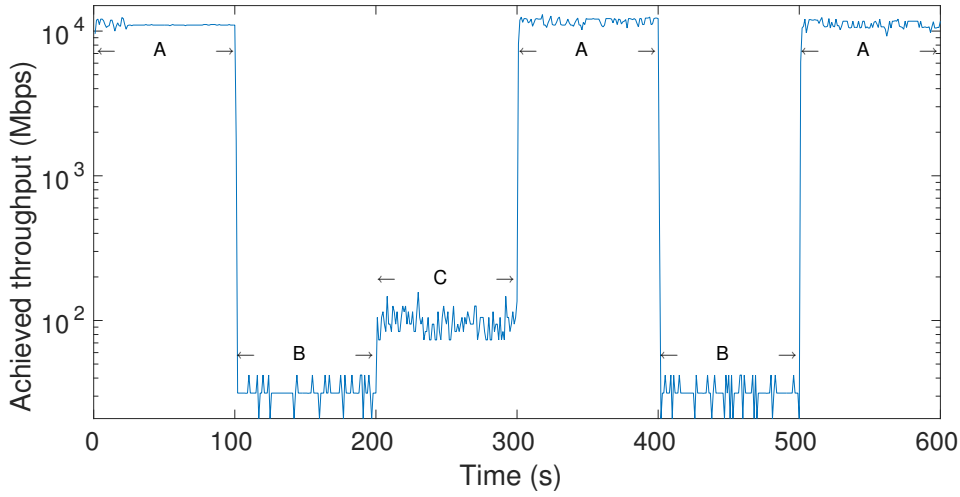


Fig. 3.10 iPerf3 throughput at different sampling rates of IPFIX: no sampling (A), 100% sampling (B), 50% sampling (C).

3.4.2 Introducing cache and flow aggregation

The usual way of configuring IPFIX is with a non-zero value for caching and aggregation. This allows to aggregate several packets from the same flow (TCP or UDP conversation) into a single flow record. When a flow is aggregated on board of the switch, only one report about this flow is sent containing aggregated statistics on it.

Considering the results from the previous section (without aggregation), the next natural question is to evaluate how flow aggregation impacts CPU consumption, as flow aggregation induces more computation on the virtual switch while reducing the network footprint of the measurements.

In a first experiment, we consider the same workload as in the previous section (one long run UDP flow generated by iPerf3) and tune IPFIX caching capacity to cover the entire experiment, hence reducing drastically the number of reports: only one report is sent at the end of the experiment. By doing so, we reduce to almost zero the networking cost of reporting and we only leave the cost of flow aggregation. Results for this experiment are reported in Fig. 3.9a along with those of sFlow and IPFIX without caching. The cost of on-board processing is, as expected, smaller with caching than without caching for IPFIX. What is striking here, however, is that IPFIX still consumes more CPU than sFlow even when it does not send reports. Note that IPFIX, similarly to sFlow, is implemented in the user space. It is thus normal that IPFIX suffers from the same performance problem at high sampling

rate as sFlow (we did not report the impact on traffic of the IPFIX measurement process, but it is similar to the one of sFlow). We can thus conclude on the benefit of aggregation in IPFIX, but also on the importance of the CPU cost of IPFIX, which for its two variants (with and without caching), remains more greedy than sFlow in terms of CPU consumption.

We performed a second set of experiments with a richer workload in terms of number of flows (1000), which is more challenging for IPFIX as it induces more computation on the OvS side. Comparing the results of Fig. 3.9a and 3.9b, we can observe that the CPU consumption of IPFIX without caching has slightly increased as compared to the single flow experiment. A further increase in the number of flows could have led to observe a more pronounced difference between IPFIX and sFlow. However, at the scale of physical servers hosting a few tens of VMs in a data center, 1000 flows is already a reasonably large value.

3.5 Conclusion and discussion

We have investigated the influence of virtual network monitoring on the physical server performance and the throughput of monitored applications. Two legacy monitoring tools were considered, sFlow and IPFIX. We performed a sensitivity analysis of CPU consumption and network footprint of the two tools regarding different traffic profiles and monitoring configuration parameters.

Among the set of influencing parameters, sampling rate and traffic throughput in packets per second are the two dominant factors. Indeed, both cause an increase in the number of samples to be generated, thus leading to an increase in the physical resources consumed at the virtual switch. As for sFlow, the polling interval (of counters) induces no CPU overhead for the virtual switch, as counters sent within this interval are small and this interval may be adjusted by the sFlow agent for efficiency reasons. IPFIX appears to be more expensive than sFlow (in terms of CPU consumption) because of its on board flow aggregation feature. Finally, we observed interference between monitoring tasks and monitored traffic, as the cost of monitoring transforms into reduced throughput for the monitored applications.

To reduce the load on the CPU and the impact on regular traffic, the best option is to tune the sampling rate in a way to balance between monitoring accuracy and application performance. Finding this optimal sampling rate is an interesting future direction for our work. We also intend to study an alternative approach whereby the OvS switch would simply

mirror traffic to an external measurement process embedded in a dedicated virtual machine. This approach could consume more networking resources but has the potential to alleviate the impact on the regular traffic that we intend to measure.

To reduce the load on the CPU and the impact on regular traffic, the best option is to tune the sampling rate in a way to balance between monitoring accuracy and application performance. Finding this optimal sampling rate is an interesting future direction for our work. An alternative approach could be the case when the OvS switch would simply mirror traffic to an external measurement process embedded in a dedicated virtual machine. This approach could consume more resources but has the potential to alleviate the impact on the regular traffic that one intend to measure.

Chapter 4

Tuning Optimal Traffic Measurement Parameters in Virtual Networks with Machine Learning

4.1 Introduction

The ability to perform traffic monitoring in virtual networks is a key instrument in the troubleshooting toolbox of both cloud tenants and providers. sFlow [11] is the reference tool that allows to perform such traffic monitoring in virtual networks based on Open vSwitch (OvS) [32]. The tool has been initially designed for physical switches and routers, then adapted to virtual networks. With sFlow, network equipment *samples* the packets at a rate indicated by the user before packing them (after a controlled truncation) into an sFlow packet that is sent to a dedicated machine called the *collector*. The key idea is to minimize the amount of work done at the network equipment and leave the analysis to the collector itself. The sampling rate can range from a few percent to a full capture, with the latter similar to port mirroring. As an example, Facebook continuously monitors its data centers servers at a rate of 1 out of 30,000 packets with a tool akin to sFlow [13].

In a previous chapter, we have demonstrated that traffic monitoring with sFlow costs not only in terms of the CPU cycles of the physical system (where the virtual network is deployed), but also causes reductions in the throughput of the operational traffic of the embedded virtual network. Fig. 4.1 illustrates the case: we can observe that the traffic

between two Virtual Machines (VMs) inside the same physical node decreases in terms of throughput immediately after sFlow is turned on, with a penalty that is proportional to the configured sampling rate. We refer to this throughput reduction as *drop of throughput* or, alternatively, *impact of sFlow sampling*.

To determine the root cause of this interference, we were questioning the system resource limitations for OvS and sFlow (server CPU, context switching); however it appeared that such limitations do not exist, and that the impact is coming from other limitations related to the operating system and the way it handles the data path of OvS and the forwarding of sampled packets between OvS and sFlow. Even though the nature of such interference is not clear, we believe that the footprint of network monitoring needs to be reduced to the minimum, or at least to be well modeled so that this footprint could be better anticipated and controlled. The best option would likely be to optimally tune sFlow monitoring parameters (sampling rate in particular) so as to alleviate traffic disturbance while providing a good monitoring service at the same time. However, the exact correlation between different monitoring parameters and their impact on the application traffic is non-obvious. Certain Machine Learning techniques (e.g., decision trees) could assist us in uncovering such a correlation, which would enable us to tune the monitoring parameters optimally.

Considering these findings, in this chapter we aim to propose a solution that is based on Machine Learning to *(i)* identify potential drop in throughput due to traffic measurement and *(ii)* automatically tune monitoring parameters so as to limit the measurement and traffic interference in the virtual environment. The objective is not to exceed a desired level of throughput reduction in a context where throughput varies, so the parameters of monitoring have to be adapted accordingly.

Next, we summarize the related work, then we describe our methodology based on data analysis and machine learning and provide an overview of our dataset. In Section 4.3 we present an offline study on our dataset where we model the relationship between throughput and throughput loss given the sampling rate. Section 4.4 builds upon the offline analysis to propose an online variant; it iteratively learns from previous experiences (with monitoring and its impact) to build a model, which is able to pinpoint the optimal tuning of sampling rate in sFlow, so that the impact of monitoring is limited to a certain desired level. We validate the online variant with two synthetic traces we built for the purpose of the study and compare its performance to the offline variant. We conclude the chapter in Section 4.5 with some perspectives on our future research.

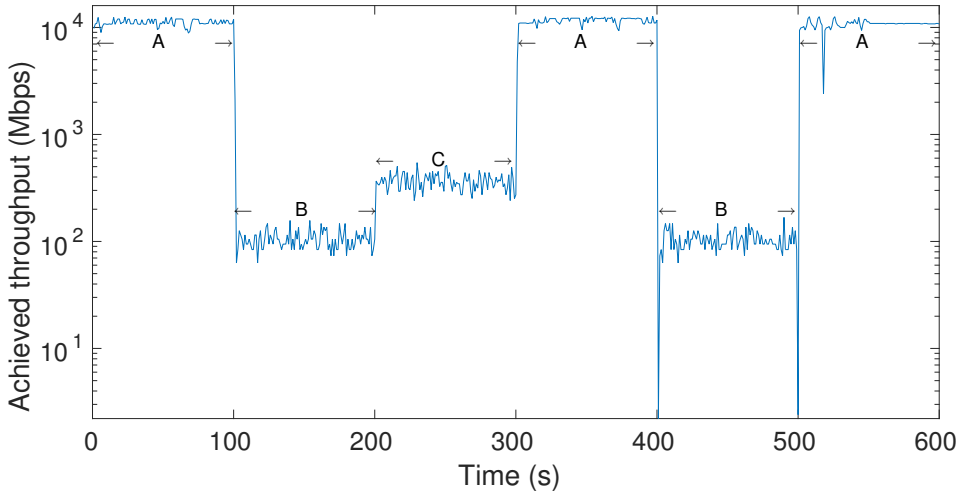


Fig. 4.1 Traffic throughput at different sampling rates of sFlow: no sampling (A), 100% sampling (B), 50% sampling (C).

4.2 Dataset construction and methodology

4.2.1 Methodology

In this chapter, we address two problems, namely a classification and a regression problem, using machine learning. The classification problem can be cast as follows: given a sampling rate S (in percent), a traffic rate R (in bps) measured when there is no sampling (the reference throughput), a drop in throughput of I percent because of sampling at rate S (which then becomes $R \times (1 - I/100)$), and a maximum authorized impact level thr_I (in percent), the question is to know whether I is less or greater than thr_I. This problem can be solved using binary classification supervised machine learning techniques such as Decision Trees and Bayesian Networks.

The regression problem however is about the modeling of the impact I itself. This regression problem, which can be solved by regression machine learning techniques such as Random Forest, will open the door to our data-driven optimization which consists in finding the optimal sampling rate to use in sFlow such that the (estimated) impact I does not exceed the maximum authorized level thr_I.

These two problems are investigated throughout this chapter in an offline and online set-up. The offline set-up, where we train a classifier on a representative traffic dataset, enables to assess whether we can predict the impact of sampling in sFlow and to select the

appropriate machine learning algorithm among a set of candidates. We consider several algorithms available in the scikit-learn library [156]:

- Decision Tree;
- K Nearest Neighbors;
- Naive Bayes;
- Random Forest;
- SVM.

The online set-up targets more an operational scenario where the model estimating the impact of sampling (i.e., I) is built online with the objective to make it adaptive to physical server characteristics. We focus on building iteratively this model for the regression case, then show how such model can be used to optimally set the sampling rate. This will allow us to answer the question of which sampling rate to use so as to limit the disturbance of the ongoing traffic.

Our evaluation is done using traffic traces captured in a controlled virtual network environment that we describe in the next section together with our dataset.

4.2.2 Dataset

The training and testing datasets (that we used to build and estimate the performance of the learning algorithms) were collected within a dedicated experimental set-up. Our set-up consists of one physical server with 8 cores, 12 GB of RAM, and N virtual machines (VMs) interconnected with an OvS switch. Traffic between the VMs is generated with iPerf3 [155], where half of the machines act as senders and the other half as receivers. We generate UDP traffic for a set of predefined throughput values and packet sizes, and we apply the same configuration to the different iPerf3 senders. Therefore, the total amount of traffic generated is directly proportional to the number of VMs. We do not use TCP as we do not want the transport layer to adapt to the changing network conditions that result from the usage of sFlow. While our VMs are exchanging traffic between each other, we turn on sFlow on the OvS switch to collect network statistics for a set of predefined sampling rates S. We then measure the achieved throughput and compare it to the input traffic rate R to be able to

calculate the drop in throughput I, if any. Our resulting datasets are thus composed of two sets of features: traffic-related features (number of VMs, input traffic rate, packet size) and measurement-related features (sampling rate, throughput reduction). We use the following values:

- sampling rate S: either disabled or gradually increasing from 0.1% to 100%;
- number of VMs $N \in \{2,4,6,8\}$;
- packet size: 128B, 256B, 512B, 1024B, 1448B;
- input traffic rate in bps per iPerf sender: from 100 Mbps to 1000 Mbps with steps of 100 Mbps;
- input traffic rate in packets per second.

Each experiment, consisting of one combination of the above parameters, is repeated 10 times to remove any bias and smooth average values. We perform experiments for all the above values of number of VMs, input traffic rate and sampling rate. As for the packet size, and because of the impossibility to accommodate small packet sizes at high rates in bps, we limit the experimentation of packets smaller than 1448B to an input traffic rate equal to 100 Mbps, and scan the entire range of input traffic rate defined above for only large packets of 1448B. It follows that our dataset consists of approximately 13,000 experiments that we split between a training set and a testing set as described next. Lastly, for the part on binary classification, we consider threshold values (i.e., thr_I) of the impact I ranging from 1% to 25%.

Fig. 4.2 gives a flavour of this dataset, where we show the impact as a heatmap versus the number of VMs and the sampling rate. Clearly, the more we go to the top right corner, the darker the colour as the impact has tendency to increase with the traffic load on the physical server and the sampling rate. We use the same dataset to validate both the offline and online models for impact I, though in a different manner. In the offline case, and to avoid any bias during the learning phase, we replace the 10 repeated experiments of the same scenario by a single experiment, where the impact is the average observed impact. This means that the offline dataset consists of 1,300 experiments. In the online case, we emulate the actual traffic variations observed in an operational scenario by concatenating the 13,000 experiments in a controlled manner so as to emulate either large or small traffic variations over time (more

details in Section 4.4). For each case, we split our dataset into two parts for training (80% of dataset) and testing (the rest 20% of dataset). We train our models on the training dataset and assess their prediction accuracy on the testing dataset.

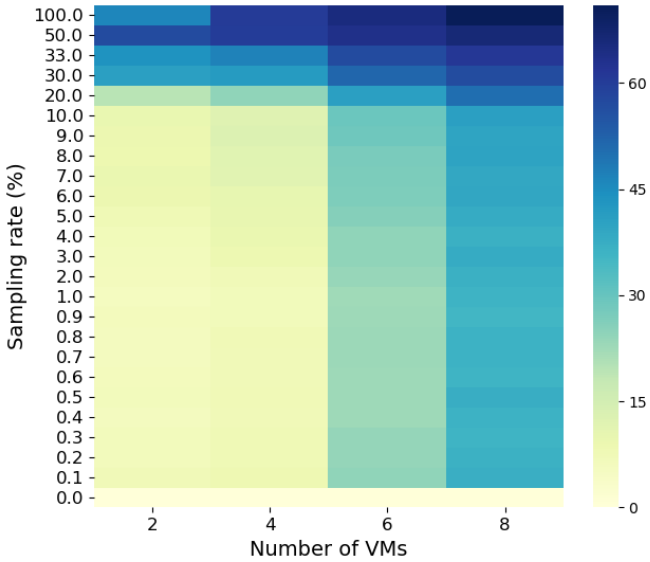


Fig. 4.2 Sampling impact vs. sampling rate and number of VMs

Handling the balance of our classes

Machine learning is sensitive to the distribution of classes in the datasets. If one class is prevalent over the other, the learner may make biased decisions towards the majority class. As we are using different values of impact threshold values, our dataset may show a disproportion between the two classes (i.e., impacted scenarios versus non-impacted ones). Fig. 4.3 depicts the distribution of classes in our dataset with respect to threshold thr_I. While the dataset shows a good balance between the two classes for large threshold values, there is a clear unbalance at low values of threshold (below 10%). To counter this unbalance, we use a well-known technique in the literature called *Random Undersampling* which consists in randomly extracting instances from the majority class until the two classes are equal. Hereafter, and for the offline study, we compare our classification models on both balanced and unbalanced datasets.

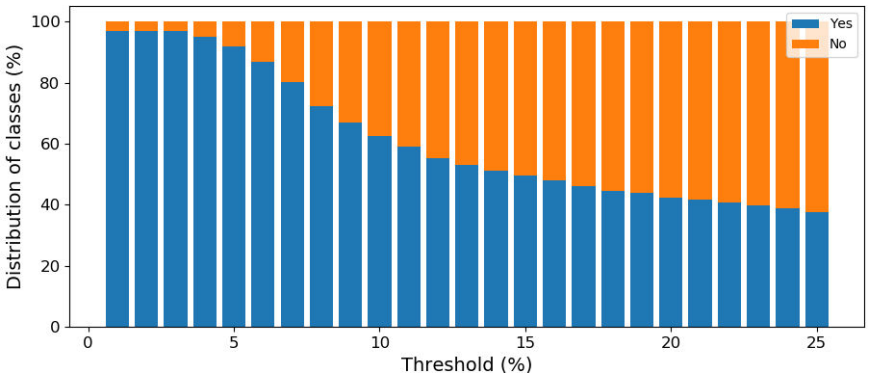


Fig. 4.3 Unbalanced dataset: classes distribution vs threshold

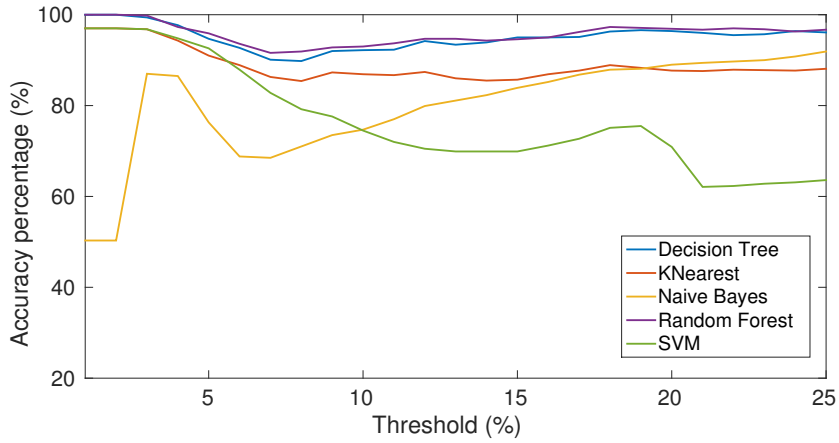
4.3 Offline analysis

In this section, we investigate the ability of different machine learning algorithms to detect and estimate the impact of turning on sampling in sFlow on the application traffic, given a wide range of experimental scenarios (i.e., traffic rate, number of VMs and sampling rate).

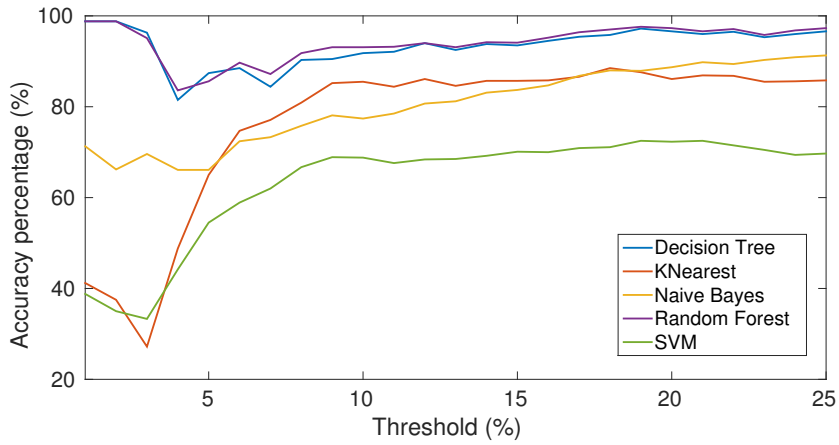
4.3.1 Detecting the impact of monitoring

The presence or absence of impact depends on the threshold of acceptable throughput reduction thr_I. For a given threshold, we can split our dataset into two classes: the YES class when there is impact (i.e., $I > thr_I$) and the NO class when there is no impact (i.e., $I \leq thr_I$). Our objective is to build a model that can detect in what "case" the system is running.

We train and validate the different machine learning algorithms listed in Section 4.2.1 using the classical 10-fold cross-validation technique. We report each time the results for both the balanced and the unbalanced datasets, the former being threshold dependent. We first evaluate the algorithms using the accuracy metric, which is defined as the fraction of correctly classified YES and NO instances. Fig. 4.4 reports the achieved accuracy of prediction for both unbalanced (Fig. 4.4a) and balanced (Fig. 4.4b) datasets. The two most accurate classifiers appear to be the Decision Tree and the Random Forest classifiers. They perform at a minimum of 90% and 92% of accuracy for certain thresholds with an unbalanced dataset; and 81% and 78% for a balanced one. We can thus conclude that these models are able to correctly capture the impact of sFlow on application traffic.



(a)



(b)

Fig. 4.4 Accuracy of different classifiers vs impact threshold for unbalanced (4.4a) and balanced (4.4b) datasets.

We now focus on per-class classification results using the precision and recall metrics. Precision reports the fraction of correctly classified samples, namely "condition positive", while recall reports the overall fraction of correctly classified instances of the "condition positive" class. An ideal classifier should achieve a precision and recall equal to one.

Fig. 4.5 provide scatter plots of the precision and recall scores for the YES and NO classes for the set of considered impact thresholds. We conduct the study of precision/recall scores for all mentioned classifiers and only present the results for the best performing ones: Decision Tree and Random Forest classifiers. We can observe that the two scores can be

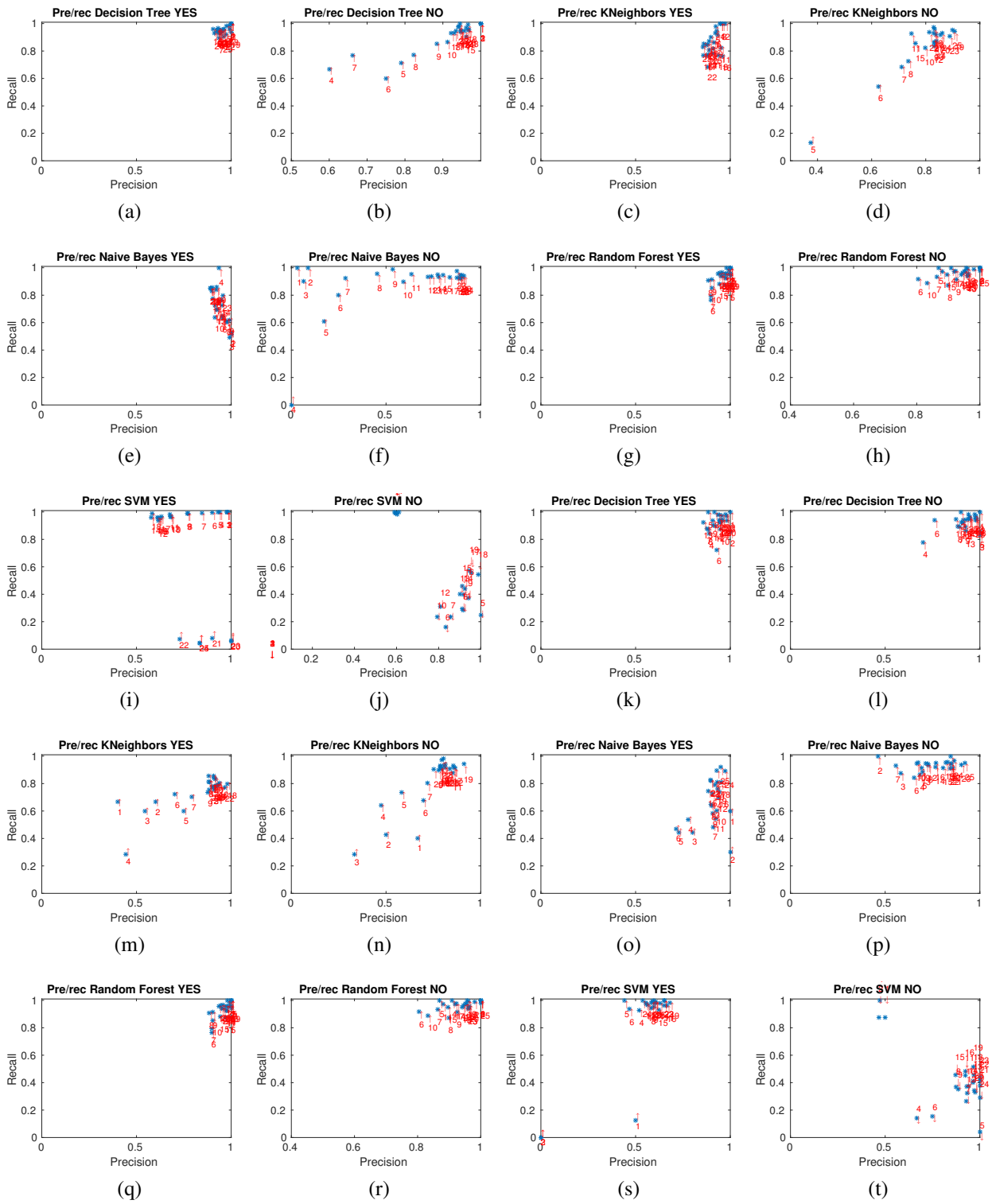


Fig. 4.5 Precision/recall for each class with respect to threshold (YES - impact is present, NO - impact is absent) and classifier: balanced dataset.

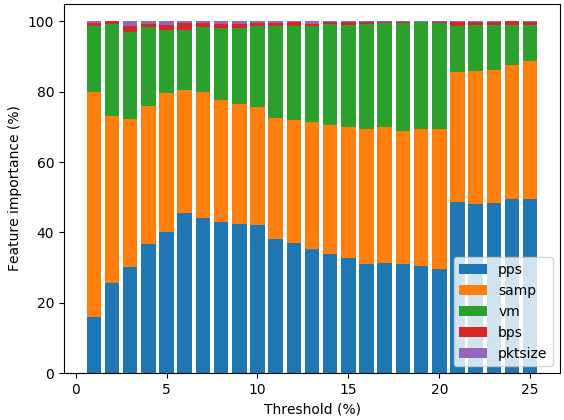
indeed close to one for both classes. We can also observe that the distribution of observations per class has little impact on the obtained scores for these two classifiers.

Feature importance

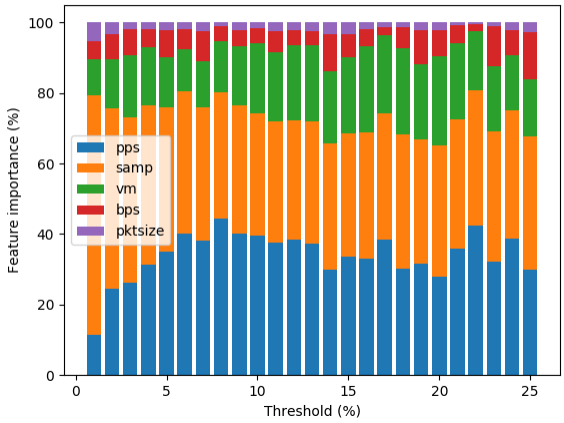
While trying to understand the conditions under which impact emerges, it is important to investigate to which extent every feature contributes to the classifier decision. By means of embedded method in Decision Tree and Random Forest regressors, we obtain feature importances reported by both classifiers represented in Fig. 4.6. We conclude from it that sampling rate and throughput in packets per second contribute the most to the classification decision. Indeed, as sampling rate is the most influencing parameter on the amount of traffic to be sampled by OvS, it boosts switch to compose samples, while processing ongoing traffic. In such manner, the interaction of monitoring and forwarding brings traffic losses. In addition, by the fact that sFlow is packet-based monitoring framework, classifiers confirm that the throughput in terms of packets per second matters over bits per second within investigated environment.

VM feature

Feature importance analysis reports a significant weight of VM feature in dataset, it takes the third place among the others. In our set-up network traffic is produced by 2, 4, 6, and 8 VMs. Every pair of VMs performs as source and destination. Therefore, comparing to single source and destination case (2 VMs), amount of traffic doubles for 4 VMs, triples for 6 VMs and quadruples for 8 VMs. Knowing that, we question such VM feature weight in classifiers decisions and choose to eliminate this feature from dataset, train our model again and check how this removing impacts classification accuracy. Fig. 4.7 portrays delta of classifiers predicting accuracies, when VM feature is present and removed from dataset. The order of difference is in range of several percents. It makes us conclude and confirm that VM feature, being related to traffic in packet per second feature, does not play major role in classification decisions and may not be considered as important during feature engineering.



(a)

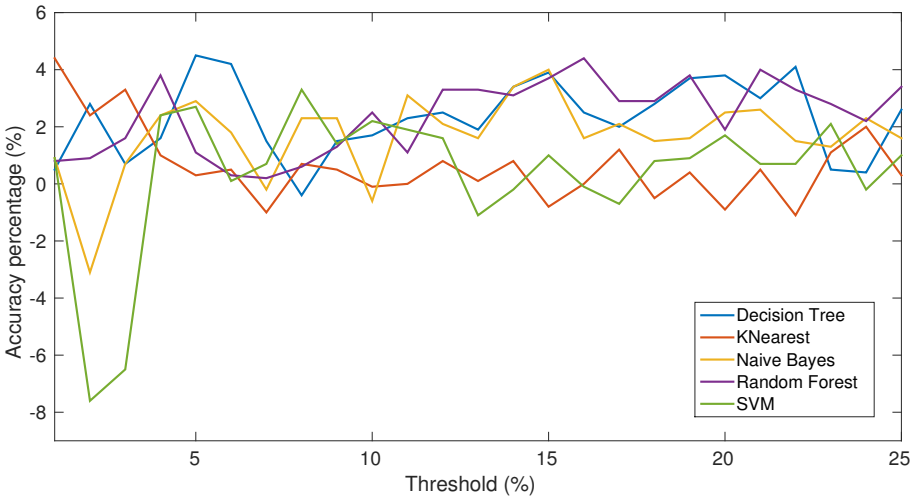


(b)

Fig. 4.6 Feature importance Decision Tree 4.6a Random Forest 4.6b

4.3.2 Quantifying the impact

It is important not only to identify the presence of impact, but also to predict its numerical value. In this section, we investigate this regression problem with the help of Decision Tree and Random Forest algorithms, as they appear to provide best predictions in the classification case. The validation shows values of mean absolute error (MAE) of 1.96% for Decision Trees and 1.81% for Random Forests, which stand for the average absolute difference between the real impact (in percent) and its predicted value (also in percent). We can thus conclude that offline regression models are able to accurately predict the impact value and can thus be used to tune the sampling rate so as to reach a desired impact level.



(a)

Fig. 4.7 Accuracy delta with VM feature eliminated from dataset

4.4 Online analysis

The previous section enabled us to assess the ability of machine learning to detect the presence of impact of sFlow on the application data plane and to estimate the amplitude of this impact. While the models constructed offline are efficient, there is no reason to believe they will be generic enough and applicable to all virtualized environments. To avoid generating a model per server offline, we study in this section whether these models can be produced online in an adaptive way. We explore this problem in this section and propose an approach to optimally set the sampling rate using the built model.

As algorithms belonging to the tree and forest families showed the best performance in the offline case, we rely on a particular decision tree algorithm called Hoeffding [157] to incrementally learn from instances coming one by one. Incremental Regression Hoeffding Tree is known for its efficiency in case of high-speed data streams. It is available in the multi-output streaming framework scikit-multiflow [158].

Building the model online requires the definition of a strategy for data acquisition to learn from. We will have to periodically collect several metrics, such as the traffic state when sampling is disabled, the throughput after sampling at some rate, the impact of this sampling rate on the throughput, and the number of VMs. Traffic state is available to measure at the switch ports. Sampling rate to test can be chosen randomly and configured on the switch

interface by the administrator. After this sampling rate is configured, one can measure again the traffic at the ports of the switch and calculate the drop in throughput in comparison with the no sampling case. The number of VMs can be estimated from the traffic collected by sFlow, at least for active VMs. All this procedure provides an instance of features, that can be used to update the machine learning algorithm. When done, we reset the sampling rate to a value that is judged appropriate, then wait until the next measurement epoch, where again we disable sFlow sampling, measure traffic, sample at some random sampling rate, remeasure traffic and number of VMs, and update the learner. We keep repeating this process until the model converges.

Traffic load in a virtual network is dynamic. Its variable nature has to be taken into account while building and validating our models and for optimally setting the sampling rate. To evaluate the performance of the online machine learning algorithm, we emulate the above procedure thanks to our offline dataset. We first split our initial set of 13,000 experiments into two sets: one used for learning (containing 80% of the instances) and one used for testing (the remaining 20%). At each measurement period, we provide to the learner a new measurement instance of the learning set and ask it to emit a prediction for all the experiments in the testing set.

We define two strategies for deciding the order in which experiments in the learning set are provided to the learner:

- A *global strategy* where the next experiment is chosen at random from the learning set. This mimics the case of a network with rapid traffic variations. Note that the sampling rate is chosen at random here.
- A *local strategy* where the next experiment is chosen at random in the neighborhood of the current experiment with respect to the number of VMs and traffic rate. For a given instance, the sampling rate to test is again chosen randomly. Contrary to the first strategy, this local strategy is supposed to mimic traffic with smooth fluctuations.

Following these strategies, we manage to transform our dataset into a synthetic network workload to be used for building and validating our models online. Within this synthetic workload, each instance corresponds to a scenario when virtual network is being under monitoring: at a given time there is a certain traffic condition on the server (some amount

of VMs sending traffic with some input rate) and certain measurement condition (some sampling rate is turned on).

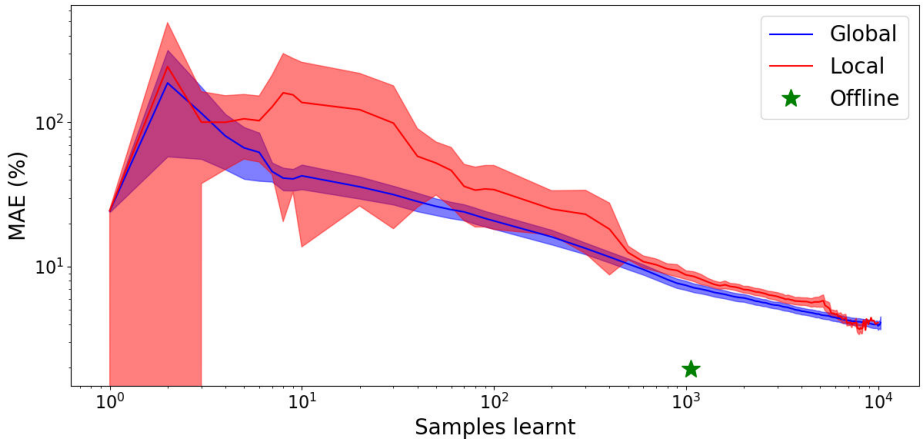


Fig. 4.8 MAE of regression for online global & local & offline

4.4.1 Predicting the drop in throughput

Fig. 4.8 presents mean absolute prediction errors for global random, local random and offline strategies as instances are being collected and learned from. The regression case is considered here where the objective is to predict the real value of the throughput reduction I (in percent). With the global random strategy, the learner initially provides predictions with high mean absolute errors and reaches 10% errors only after 1000 samples have been provided. The local random strategy leads to almost the same result at long run: about 1000 samples are needed to achieve less than 10% errors. However, the errors for the local random strategy are initially higher than the global one as, at the beginning, the model learns from a small set of similar traffic instances while it is asked to predict over diverse traffic samples, some of which it might have never seen before.

Comparison with the offline model suggests that the learning phase is going to be very long if one wants a error in the order of a few percents between the online and the offline approaches. An alternative, less complex but operationally meaningful approach is to ask the learner to provide a sampling rate such that the impact value stays below a predefined threshold. This is the question we investigate in the next section and check whether it can provide a faster convergence to the desired impact level.

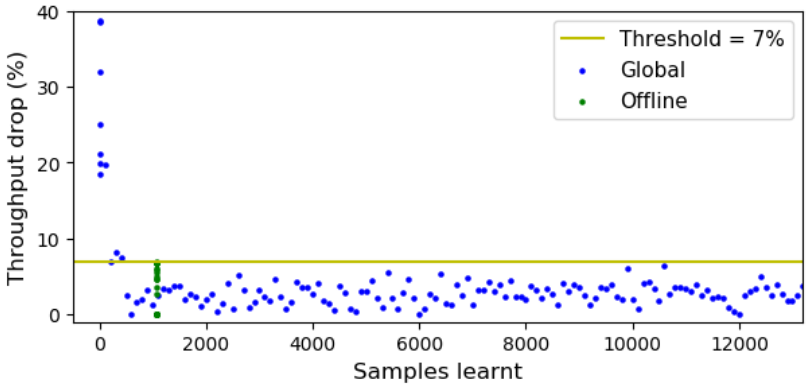
4.4.2 Finding optimal monitoring parameters

We consider the same online learning algorithm as in the previous section, using again the local and global synthetic traces. The procedure is as follows: once a new instance is selected to train the online algorithm, the learner is asked to predict the optimal sampling rate corresponding to this new instance. The optimal sampling rate is defined as the maximum sampling rate allowing the impact on the traffic to be within some desired threshold thr_I. The prediction is then compared to the exact value of the optimal sampling rate computed by considering all the instances in the complete dataset with the same traffic conditions while scanning all known sampling rates. The procedure is repeated until exhaustion of all 13,000 instances (there is no training and test sets here).

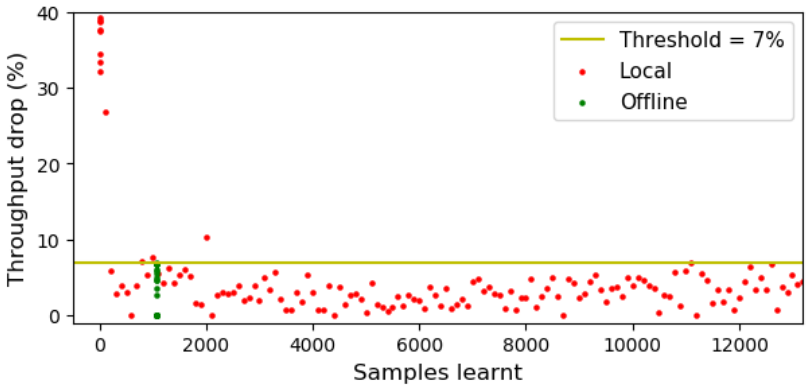
We present results using a threshold thr_I of 7%, as results are qualitatively similar for other thresholds. Fig. 4.9a and Fig. 4.9b report the results for the global and local synthetic workload respectively. The figures show the achieved impact when the optimal sampling rate is set (which should be less than, still close to, the desired impact thr_I). In comparison to the previous section (see Fig. 4.8) where the mean absolute error was improving at slow pace over time, we observe here a faster convergence of the learner. Only a few tens of tests are needed to train the online model for the task of setting optimally the sampling rate and limiting its impact on traffic to the desired value. This is because the prediction complexity of the task is lower given the finite number of sampling rates that can be implemented in sFlow (1 out of n packets). We can further observe that the online learner performs better under the local than the global workload. This is because under the local workload the learner is asked a prediction for traffic conditions close to the ones it got trained with.

4.5 Conclusion and Discussion

In this chapter, we considered the application of machine learning algorithms to predict the impact of capturing packets with a tunable sampling rate, in virtual environments. We further use these algorithms to determine the best performing measurement parameters. As each network features its own traffic profile, we considered the case where the training phase has to be done online. We also considered the offline case where the learner is fed at once with a large set of instances that correspond to a wide range of traffic and sampling conditions. The evaluations were performed with a large dataset of traces obtained in a controlled



(a)



(b)

Fig. 4.9 Throughput drop at estimated optimal sampling rate: global workload (4.9a) and local workload (4.9b).

environment. The results obtained suggest that our machine learning-based solution is able to predict the impact of virtual network monitoring, as well as optimal monitoring parameters w.r.t the impact of measuring the network.

Chapter 5

Conclusions and Future work

5.1 Conclusions

Advances in networking and computing technologies continue to transform the habits of network monitoring. On the one hand SDN, NFV and cloud paradigms simplify the network management, though at the same time inducing new challenges and promote development of new solutions for monitoring. In this thesis we attempt to contribute towards this subject.

In Chapter 3 we started with exploring the problem of virtual network monitoring. Upon experimentation with sFlow and IPFIX within a setup around a OvS switch, we have observed the relationship between monitoring process and network traffic. It is expressed in losses of traffic to a certain extent once monitoring is enabled in the setup. The fact that the monitoring process is implemented within OvS is executed within the host OS, i.e., directly on the hardware, drove us to investigate how the monitoring process interacts with the hardware aspect, because the hardware limitations could have explained the problems encountered with using sFlow. Analysing the system resources in terms of CPU and context switching, we confirmed that the host OS/hardware is not the one to blame for network degradation due to monitoring. Considering the obscurity of the observed interaction of monitoring and network, one of the opportunities to overcome it can be machine learning with its ability to reveal the relations between variables. With an objective to avoid a negative influence of monitoring onto network, we turned our research into this direction.

Thus in Chapter 4 we started to explore how to apply machine learning to our problem. At the beginning we collected the data for further manipulations. We took a closer look at the obtained dataset and analyse the relation between the components of virtual network, amount of sampling and observed traffic impact. This impact is expectedly proportional to sampling rate as well as to the number of virtual machines in the setup, which in their turn are proportional to amount of traffic. Having demonstrated that, we also conducted a feature analysis: which dataset component contributed the most to the decision of the model. The sampling and traffic features appeared the most important. We also verified that eliminating the VM feature (which is proportional to traffic) does not impact the model decision dramatically. We paid attention to the dataset balance, as it need to be accounted in ML models and could cause biased decisions.

In Chapter 4 we also identified the potential of machine learning modeling to detect the presence of impact (classification) and define its value (regression) in the setup with respect to different definitions of impact and obtained decent levels of performance. Obtained results confirmed the ability of machine learning to be efficiently applied to our problem and motivated us to improve the solution. It was clear, that as the obtained models were trained on synthetic data, they hardly could be generalized to other virtual setups. To overcome this limitation, we targeted online modelling. An online approach can build up a model on the fly. It does not require the whole dataset at once. Instead, it is able to update the model as new data appears. We proposed two online models with regard to network traffic dynamics, each signifying a different traffic scope: global or local. These strategies could recreate a network situation with rapid or smooth traffic fluctuations. Online models built on the basis of these strategies reported acceptable performance only at the late phases of learning: the absolute errors of predicted impact were higher comparatively to offline case.

The initial goal of this ML study was to be able to discover the necessary monitoring parameters for the setup, so as to avoid any interference with traffic. On the basis of online models, while the learning phase is in process, we demanded to provide a sampling rate for the current traffic profile such that a certain level of impact is not exceeded. Our study have shown that it is possible to provide optimal sampling rates at much earlier stages of learning phase, which in its turn indicates the applicability of machine learning in the context of measurement with sFlow in virtual network built around OvS.

Overall, in this thesis we show that network monitoring domain has high potential to benefit from machine learning. Considering recent research works and our obtained results,

the combination of these two areas can bring useful insights and stimulate new research ideas.

5.2 Future work

This thesis has demonstrated the potential of machine learning techniques to be used for calibration of network monitoring parameters, however few peculiarities remained uncovered.

Our model now requires tests in the wild, which could potentially bring up several issues; those can become further steps for future work. During their lifetime, networks usually experience repeated traffic variations. If the training phase entails a complex procedure, e.g., turning sampling on at different values in a regular manner, it can be interesting to collect data for training only when this is relevant. One could leverage active learning to make training phase more efficient, as active learning technique can decide when the learner needs additional training data to increase its accuracy.

Further, the model we obtained from the observation of the network would later suffer from aging. Model aging is a problem to be solved by periodical model retraining. One of the solution to avoid a complete retraining phase is to use algorithm like the CVFDT algorithm [159] that can keep a decision tree up to date over time.

Another issue that would be interesting to disclose is the computational overhead of the learning algorithms during learning phase. How the overhead would differ, for example, in the case of a single decision tree vs. a random forest of multiple trees or vs. other algorithms with different hyperparameters. Considering that in this work the overhead minimization is the main idea behind this modelling, revealing its computational overhead would be important as well, especially, for the targeted cloud environment which are so sensitive and competitive for computational resources.

Also, sFlow, being a packet-based sampling tool, provides results with quantifiable accuracy. If the suggested model will suggest very small sampling rate on the way to minimize impact, there may not be enough sampling data to provide accurate statistics on operating traffic. The trade-off between accuracy and impact avoidance could be taken into account. The requirements for accuracy could be introduced as input features when building and calibrating models.

References

[1] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[2] "Openstack: Build the future of open infrastructure." Available at https://www.openstack.org/.

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM CCR*, vol. 38, no. 2, 2008.

[4] "Opflex: An open policy protocol white paper." Available at https://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/white-paper-c11-731302.html year = 2014,.

[5] F. Moradi, C. Flinta, A. Johnsson, and C. Meirosu, "Conmon: An automated container based network performance monitoring system," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pp. 54–62, IEEE, 2017.

[6] P. Zhang, H. Li, C. Hu, L. Hu, L. Xiong, R. Wang, and Y. Zhang, "Mind the gap: Monitoring the control-data plane consistency in software defined networks," in *ACM CoNext*, 2016.

[7] J. Liu, Y. Li, H. Song, and D. Jin, "Netwatch: End-to-end network performance measurement as a service for cloud," *IEEE Transactions on Cloud Computing*, 2016.

[8] P. Sharma, S. Chatterjee, and D. Sharma, "Cloudview: Enabling tenants to monitor and control their cloud instantiations," in *IFIP/IEEE IM*, 2013.

[9] M. Ghasemi, T. Benson, and J. Rexford, "Rinc: Real-time inference-based network diagnosis in the cloud," *Under Submission*, 2015.

[10] Z. Su, T. Wang, Y. Xia, and M. Hamdi, "Cemon: A cost-effective flow monitoring system in software defined networks," *Computer Networks*, vol. 92, pp. 101–115, 2015.

[11] S. Panchen, P. Phaal, and N. McKee, "Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks," 2001.

[12] "Requirements for ip flow information export (ipfix)." Available at https://tools.ietf.org/html/rfc3917.

[13] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, “Inside the social network’s (datacenter) network,” *Computer Communication Review*, vol. 45, no. 5, 2015.

[14] A. Desai, R. Oza, P. Sharma, and B. Patel, “Hypervisor: A survey on concepts and taxonomy,” *International Journal of Innovative Technology and Exploring Engineering*, vol. 2, no. 3, pp. 222–225, 2013.

[15] KVM, “Documents — kvm,,” 2017. Available at https://www.linux-kvm.org/index.php?title=Documents&oldid=173827 = "[Online; accessed 18-July-2019]".

[16] “Vmware.” Available at https://www.vmware.com/.

[17] “"Xen Project".” Available at https://xenproject.org/.

[18] “Xenserver.” Available at https://xenserver.org/.

[19] “Hyper-v.” Available at https://en.wikipedia.org/wiki/Hyper-V.

[20] “Powervm.” Available at https://www.ibm.com/support/knowledgecenter/en/POWER9/p9eew/p9eew_intro.htm.

[21] “Vx32.” Available at https://pdos.csail.mit.edu/~baford/vm/.

[22] D. Merkel, “Docker: lightweight linux containers for consistent development and deployment,” *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[23] “"Linux containers".” Available at https://linuxcontainers.org/.

[24] “Openvz.” Available at https://openvz.org/.

[25] “"Creating and controlling jails".” Available at https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/jails-build.html.

[26] R. Morabito, J. Kjällman, and M. Komu, “Hypervisors vs. lightweight virtualization: a performance comparison,” in *2015 IEEE International Conference on Cloud Engineering*, pp. 386–393, IEEE, 2015.

[27] J. Hwang, S. Zeng, F. y Wu, and T. Wood, “A component-based performance comparison of four hypervisors,” in *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pp. 269–276, IEEE, 2013.

[28] “"Apache CloudStack:Open Source Cloud Computing".” Available at https://cloudstack.apache.org/.

[29] “"Red Hat Virtualization".” Available at https://access.redhat.com/products/red-hat-virtualization.

[30] “"oVirt is a free open-source virtualization solution for your entire enterprise".” Available at https://www.ovirt.org/.

[31] S. Clayman, A. Galis, and L. Mamatas, “Monitoring virtual networks with lattice,” in *2010 IEEE/IFIP Network Operations and Management Symposium Workshops*, pp. 239–246, IEEE, 2010.

[32] "Open vSwitch - Production Quality, Multilayer Open Virtual Switch." Available at https://www.openvswitch.org/.

[33] M. Paolino, N. Nikolaev, J. Fanguede, and D. Raho, "Snabbswitch user space virtual switch benchmark and performance optimization for nfv," in *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, pp. 86–92, IEEE, 2015.

[34] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "Softnic: A software nic to augment hardware," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155*, 2015.

[35] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, "Pisces: A programmable, protocol-independent software switch," in *Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 525–538, ACM, 2016.

[36] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pp. 5–16, IEEE, 2015.

[37] "Vpp - vector packet processing platform." Available at https://wiki.fd.io/view/VPP.

[38] L. Rizzo and G. Lettieri, "Vale, a switched ethernet for virtual machines," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pp. 61–72, ACM, 2012.

[39] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, *et al.*, "The design and implementation of open vswitch.," in *Usenix NSDI*, 2015.

[40] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015.

[41] I. Cerrato, M. Annarumma, and F. Risso, "Supporting fine-grained network functions through intel dpdk.," in *EWSDN*, pp. 1–6, 2014.

[42] M.-A. Kourtis, G. Xilouris, V. Riccobene, M. J. McGrath, G. Petralia, H. Koumaras, G. Gardikis, and F. Liberal, "Enhancing vnf performance by exploiting sr-iov and dpdk packet processing acceleration," in *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, pp. 74–78, IEEE, 2015.

[43] T. Zhang, L. Linguaglossa, J. Roberts, L. Iannone, M. Gallo, and P. Giaccone, "A benchmarking methodology for evaluating software switch performance for nfv," in *IEEE Conference on Network Softwarization (NetSoft'19)*, 2019.

[44] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "Performance characteristics of virtual switching," in *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, pp. 120–125, IEEE, 2014.

[45] G. A. Gallardo, B. Baynat, and T. Begin, "Performance modeling of virtual switching systems," in *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 125–134, IEEE, 2016.

[46] P.-W. Tsai, C.-W. Tsai, C.-W. Hsu, and C.-S. Yang, "Network monitoring in software-defined networking: A review," *IEEE Systems Journal*, no. 99, 2018.

[47] P. Perešíni, M. Kuźniar, and D. Kostić, "Monocle: Dynamic, fine-grained data plane monitoring," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, ACM, 2015.

[48] A. Beloglazov and R. Buyya, "Energy efficient allocation of virtual machines in cloud data centers," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pp. 577–578, IEEE, 2010.

[49] J. T. Piao and J. Yan, "A network-aware virtual machine placement and migration approach in cloud computing," in *2010 Ninth International Conference on Grid and Cloud Computing*, pp. 87–92, IEEE, 2010.

[50] C. Dupont, T. Schulze, G. Giuliani, A. Somov, and F. Hermenier, "An energy aware framework for virtual machine placement in cloud federated data centres," in *2012 Third International Conference on Future Systems: Where Energy, Computing and Communication Meet (e-Energy)*, pp. 1–10, IEEE, 2012.

[51] X. Ruan, H. Chen, Y. Tian, and S. Yin, "Virtual machine allocation and migration based on performance-to-power ratio in energy-efficient clouds," *Future Generation Computer Systems*, vol. 100, pp. 380–394, 2019.

[52] A. Segalini, D. L. Pacheco, Q. Jacquemart, M. Rifai, G. Urvoy-Keller, and M. Dione, "Towards massive consolidation in data centers with seamless," in *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 233–242, IEEE Press, 2018.

[53] W. Wu, K. He, and A. Akella, "Perfsight: Performance diagnosis for software data-planes," in *ACM IMC*, 2015.

[54] C. Zeng, F. Liu, S. Chen, W. Jiang, and M. Li, "Demystifying the performance interference of co-located virtual network functions," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pp. 765–773, IEEE, 2018.

[55] T. Zhang, M. Chen, X. Wei, B. Chen, and C. Hu, "Sdnms: A software defined network measurement system for nfv networks," *China Communications*, vol. 16, no. 4, pp. 59–74, 2019.

[56] J. M. A. Calero and J. G. Aguado, "Monpaas: an adaptive monitoring platformas a service for cloud computing infrastructures and services," *IEEE Transactions on Services Computing*, vol. 8, no. 1, pp. 65–78, 2014.

[57] D. Sharma, R. Poddar, K. Mahajan, M. Dhawan, and V. Mann, "H ansel: diagnosing faults in openstack," in *ACM CoNext*, 2015.

[58] A. Goel, S. Kalra, and M. Dhawan, "Gretel: Lightweight fault localization for openstack," in *ACM CoNext*, 2016.

[59] H. Baek, A. Srivastava, and J. Van der Merwe, "Cloudsight: A tenant-oriented transparency framework for cross-layer cloud troubleshooting," in *IEEE/ACM CCGRID*, 2017.

[60] A. Shatnawi, M. Orrù, M. Mobilio, O. Riganelli, and L. Mariani, "Cloudhealth: a model-driven approach to watch the health of cloud services," in *IEEE/ACM 1st international workshop on Software Health (SoHeal)*, 2018.

[61] W. Wu, G. Wang, A. Akella, and A. Shaikh, "Virtual network diagnosis as a service," in *ACM SoCC*, p. 9, 2013.

[62] M. Ghasemi, T. Benson, and J. Rexford, "Dapper: Data plane performance diagnosis of tcp," in *ACM SOSR*, 2017.

[63] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about sdn flow tables," in *International Conference on Passive and Active Network Measurement*, pp. 347–359, Springer, 2015.

[64] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pp. 113–126, 2012.

[65] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pp. 99–111, 2013.

[66] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pp. 15–27, 2013.

[67] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat, "Libra: Divide and conquer to verify forwarding tables in huge networks," in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pp. 87–99, 2014.

[68] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," in *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pp. 1–13, 2013.

[69] T. Wang, F. Liu, J. Guo, and H. Xu, "Dynamic sdn controller assignment in data center networks: Stable matching with transfers," in *IEEE INFOCOM*, 2016.

[70] K. Benzekki, A. El Fergougui, and A. Elbelrhiti Elalaoui, "Software-defined networking (sdn): a survey," *Security and communication networks*, vol. 9, no. 18, pp. 5803–5833, 2016.

[71] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 351–362, 2011.

[72] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, pp. 254–265, ACM, 2011.

[73] J. C. Mogul and P. Congdon, "Hey, you darned counters!: get off my asic!," in *Proceedings of the first workshop on Hot topics in software defined networks*, pp. 25–30, ACM, 2012.

[74] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, *et al.*, "Onix: A distributed control platform for large-scale production networks.," in *OSDI*, vol. 10, pp. 1–6, 2010.

[75] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: a framework for efficient and scalable offloading of control applications," in *Proceedings of the first workshop on Hot topics in software defined networks*, pp. 19–24, ACM, 2012.

[76] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, vol. 3, 2010.

[77] "Maestro platform." Available at http://code.google.com/p/maestro-platform/.

[78] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pp. 71–85, 2014.

[79] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, *et al.*, "Packet-level telemetry in large datacenter networks," in *ACM CCR*, vol. 45, 2015.

[80] P. Tammana, R. Agarwal, and M. Lee, "Simplifying datacenter network debugging with pathdump.," in *Usenix OSDI*, 2016.

[81] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "Snap: Stateful network-wide abstractions for packet processing," in *Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 29–43, ACM, 2016.

[82] O. Tilmans, T. Bühler, I. Poese, S. Vissicchio, and L. Vanbever, "Stroboscope: Declarative network monitoring on a budget," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pp. 467–482, 2018.

[83] P. Emmerich, M. Pudelko, S. Gallenmüller, and G. Carle, "Flowscope: Efficient packet capture and storage in 100 gbit/s networks," in *2017 IFIP Networking Conference (IFIP Networking) and Workshops*, pp. 1–9, IEEE, 2017.

[84] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *ACM SIGCOMM Computer Communication Review*, vol. 41, pp. 290–301, ACM, 2011.

[85] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker, "Compiling path queries," in *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pp. 207–222, 2016.

[86] "Sampled netflow." Available at https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/fnetflow/configuration/15-mt/fnf-15-mt-book/use-fnflow-redce-cpu.html.

[87] N. G. Duffield and M. Grossglauser, "Trajectory sampling for direct traffic observation," *IEEE/ACM Transactions on Networking (ToN)*, vol. 9, no. 3, pp. 280–292, 2001.

[88] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers," in *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pp. 311–324, 2016.

[89] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, "Opensample: A low-latency, sampling-based measurement platform for commodity sdn," in *IEEE ICDCS*, 2014.

[90] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pp. 29–42, 2013.

[91] D. R. Teixeira, J. M. C. Silva, and S. R. Lima, "Deploying time-based sampling techniques in software-defined networking," in *IEEE SoftCOM*, 2018.

[92] S. Shirali-Shahreza and Y. Ganjali, "Flexam: flexible sampling extension for monitoring and security applications in openflow," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 167–168, ACM, 2013.

[93] "sflow® is an industry standard technology for monitoring high speed switched networks.." Available at https://sflow.org/.

[94] M. Flittner and R. Bauer, "Trex: Tenant-driven network traffic extraction for sdn-based cloud environments," in *2017 Fourth International Conference on Software Defined Systems (SDS)*, pp. 48–53, IEEE, 2017.

[95] A. Wang, Y. Guo, F. Hao, T. Lakshman, and S. Chen, "Umon: Flexible and fine grained traffic monitoring in open vswitch," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, p. 15, ACM, 2015.

[96] H. Chen, N. Foster, J. Silverman, M. Whittaker, B. Zhang, and R. Zhang, "Felix: Implementing traffic measurement on end hosts using program analysis," in *Proceedings of the Symposium on SDN Research*, p. 14, ACM, 2016.

[97] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: a stream database for network applications," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pp. 647–651, ACM, 2003.

[98] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *ACM Sigplan Notices*, vol. 46, no. 9, pp. 279–291, 2011.

[99] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," *Acm sigplan notices*, vol. 49, no. 1, pp. 113–126, 2014.

[100] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: Simplifying sdn programming using algorithmic policies," in *ACM SIGCOMM Computer Communication Review*, vol. 43, pp. 87–98, ACM, 2013.

[101] L. Yuan, C.-N. Chuah, and P. Mohapatra, "Progme: towards programmable network measurement," *IEEE/ACM Transactions on Networking*, vol. 19, no. 1, pp. 115–128, 2010.

[102] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Dream: dynamic resource allocation for software-defined measurement," *ACM SIGCOMM CCR*, vol. 44, no. 4, pp. 419–430, 2015.

[103] X. T. Phan and K. Fukuda, "Sdn-mon: Fine-grained traffic monitoring framework in software-defined networks," *Journal of Information Processing*, vol. 25, pp. 182–190, 2017.

[104] K. C. Claffy, H.-W. Braun, and G. C. Polyzos, "A parameterizable methodology for internet traffic flow profiling," *IEEE Journal on selected areas in communications*, vol. 13, no. 8, pp. 1481–1494, 1995.

[105] F. P. Tso and D. P. Pezaros, "Baatdaat: Measurement-based flow scheduling for cloud data centers," in *2013 IEEE Symposium on Computers and Communications (ISCC)*, pp. 000765–000770, IEEE, 2013.

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

[107] L. Jose, M. Yu, and J. Rexford, "Online measurement of large traffic aggregates on commodity switches.," in *Hot-ICE*, 2011.

[108] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, "Opentm: traffic matrix estimator for openflow networks," in *International Conference on Passive and Active Network Measurement*, pp. 201–210, Springer, 2010.

[109] N. L. Van Adrichem, C. Doerr, and F. A. Kuipers, "Opennetmon: Network monitoring in openflow software-defined networks," in *2014 IEEE Network Operations and Management Symposium (NOMS)*, pp. 1–8, IEEE, 2014.

[110] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "Payless: A low cost network monitoring framework for software defined networks," in *IEEE NOMS*, 2014.

[111] Y. Zhang, "An adaptive flow counting method for anomaly detection in sdn," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pp. 25–30, ACM, 2013.

[112] "Juniper flow monitoring." Available at https://www.juniper.net/us/en/local/pdf/app-notes/3500204-en.pdf year = 2011,.

[113] "Cflowd." Available at https://documentation.nokia.com/html/0_add-h-f/93-0073-10-01/7750_SR_OS_Router_Configuration_Guide/Cflowd-Intro.html.

[114] B.-Y. Choi and S. Bhattacharyya, "On the accuracy and overhead of cisco sampled netflow," in *Proceedings of ACM SIGMETRICS Workshop on Large Scale Network Inference (LSNI)*, pp. 1–6, 2005.

[115] V. Mann, A. Vishnoi, and S. Bidkar, "Living on the edge: Monitoring network flows at the edge in cloud data centers," in *IEEE COMSNETS*, 2013.

[116] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo, "A comprehensive survey on machine learning for networking: evolution, applications and research opportunities," *Journal of Internet Services and Applications*, vol. 9, no. 1, p. 16, 2018.

[117] J. Xie, F. R. Yu, T. Huang, R. Xie, J. Liu, C. Wang, and Y. Liu, "A survey of machine learning techniques applied to software defined networking (sdn): Research issues and challenges," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 393–430, 2018.

[118] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE transactions on information theory*, vol. 13, no. 1, pp. 21–27, 1967.

[119] L. Breiman, *Classification and regression trees*. Routledge, 2017.

[120] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[121] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[122] G. E. Box and G. C. Tiao, *Bayesian inference in statistical analysis*, vol. 40. John Wiley & Sons, 2011.

[123] T. D. Nielsen and F. V. Jensen, *Bayesian networks and decision graphs*. Springer Science & Business Media, 2009.

[124] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian network classifiers," *Machine learning*, vol. 29, no. 2-3, pp. 131–163, 1997.

[125] P. Velan, "The future of network flow monitoring," 2019.

[126] Y. Li and J. Li, "Multiclassifier: A combination of dpi and ml for application-layer classification in sdn," in *The 2014 2nd International Conference on Systems and Informatics (ICSAI 2014)*, pp. 682–686, IEEE, 2014.

[127] D. Rossi and S. Valenti, "Fine-grained traffic classification with netflow data," in *Proceedings of the 6th international wireless communications and mobile computing conference*, pp. 479–483, ACM, 2010.

[128] B. Li, M. H. Gunes, G. Bebis, and J. Springer, "A supervised machine learning approach to classify host roles on line using sflow," in *Workshop on High performance and programmable networking*, 2013.

[129] A. Nakao and P. Du, "Toward in-network deep machine learning for identifying mobile applications and enabling application specific network slicing," *IEICE Transactions on Communications*, p. 2017CQI0002, 2018.

[130] E. Liang, H. Zhu, X. Jin, and I. Stoica, "Neural packet classification," *arXiv preprint arXiv:1902.10319*, 2019.

[131] M. J. Khokhar, T. Spetebroot, and C. Barakat, "An online sampling approach for controlled experimentation and qoe modeling," in *2018 IEEE International Conference on Communications (ICC)*, pp. 1–6, IEEE, 2018.

[132] S. Jain, M. Khandelwal, A. Katkar, and J. Nygate, "Applying big data technologies to manage qos in an sdn," in *2016 12th International Conference on Network and Service Management (CNSM)*, pp. 302–306, IEEE, 2016.

[133] R. Pasquini and R. Stadler, "Learning end-to-end application qos from openflow switch statistics," in *2017 IEEE Conference on Network Softwarization (NetSoft)*, pp. 1–9, IEEE, 2017.

[134] P. Wang, S.-C. Lin, and M. Luo, "A framework for qos-aware traffic classification using semi-supervised machine learning in sdns," in *2016 IEEE International Conference on Services Computing (SCC)*, pp. 760–765, IEEE, 2016.

[135] M. Glick and H. Rastegarfar, "Scheduling and control in hybrid data centers," in *2017 IEEE Photonics Society Summer Topical Meeting Series (SUM)*, pp. 115–116, IEEE, 2017.

[136] P. Xiao, W. Qu, H. Qi, Y. Xu, and Z. Li, "An efficient elephant flow detection with cost-sensitive in sdn," in *2015 1st International Conference on Industrial Networks and Intelligent Systems (INISCom)*, pp. 24–28, IEEE, 2015.

[137] T. Hurley, J. E. Perdomo, and A. Perez-Pons, "Hmm-based intrusion detection system for software defined networking," in *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 617–621, IEEE, 2016.

[138] A. S. da Silva, J. A. Wickboldt, L. Z. Granville, and A. Schaeffer-Filho, "Atlantic: A framework for anomaly traffic detection, classification, and mitigation in sdn," in *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*, pp. 27–35, IEEE, 2016.

[139] S. Nanda, F. Zafari, C. DeCusatis, E. Wedaa, and B. Yang, "Predicting network attack patterns in sdn using machine learning approach," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pp. 167–172, IEEE, 2016.

[140] T. A. Tang, L. Mhamdi, D. McLernon, S. A. R. Zaidi, and M. Ghogho, "Deep learning approach for network intrusion detection in software defined networking," in *2016 International Conference on Wireless Networks and Mobile Communications (WINCOM)*, pp. 258–263, IEEE, 2016.

[141] P. Wang, K.-M. Chao, H.-C. Lin, W.-H. Lin, and C.-C. Lo, "An efficient flow control approach for sdn-based network threat detection and migration using support vector machine," in *2016 IEEE 13th International Conference on e-Business Engineering (ICEBE)*, pp. 56–63, IEEE, 2016.

[142] N. Shone, T. N. Ngoc, V. D. Phai, and Q. Shi, "A deep learning approach to network intrusion detection," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 2, no. 1, pp. 41–50, 2018.

[143] R. Braga, E. de Souza Mota, and A. Passito, "Lightweight ddos flooding attack detection using nox/openflow.," in *LCN*, vol. 10, pp. 408–415, 2010.

[144] L. Barki, A. Shidling, N. Meti, D. Narayan, and M. M. Mulla, "Detection of distributed denial of service attacks in software defined networks," in *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 2576–2581, IEEE, 2016.

[145] C. Li, Y. Wu, X. Yuan, Z. Sun, W. Wang, X. Li, and L. Gong, "Detection and defense of ddos attack–based on deep learning in openflow-based sdn," *International Journal of Communication Systems*, vol. 31, no. 5, p. e3497, 2018.

[146] Q. Niyaz, W. Sun, and A. Y. Javaid, "A deep learning based ddos detection system in software-defined networking (sdn)," *arXiv preprint arXiv:1611.07400*, 2016.

[147] L. J. Jagadeesan and V. Mendiratta, "Programming the network: Application software faults in software-defined networks," in *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 125–131, IEEE, 2016.

[148] Z. Din and J. de Oliveira, "Anomaly free on demand stateful software defined firewalling," in *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pp. 1–9, IEEE, 2017.

[149] A. Gulenko, M. Wallschläger, F. Schmidt, O. Kao, and F. Liu, "Evaluating machine learning algorithms for anomaly detection in clouds," in *IEEE Big Data*, 2016.

[150] C. Sauvanaud, K. Lazri, M. Kaâniche, and K. Kanoun, "Towards black-box anomaly detection in virtual network functions," in *IEEE/IFIP DSN-W*, 2016.

[151] Z. M. Fadlullah, F. Tang, B. Mao, N. Kato, O. Akashi, T. Inoue, and K. Mizutani, "State-of-the-art deep learning: Evolving machine intelligence toward tomorrow's intelligent network traffic control systems," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, 2017.

[152] A. Mestres, A. Rodriguez-Natal, J. Carner, P. Barlet-Ros, Alarcón, *et al.*, "Knowledge-defined networking," *ACM SIGCOMM Computer Communication Review*, 2017.

[153] P. Aitken, B. Claise, and B. Trammell, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information." RFC 7011, 2013.

[154] A. Zimmermann, A. Hannemann, and T. Kosse, "Flowgrind-a new performance measurement tool," in *IEEE GLOBECOM*, 2010.

[155] "iPerf - The ultimate speed test tool for TCP, UDP and SCTP." Available at https://iperf.fr/.

[156] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, Thirion, *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, 2011.

[157] P. Domingos and G. Hulten, "Mining high-speed data streams," in *KDD*, vol. 2, 2000.

[158] J. Montiel, J. Read, A. Bifet, and T. Abdessalem, "Scikit-multiflow: a multi-output streaming framework," *The Journal of Machine Learning Research*, vol. 19, no. 1, 2018.

[159] G. Hulten, L. Spencer, and P. Domingos, "Mining time-changing data streams," in *ACM SIGKDD international conference on Knowledge Discovery and Data Mining*, ACM, 2001.