Cobos, A., Guimaraes, C., de la Oliva, A. & Zabala, A. (9 november, 2021). *OpenFlowMon: A Fully Distributed Monitoring Framework for Virtualized Environments* [proceedings]. 2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), Heraklion, Greece.

DOI: 10.1109/nfv-sdn53031.2021.9665014

OpenFlowMon: A Fully Distributed Monitoring Framework for Virtualized Environments

Antonio Cobos*, Carlos Guimarães†, Antonio de la Oliva†, Aitor Zabala*,
*Telcaria Ideas S.L., Spain
†Universidad Carlos III de Madrid, Spain
Email: antonio.cobos@telcaria.com, cmagalha@pa.uc3m.es, aoliva@it.uc3m.es, aitor.zabala@telcaria.com

***Abstract*—Network monitoring allows a continuous assessment on the health and performance of the network infrastructure. With the significant change on how networks are deployed and operated, mainly due to the advent of virtualization technologies, alternative monitoring approaches are emerging to provide a finer-grained flow monitoring to complement already existing mechanisms and capabilities.**

In this paper, we proposed and developed an Open-Source Flow Monitoring Framework (OpenFlowMon), a fully distributed monitoring framework implemented solely with open-source solutions. This framework is used to assess the performance and the overhead introduced by two different flow monitoring approaches: *(i)* switch level and *(ii)* compute node level monitoring. Results show that monitoring at compute node level not only reduces the overhead but also mitigates a potential complex post-processing in east-to-west traffic.

I. Introduction

Computer networks are playing a key role in today's society by interconnecting people, services and all types of computer devices. Thus, network monitoring is a critical process to ensure that it performs optimally and that failures can be predicted or quickly identified, such as performance issues, bottlenecks, and security breaches. Different network components (e.g., processes, switches, routers, access points, middleboxes, etc) at distinct network segments are continuously monitored, empowering network managers with information such as what is being transmitted in the network, how it is working, and what it is happening. In doing so, network managers can correctly apply network reconfigurations to maintain and optimize the performance and overall availability of the network, as well as, overcoming failures and security breaches.

Given the importance of the networking asset to several businesses, large companies bear large costs in proprietary and dedicated monitoring hardware, as they own great part of the network infrastructure that affects their businesses. However, these are sometimes not in the reach of smaller companies since they just do not own the infrastructure to enforce monitoring by themselves. In turn, they have to rely on infrastructure owner and on the exposed interfaces in order to increase their profit margins and maintain the cost of their products, thus not losing leverage against larger companies.

However, the application of virtualization technologies into an ever-increasing set of services is changing the overall stakeholders' roles. Moreover, it is paving the way for new network monitoring approaches that shift from traditional availability monitoring. Altogether, accurate but still efficient monitoring can reduce the management traffic transmitted across the network, releasing bandwidth for the real data plane traffic, while making the network more resilient to downtime or failures. It is not only possible to have traditional monitoring at **switch level**, namely at switches, routers and middleboxes (purple boxes at Fig. 1), but also to have monitoring at the end node, namely at **compute node level** (red boxes at Fig. 1).

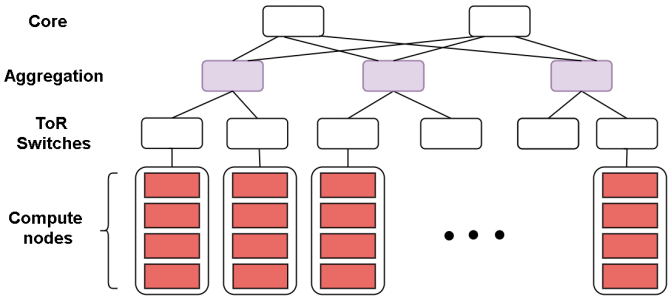


Fig. 1: Hierarchy Tier 3 Data Center network topology

This is where resides the main contribution of this paper. An open-source monitoring framework, dubbed OpenFlowMon, is proposed as a fully distributed monitoring framework that relies solely on open-source solutions for its implementation. Such framework supports two monitoring approaches: *(i)* at the switch level; and *(ii)* at the compute node level. These are then experimentally evaluated in order to assess the advantages and drawbacks of each approach. Such evaluation can contribute to small and medium-sized enterprises (SMEs), universities, laboratories, data centers, etc. to measure several Key Performance Indicators (KPIs) effectively, with great accuracy, at lower costs, and without requiring to own the entire network infrastructure.

The remainder of this paper is organized as follows. Section II surveys the components of a monitoring tool, identifying available open-source solutions. Section III depicts the implemented open-source monitoring framework (i.e., OpenFlowMon), which is being leveraged for the experimental analysis in Section IV. Finally, Section V concludes this work, identifying the main applications and exploitation of each network monitoring approach.

II. Flow Monitoring Components

For several years, network monitoring has been a focus for research, mostly targeting the overall health of a network infrastructure and the detection of problems. As a result, Simple Network Management Protocol (SNMP) [1] become one of the *de-facto* network monitoring protocols, providing a common mechanism for network devices (such as, routers, switches, servers, firewalls, wireless access points, etc) to relay management information to a centralized entity (i.e., SNMP manager).

However, the network infrastructures have increased in size and complexity, far beyond the infrastructure availability capabilities provided by SNMP. Flow monitoring appears as a new category of network monitoring that provides a finer-grained understanding on the traffic structure traversing the network. Its components are depicted in Figure 2 and are categorized as follows:

1) **Flow exporter:** aggregates packets into flows and exports the corresponding *flow data* towards one or more *flow collectors*.
2) **Flow information export protocol:** transports the flow information between a *flow exporter* and a *flow collector*.
3) **Flow collector:** receives, stores, and pre-processes *flow data* received from a *flow exporter*.
4) **Analysis applications:** analyze received *flow data* in the context of a certain application. For example, intrusion detection or traffic profiling applications.

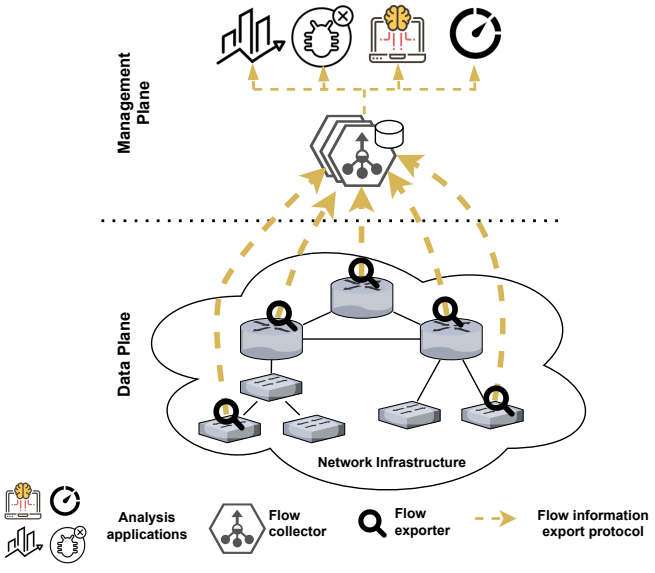


Fig. 2: Flow Monitoring Components

A. Flow Exporter

A *flow exporter* aggregates packets into flows and exports the corresponding *flow data* towards one or more *flow collectors*. It can be categorized with respect to where it is running: *(i)* for switch monitoring; or *(ii)* for compute nodes monitoring.

1) Flow Exporter for Switch Monitoring: Typically, commercial hardware switches provide their own proprietary exporters, making use of data interfaces only available for their manufacturer solution. As such, the usage of *generic flow exporters* is limited, since the data plane cannot be directly monitored, being only possible to get some aggregated metrics provided by the management interfaces of the switch. Therefore, a common approach to fine-grain monitoring is to perform port mirroring. This is the most extended and standard way to export traffic that passes through a switch. In doing so, all network packets (possibly filtered by some criteria) passing through a given switch port are copied and sent to a network monitoring application running somewhere in the network. This workaround can be used for most of the switches, being simple to set up and easy to use. However, since this mechanism relies on the replication of packets, it not only introduces additional overhead in the network and in the switch itself but also might produce increased congestion on the network and increased jitter of packets. Consequently, it affects the measurement accuracy [2].

On the other side, hardware switches based on open solutions (such as, OpenWrt or DD-WRT) or software switches (such as, *Open vSwitch* (OvS)) enable higher flexibility of exporters. In particular, *OvS* is a multi-layer virtual switch, designed to enable massive network automation through programmatic extension, while supporting standard management interfaces and protocols (e.g., *NetFlow*, *sFlow*, *IPFIX*, *RSPAN*, *CLI*, *LACP*, *802.1ag*). Such solutions open space for more flexible and powerful *flow exporters*, allowing different metrics to be defined and exported.

Although *flow exporters* for switch monitoring is complex to be achieved, they provide inherent advantages such as a fast configuration and more fine-grained monitoring.

2) Flow Exporter for Compute Nodes Monitoring: In the same way aggregated data can be obtained from the switches between the flows' endpoints, monitoring metrics can also be gathered at the endpoint itself, namely on the computing node where the endpoint is running. Multiple exporters based on *pcap* (an API for capturing network traffic) are baseline of several solutions. Morever, as *OvS* can operate as a soft switch running within the hypervisor, and even operate as the control stack for switching silicon [3], OvS-compliant exporters can also be applied into compute nodes. The following exporters are already deployed in real scenarios:

- **nProbe:** captures network traffic and efficiently aggregates it into flows. The program shows a limited memory footprint (less than 2 MB of memory regardless of the network size) and low CPU impact, enabling it to run on constraint platforms.
- **pmacct:** gathers and aggregates IP traffic. Its operation is based on the storage of local data in an in-memory table which content can be retrieved by a client program via a remote connection. It is compatible with the Berkeley Packet Filters (BPF), thus compatible with a common language used by several network tools, including protocol analyzers and packet sniffers, network monitors,

network intrusion detection systems, traffic-generators, and network-testers.

Although exporting monitoring metrics at the compute nodes requires more extensive configuration, it is easy to be adapted to different usages and metrics. Moreover, this solution allows filtering the flows before sending them to the *flow collector*.

B. Flow Information Export Protocol

Flow exporters and *flow collectors* exchange *flow data* information through a *flow information export protocol*. Several protocols have been implemented in the recent years, being presented the most commonly deployed below.

1) ***Netflow [4]****: Netflow*, developed by Cisco Systems to collect information about IP traffic, has become an industry standard for network traffic monitoring and it is currently supported by various platforms. In addition to Cisco IOS and NXOS, it is also supported by network devices from other manufacturers such as Juniper and Enterasys Switches, and operating systems such as Linux, FreeBSD, NetBSD and OpenBSD. It works in a push mode, sending *flow data* from a cache to a *flow collector*. Finally, *NetFlow* protocol is encapsulated in either SCTP or UDP protocols at the transport layer. While SCTP protocol is used when congestion awareness and reliability mechanisms are required, whenever such mechanisms are not required UDP can be used instead in a best-effort fashion.

2) ***IPFIX [5]****:* Internet Protocol Flow Information Export (*IPFIX*) is defined in RFC 5153 [6]. The *IPFIX* protocol defines how IP flow information is formatted and transferred from a *flow exporter* to a *flow collector*. It defines a *flow data* as any number of packets sharing a number of properties and observed in a specific time slot. Cisco *NetFlow* Version 9 was the basis for *IPFIX*. The difference between them is that the actual format of flows in *IPFIX* messages is to a great extent up to the sender. *IPFIX* introduces the makeup of these messages to the receiver with the help of special *templates*. The sender is also free to use user-defined data types in its messages, so the protocol is freely extensible and can adapt to different scenarios.

It implements a push-based communication model, being encapsulated in SCTP, TCP or UDP. Nevertheless, it defines SCTP as the preferable transport protocol.

3) ***sFlow [7]****: sFlow*, short for "sampled flow" and currently in version 5, is an industry-standard for packet export at Layer 2 (i.e., Data Link Layer). It exports truncated packets and interface counters, which are defined as a flow samples and counter samples, respectively. *sFlow* provides two types of sampling: *(i)* random sampling of packets; and *(ii)* time-based sampling of counters. In both cases, *flow data* and counter samples are sent over UDP by the *flow exporter* to the *flow collector*. Although it does not significantly affect the accuracy of the measurements, it affects the fidelity of the data, since no reliability or acknowledgments are enforced within the protocol.

C. Flow Collectors

The *flow collector* is responsible for the reception, storage, and pre-processing of *flow data* issued by the *flow exporters*. A summary of the main characteristics of a *flow collector* is presented in Table I.

1) ***IPFIXcol2 [8]****:* IPFIXcol2 is a flexible, high-performance *flow collector* designed to be extensible by plugins. It has a parallelized design for high-performance, supports bidirectional flows, structured data types (i.e., lists), and company-specific information elements (Cisco, Netscaler, etc.). Its design is not modular itself, but it supports extensions by means of plugins (input, intermediate, and output) which enables *IPFIX* over TCP or UDP, anonymized IP addresses, or conversion of *flow data* to *JSON* before their storage. *IPFIXcol2* is based on *flow data storage library* (*libfds*) [9] that provides functions for *IPFIX* parsing and manipulation. Future releases are expected to allow reconfiguration, filtering, and aggregating flows at runtime.

2) ***Goflow [10]****:* Goflow gathers network information from different flow information export protocols, serializing them in a *protobuf* format to be sent to e.g. *Kafka* using *Sarama's* library [11] or *Prometheus*. It implements a modular design, allowing different transport (e.g., *RabbitMQ* instead of *Kafka*), conversion to different formats (e.g., *Cap'n Proto* or *Avro* instead of *protobuf*), decoding of different samples (e.g., *MPLS* in addition to IP), and different metrics system (e.g., use *expvar* instead of *Prometheus*). Furthermore, it provides *Docker* support, but without integration with *Kafka* at its default level. Although it is perfectly suitable to get the data in *IPFIX* format over TCP, it is not compatible with flows exported by *OvS* due to a *IPFIX* template format mismatch.

3) ***vFlow [12]****: vFlow* is a high-performance, scalable, and reliable *flow collector* which uses *Sarama's* library for the communication with *Kafka*, and can be easily integrated with *Prometheus*, *InfluxDB*, and *Grafana*. It implements modular message queues, allowing *NATS* or *NSQ* to be used instead of *Kafka*. In terms of deployment, *Docker* and *Kubernetes* can be used in its default version.

4) ***Pmacct [13]****: Pmacct* implements a subset of multi-purpose passive network monitoring tools. It can account, classify, aggregate, replicate, and export forwarding-plane data; collect and correlate control-plane data via *BGP* and *BMP*; collect and correlate RPKI data; and collect infrastructure data via Streaming Telemetry. This set of tools are divided into *nfacct*, *pmacct*, *pmbgp*, *pmbmp*, *pmtelemetry*, *sfacct* and *uacct*, being tools dedicated to *Netflow/IPFIX* accounting, *libpcacp-based* accounting, *BGP* collector, *BMP* collector, streaming telemetry collector, *sFlow* accounting and Linux *Netlink NFLOG* accounting, respectively. Each tool works either as a standalone dockerized daemon, or as a thread of execution for correlation purposes. Its modularity is also compatible with a variety of plugins to integrate with *MySQL*, *PostgreSQL*, *SQLite*, *MongoDB*, *BerkeleyDB*, *RabbitMQ*, and *Kafka*. It implements its own *flow exporter*, therefore data types defined in templates are well understood and interpreted between *flow exporter* and *flow collector*. Furthermore, it

	IPFIXcol2	Goflow	vFlow	Pmacct	Manito	KFlow
Language	C/C++	Golang	Golang	C	Python	Kotling
License	GNU	GNU	GNU	GNU	GNU	GNU
Dockerizable	No	Yes	Yes	Yes	No	No
Flow information port protocol	NetFlow and IPFIX	Netflow, IPFIX and sFlow	Netflow, IPFIX and sFlow	Netflow, IPFIX and sFlow	Netflow, IPFIX, sFlow, Traffic Flow and Netstream	IPFIX
Flow aggregation	Yes	Yes	Yes	Yes	Yes	Yes
Serialization frameworks	protobuf	Cap'n Proto, Avro, protobuf	protobuf	Avro, protobuf	Avro, protobuf	Not specified
Modularity	No	Yes	No	Yes	No	No
Output format	FDS, JSON, IPFIX	JSON, XML	JSON	JSON, CSV	JSON, CSV, PLAIN	JSON
IPFIX template work with OvS exporter	No	No	No	Yes	No	No
Implement their own *flow exporter*	No	No	No	Yes	No	No
Modular design	No	Yes	Yes	Yes	No	No
OS supported	Debian/Ubuntu and RHEL/CentOS	Linux	Windows and Linux	Linux, FreeBSD, OpenBSD, NetBSD and Solaris	Ubuntu Server	Linux and MacOS

TABLE I: Flow collectors' main characteristics.

has a wide compatible plugins which make it perfect for heterogeneous environments.

5) ***Manito Networks Flow Analyzer [14]:*** The *Manito Networks Flow Analyzer* is a collector and parser that stores flows in *Elasticsearch* and visualizes them in *Kibana*. It can be hosted in the cloud or in a on-premise data center, providing immediate insight into network performance and behavior. This *flow collector* presents a closed environment since it is oriented to be integrated with *ELK* (*Elasticsearch* + *Logstash* + *Kibana*) services. However, the lack of modularity makes this *flow collector* a very inflexible solution to heterogeneous network environments. Its most suitable applications comprises concrete standard and non-customizable usages which require precise data visualization and scalable solutions.

6) ***KFlow [15]:*** KFlow is a *flow collector* intended for performance with low resources written in *Kotlin*. It takes care of exporting the data to *Kafka*. It is inflexible in terms of extensions and integration with other technologies. However, it is a portable application that can run over any system supporting *JVM*. It does not collect all the *IPFIX* fields defined in RFC 7012 [16], permitting only the most general ones in a compromise towards better performance.

D. Analysis Applications

Residing on top of the *flow collectors*, several *analysis applications* are responsible for extracting meaningful information from the raw monitoring data. Thus, *analysis applications* can embody different purposes, ranging from network security, anomaly detection, to performance assessment, or just enhanced visualization. All these applications require a continuously monitoring and analysis of network metrics, which are implemented through a processing pipeline: filtering, correlation, and visualization. These processes are described as follows:

- **Filtering**: process responsible for remove duplicated and/or irrelevant flows.
- **Correlation**: process responsible for extrapolate information from a large number of events, pinpointing those with especially importance for the application. This can be accomplished by looking for and analyzing relationships between events, or by applying big data and machine learning techniques.
- **Visualization**: process responsible for presenting the flows and other relevant information in a visual format, including statistics on the traffic traversing the network.

III. Open-Source Flow Monitoring Framework (OpenFlowMon)

Open-Source Flow Monitoring (OpenFlowMon) consists in a fully distributed monitoring framework, characterized by running only on open-source software. Thus, it does not require any licenses or proprietary hardware, being still capable of scaling up to large business networks. An overview of the platform is presented in Figure 3.

A. Implementation Choices

The components of the OpenFlowMon are described as follows:

- **Flow exporter**: To export *flow data* from compute nodes, OpenFlowMon uses *pmacct* as the open-source *flow exporter*. In doing so, it can exploit the main features of this software, including *(i)* each component works as a standalone daemon (including a BGP daemon for BGP multipath visibility) and as a thread for correlation purposes (i.e., enriching NetFlow with BGP data); *(ii)* integration with third-party *flow collectors*; and *(iii)* storage of data in different backends (e.g., relational DB, non-SQL DB, flat files, etc.). In the case of *flow exporters* deployed at the switches, OpenFlowMon have opted for

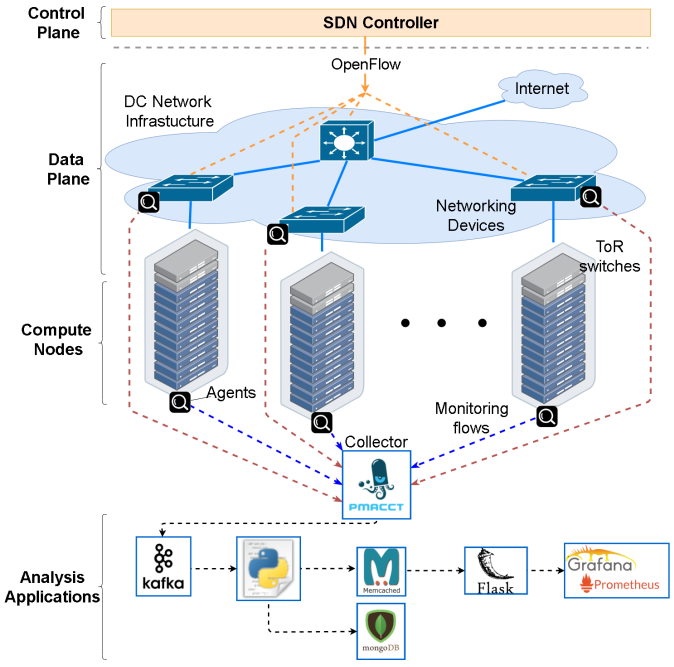


Fig. 3: Open-source monitoring testbed

OvS exporter due to its widely adoption in both hardware- and software-based switches.

- **Flow Information Export protocol**: OpenFlowMon leverages on *IPFIX* as the flow information export protocol, since it provides the best support within the remaining components of the OpenFlowMon framework.
- **Flow Collector**: In order to deploy an homogeneous scenario, *pmacct flow collector* has been chosen due to its compatibility with *OvS IPFIX exporter*. This *flow collector* is used regardless of the point where the *flow exporter* is placed, and therefore it helps providing a single platform compatible with both *flow exporter* placement options.
- **Data distribution**: For that role, *Apache Kafka* has been selected due to its scalability, reliability and performance. Also, *Kafka* is highly extensible (there are as many ways by which applications can connect and make use of *Kafka*), including *Docker* support.
- **Fast access memory**: To provide access to data for processing it in real time, a cache memory has been deployed as part of the OpenFlowMon. *Memcached* is the choice for this purpose because is simple, intuitive and support multi-thread architecture (other solutions such as *Redis* does not support this feature).
- **Permanent storage of the data**: A non-relational, dockerizable, and broadly-used database such as *MongoDB* has been selected in order to permanently store the monitoring data, accessible by a *Flask* server.
- **Dashboard**: OpenFlowMon implements a combined solution by joining *Prometheus* to scrape the data and *Grafana* to plot it.

Figure 3 depicts the *flow exporters*, namely *OvS exporters* and *pmacct flow exporters* (represented as a black lens icon), sending monitoring data via *IPFIX* (red and blue dotted lines) to *pmacct* collector. When the monitoring flows arrive to the collector, it is sent to the data distribution tool, namely *Kafka broker*, to en-queue the events. Those events are consumed and processed by a script and they are set into the fast access memory and database, respectively *Memcached* and *MongoDB*. *Memcached* is used for data that must to be quickly consumed, while *MongoDB* is used for historical and persistent data in order to create datasets for machine learning or to analyse traffic in the long term. Data from *Memcached* is posted in a *Flask* server and scrapped from it by a *Prometheus* service and presented into a *Grafana* dashboard to obtain a graphical representation.

B. Flow Monitoring Approaches

Two different flow monitoring approaches can be implemented on top of OpenFlowMon, namely:

1) **Monitoring at the switch level:** This approach places the *flow exporter* in the aggregation switches, as a central point where the north-to-south and east-to-west traffic converges. This setup is regarded as a hybrid approach between the current market standard solution (sampling the traffic at the core of the network) but with the benefit of having visibility to the east-to-west traffic. This solution is exemplified in Figure 4 in the Aggregation row of switches.
2) **Monitoring at the compute node level:** This approach places the *flow exporter* at the end nodes (source and/or destination of the traffic). The placement of *flow exporters* at the sources/sinks of traffic, exposes dissagregated information, presenting a higher level of detail and precision. This solution is exemplified in Figure 4 in the compute node row.

IV. Experimental Evaluation

This section experimentally compares the two monitoring approaches supported by the proposed OpenFlowMon framework, namely *(i)* at the switch level; and *(ii)* at the compute node level.

A. Scenario and Experiment Description

The evaluation scenario, as depicted in Figure 4, represents a typical Tier 3 hierarchical data center network where each access switch provides network connectivity to compute nodes.

The network topology is emulated and virtualized via *Mininet*, with each switch in the topology implemented through *Open vSwitch*. The access switches provide connectivity towards nine *Kubernetes* cluster nodes. In turn, each *Kubernetes* cluster node is composed by four pods, of which one is simulating the Internet and the rest are acting as compute nodes in the scenario. Finally, all nodes have two network interfaces - one for the management plane and another for the data plane, therefore monitoring is carried out-of-band.

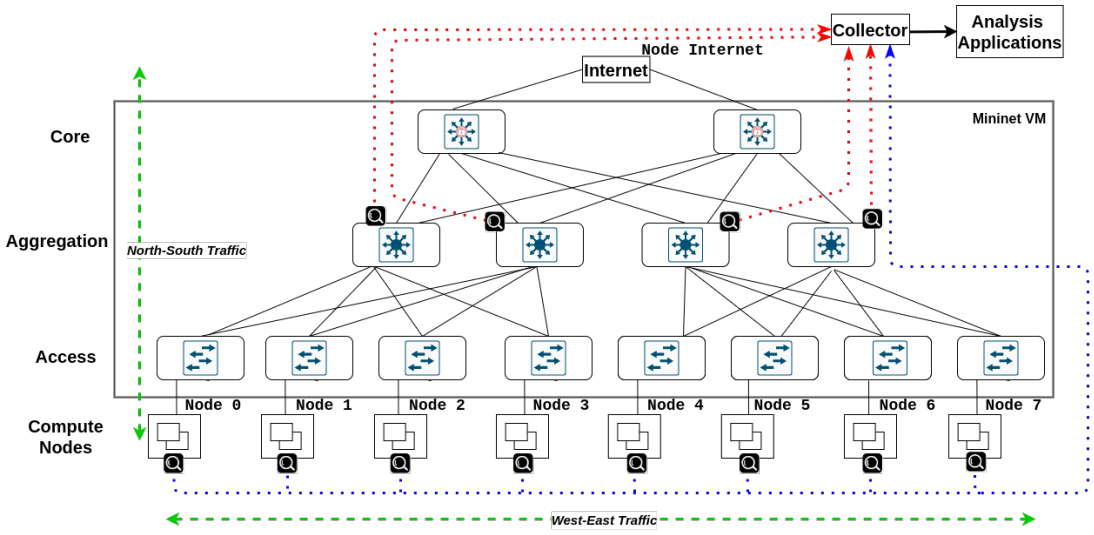


Fig. 4: Tier 3 Hierarchical Data Center Network

Each compute node is acting as a network traffic generator or consumer. Namely, there are four nodes that generate traffic (i.e., Node 1, Node 2, Node 5 and Node 6 in Figure 4), and five nodes that consume that traffic (i.e., Node 0, Node 3, Node 4, Node 7 and Node Internet in Figure 4) in a pairwise fashion, making a total of 16 different flows. To do so, every node generating traffic establishes a TCP session with all the consumers using the *iperf*[1] tool, emulating not only north-to-south traffic (i.e., between compute nodes to the Internet) but also east-to-west traffic (i.e., between compute nodes). *Flow data* are collected and sent to the *flow collector* by *flow exporters* in both network switches and compute nodes. The collected *flow data* are then filtered and processed by the *Analytics Application*.

The performed experiments aim assessing the advantages and disadvantages of both monitoring at the switch level and monitoring at the compute node level.

B. Testbed Setup

The evaluation scenario is deployed over a single server with the following specifications: 40 Intel(R) Xeon(R) CPUs E5-2630 v4 @ 2.20GHz (2 Sockets), 128 GB of RAM and 8TB of disk. It hosts all the needed VMs as follows:

- **Mininet VM:** Ubuntu 16.04 with a 4.4.0-201-generic kernel; It has 12 CPU(s), 12GB of RAM and 40GB of disk.
- **Kubernetes (K8) VMs:** Ubuntu 16.04 with a 4.4.0-201-generic kernel; It has 2 CPU(s), 4GB of RAM and 30GB of disk.
- **Monitoring VM:** Ubuntu 16.04 with a 4.4.0-201-generic kernel; It has 4 CPU(s), 4GB of RAM and 32GB of disk.

C. Experimental Results

Two experiments are carried out. In the first, bandwidth is measured in the *flow collector* and later in the *analysis application* in order to compare the monitoring information when visualized at them (Figure 5). In the second, the monitoring traffic is analyzed in terms of overhead introduced (Table II).

Results are presented for both supported flow monitoring approaches, namely when relying on monitoring at the switch level or at the compute node level.

	# of packets per second	Average packet size (bits)	Bandwidth (bps)
Compute nodes level approach	0.120	3064	366.06
Switch level approach	5.46	1120	6128.33

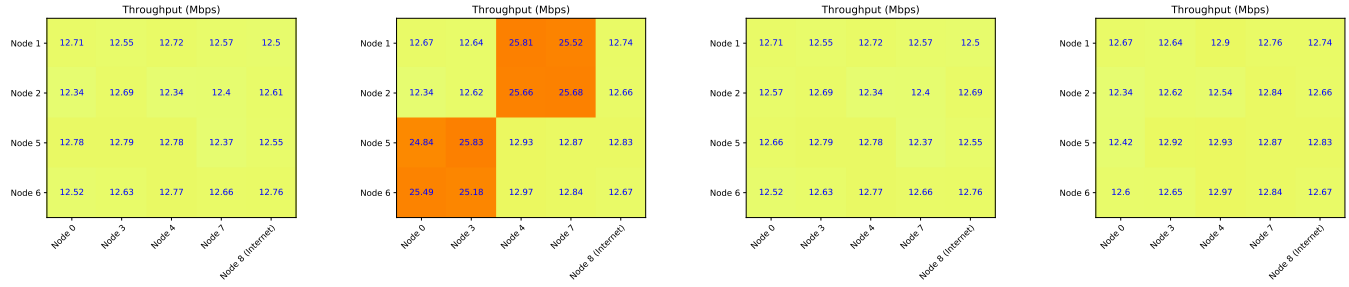
TABLE II: Overhead of each Monitoring Approach

Figure 5 presents the results used for comparison between both approaches. As expected, the metrics obtained by both approaches are similar, except for the darker areas of Figure 5b. These areas represent east-to-west traffic that goes through more than one switch on its path and, therefore, the metrics are obtained twice. This issue has to be solved by the *analysis application* in the monitoring framework, which results are presented in Figure 5c and Figure 5d.

Table II shows selected metrics related to the overhead introduced by each monitoring solution. It is possible to observe a higher amount of transmitted bytes, and consequently higher bandwidth, of monitoring packets generated at the switch compared with monitoring at the compute node. At compute node level, a pre-filter can be configured to ignore determined type of traffic, avoiding generating monitoring packets that are not of interest. However, at switch level such configuration is not possible, requiring post-processing to eliminate traffic not relevant to the measured flow. Also, as shown in Table II, the monitoring network suffers a higher saturation when deploying the *flow exporter* at the switch.

From the deployment experience, the *flow exporter* at the switch is constrained by the lack of powerful methods to

[1]https://iperf.fr/



(a) via Compute @ pccmat (b) via Switch @ pccmat (c) via Compute @ Analysis App (d) via Switch @ Analysis App

Fig. 5: Throughput Monitoring at Different Levels

export flows beyond port mirroring and the used *OvS exporter*. The configuration of the latter is very simple albeit not very powerful or flexible for its configuration. *OvS exporter* is poorly optimized and not all collectors can interpret its *flow data* because the IPFIX and NetFlow templates do not match those that most collectors can process.

On the other hand, *flow exporter* at compute-nodes level, presents a higher choice of *flow exporter*'s varieties usually based on the well known pcap library. Their configuration is very flexible and extensive and allows higher granularity by pre-processing. The possibility of using the same software for exporter-collector pair allows a better coupling between them without format compatibility problems. However, it suffers scalability problems since it requires configuring as many *flow exporters* as end nodes.

V. Conclusions

The way networks are deployed and are operated have significantly changed in the last decade, demanding new and more fine-grained mechanisms to monitor and assess the status of the network. In doing so, Open-Source Flow Monitoring Framework (OpenFlowMon) is proposed as a fully distributed monitoring framework, characterized by running only on open-source software. Through the usage of the proposed framework, a performance and overhead assessment is made regarding both switch level and compute node level monitoring (i.e., the placement of the *flow exporters*).

Results show that deploying *flow exporters* in the the compute nodes allows a higher simplicity and flexibility as it not only provides pre-processing but also reduces the overhead introduced by the monitoring process. However, the solution needs to scale to a higher number of compute nodes, which typically are much higher than the switches in a datacenter. Moreover, results also showed that in the case of deploying *flow exporters* at the switch a potential complex post-processing, for east-to-west traffic may be required.

Acknowledgments

This work has been (partially) funded by H2020 EU/TW 5G-DIVE (Grant 859881) and H2020 5Growth (Grant 856709).

References

[1] W. Stallings. "SNMP and SNMPv2: the infrastructure for network management". In: *IEEE Communications Magazine* 36.3 (1998), pp. 37–43. DOI: 10.1109/35.663326.

[2] Jakub Svoboda, Ibrahim Ghafir, Vaclav Prenosil, et al. "Network monitoring approaches: An overview". In: *Int J Adv Comput Netw Secur* 5.2 (2015), pp. 88–93.

[3] Ben Pfaff et al. "The design and implementation of open vswitch". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 2015, pp. 117–130.

[4] S Kamei and T Kimura. *Cisco IOS netflow overview. Whitepaper*. 2006.

[5] Nevil Brownlee. "Flow-based measurement: IPFIX development and deployment". In: *IEICE transactions on communications* 94.8 (2011), pp. 2190–2198.

[6] Lutz Mark et al. *IP Flow Information Export (IPFIX) Implementation Guidelines*. RFC 5153. Apr. 2008.

[7] Elisa Jasinska. "sFlow–I can feel your traffic". In: *23C3: 23rd Chaos Communication Congress*. 2006.

[8] Lukas Hutak. "A New Generation of an IPFIX Collector". In: *7 th Prague Embedded Systems Workshop*. 2019, p. 33.

[9] *CESNET/libfds*. URL: https://github.com/CESNET/libfds (visited on 07/14/2021).

[10] *cloudflare/goflow*. URL: https://github.com/cloudflare/goflow (visited on 07/14/2021).

[11] *Shopify/sarama*. URL: https://github.com/Shopify/sarama (visited on 07/14/2021).

[12] *EdgeCast/vflow*. URL: https://github.com/EdgeCast/vflow (visited on 07/14/2021).

[13] Paolo Lucente. "pmacct: steps forward interface counters". In: *Tech. Rep.* (2008).

[14] Tyler Hart. *tyjhart/flowanalyzer*. URL: https://github.com/tyjhart/flowanalyzer (visited on 07/14/2021).

[15] Boris Sukhinin. *sukhinin/kflow*. URL: https://github.com/sukhinin/kflow (visited on 07/14/2021).

[16] Benoit Claise and Brian Trammell. *Information model for IP flow information export (IPFIX)*. Tech. rep. RFC 7012, September, 2013.