Fine-Grained and Distributed Traffic Monitoring Platform in Software-Defined Networks

by

Phan Xuan Thien

Dissertation

submitted to the Department of Informatics

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

SOKENDAI (The Graduate University for Advanced Studies)

September 2019

Committee

Advisor	Dr. Kensuke FUKUDA Associate Professor of National Institute of Informatics and SOKENDAI
Sub-advisor	Dr. Yusheng JI Professor of National Institute of Informatics and SOKENDAI
Examiner	Dr. Michihiro KOIBUCHI Associate Professor of National Institute of Informatics and SOKENDAI
Examiner	Dr. Takashi KURIMOTO Associate Professor of National Institute of Informatics and SOKENDAI
Examiner	Dr. Yuji SEKIYA Associate Professor of The University of Tokyo

Abstract

Traffic engineering is an important issue for network operation. It is essential for most networks since it enables network operators and service providers to ensure efficient use of network resources and proper network performance for applications and services. Traffic engineering adapts the routing of traffic based on the network conditions and optimizes traffic demand and capacity such as big flow migrations, fine-grained QoS control, anomaly elimination. Therefore, it requires integrating of traffic monitoring and control capabilities in the network. However, in the past and current networks, network monitoring and control are conducted independently, and current monitoring techniques require separate hardware deployment or software configuration, making it hard to implement traffic engineering and other network management applications.

Different monitoring techniques are used for monitoring networks such as sFlow, NetFlow, Simple Network Management Protocol (SNMP), and other telemetry tools. Though SNMP is integrated in most network devices, it limits counters to aggregate traffic for the whole switch and each of its interfaces, disabling insight into flow-level statistics necessary for fine-grained traffic engineering. While packet-sampling tools like sFlow and NetFlow mostly require separate hardware deployment for the flow collector and they are not integrated with existing control protocol/APIs in the networks. Therefore, existing monitoring techniques remain inflexibility and drawback to meet traffic engineering requirement.

Recently, Software Defined Network (SDN) has been introduced to solve the drawbacks of network control and monitoring in current networks. SDN in general and OpenFlow as its current implementation instance in particular, provides a centralized visibility with global network and application information, a programmability without a need to handle individual network elements, and a traffic flow based controllability with flow table pipelines in OpenFlow switches making flow management more flexible and efficient. With these supports, traffic engineering mechanisms can be implemented flexibly and intelligently in SDN/OpenFlow compared to conventional approaches.

For traffic monitoring functionality, SDN/OpenFlow employs a default monitoring mechanism that records statistics of flows using forwarding flow tables. This monitoring mechanism may not ensure an effective performance for fine-grained monitoring (i.e., monitoring network traffic with high granularity) due to two main reasons: (1) installing larger number of fine-grained rules without coarse-grained forwarding rules increases cache misses, which results in additional latency that decreases the performance of the switch as it may send a Packet-In message to the controller to ask for a forwarding rule and wait for controller' response; (2) installing larger number of flow entries make the sizes of flow tables greater, which increases the latency due to the lookup process for matching incoming packets with the flow entries. Therefore, an effective monitoring method with better performance at switch is critically essential to improve traffic monitoring in SDN/OpenFlow for fine-grained traffic engineering.

In addition, in most SDN based networks, a number of selected SDN/OpenFlow switches for monitoring tasks independently monitor traffic flows. These switches may consume huge resources (e.g., throughput, CPU, memory usage) to perform monitoring tasks. When monitoring a network in a distributed scenario, for each flow that traverses through multiple monitoring switches, the switch along the flow path records almost the same statistics of the flow. This introduces a duplication issue of flow monitoring as a single switch in the flow path is enough to monitor the flow statistics. This duplication results in redundant flow-based monitoring rules in switches that consume significant resources of the switches and the network, and may cause serious problem to the performance of the switches and the network due to their limited resources. Therefore, a distributed monitoring capability that can distribute the monitoring load over multiple monitoring switches in the network and eliminate duplication is critically essential for monitoring in distributed scenarios.

In this dissertation, we propose a systematic approach that integrates fine-grained traffic monitoring capability to a capable traffic control platform, i.e., SDN, for traffic engineering applications. Our approach concentrates on both solving the monitoring performance limitation at SDN/OpenFlow switch (i.e., OpenFlow software switch in particular), and enabling a distributed monitoring capability at controller for flexible and low overhead flows monitoring that operates independently from the forwarding functionality in the switch. In the proposed method, network flows statistics are actively monitored based on monitoring match fields (e.g., 5-tuple match fields) that can be defined by controller applications. Traffic flows are forwarded based on flow entries of flow tables while their statistics at a fine-grained level are monitored at a monitoring module that is independent from the forwarding tables. The approach ensures network flows are monitored at switch with low overhead independent of the forwarding functionality of the switch, and applications use such flow data statistics via extended OpenFlow APIs. As a result, the proposal decreases the monitoring overhead in the switch even for monitoring large number of active flows.

Furthermore, for distributed monitoring scenarios, we propose a distributed monitoring method that eliminates the redundant monitoring rules, and distributes the monitoring load over multiple monitoring switches in a balancing fashion. The proposed method detects duplicated monitoring rules and for each duplication; it selects a switch with highest availability among the switches along the path to monitor the flow and eliminates the redundant monitoring rules in the other switches in a balancing fashion. The switch selection is adaptive based on the available status of the switches, which is frequently updated in each statistics query time of the controller. The proposed method decreases the number of monitoring rules per switch, therefore it decreases the monitoring load of the switches and the entire network.

We implemented our proposed methods as a monitoring platform integrated to SDN/ OpenFlow called SDN-MON. We also designed and implemented a dedicated protocol for the communication between the switches and the controller for exchanging monitoring control messages and transmitting the monitoring data. The designed protocol is implemented with OpenFlow based formats to integrate it to OpenFlow standard. We conducted a number of experiments based on the implementation instance to show the effectiveness of our proposals. The experimental results demonstrate a low monitoring overhead at switch, and a low processing time of the proposed distributed monitoring mechanism at controller.

Acknowledgments

The work presented in this dissertation would not have been possible without the association and support of many people. I would like to take this opportunity to express my sincere gratitude and appreciation to all those who have supported me throughout my research to make this Ph.D dissertation come true.

First and foremost, I would like express my sincere gratitude to my adviser, Professor Kensuke Fukuda, for his valuable advice and support for my Ph.D study. His guidance helped me a lot in the research and the writing of this dissertation. I appreciate his advice and instructions during my study, which contribute to the accomplishment of the research and the dissertation.

I would like to express my special thanks to my other advisers, Professor Shigeki Yamada and Professor Yusheng Ji, who have great support and always give me helpful advice throughout my research progress, as well as motivates me to overcome difficulties and challenges of the research.

I also want to give special thanks to the other professors in the dissertation evaluation committee, Professor Michihiro Koibuchi, Professor Takashi Kurimoto, and Professor Yuji Sekiya, for their valuable advice, insightful comments for my research and the dissertation.

I acknowledge the professors and staffs in The Graduate University for Advanced Studies (SOKENDAI) and National Institute of Informatics (NII) for the endless effort to provide an excellent study and working environment for my research.

Furthermore, I would like to thank fellow researchers and colleagues who have various collaboration and discussion with me throughout my research. I especially thank Professor Thoai Nam (Dean of the Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology, Vietnam), for being a mentor who always supports and motivates me throughout my Master and Ph.D studies. I also want to thank Professor Le Dinh Duy (NII, Japan, and University of Information Technology, Vietnam), Dr. Tran Minh Quang (Ho Chi Minh City University of Technology, Vietnam), Dr. Nguyen Kien (Chiba University, Japan), Dr. Le Duc Tung (IBM Research, Japan), Dr. Nguyen Phi Le (Hanoi University of Science and Technology, Vietnam), Dr. Phan Le Sang (NII, Japan), Dr. Nguyen Son Hoang

Quoc (KDDI, Japan), Dr. Romain Fontugne (IIJ, Japan), Dr. Johan Mazel (NII, Japan), Dr. Tiago Oliveira (NII, Japan), Ignacio Dominguez Martinez-Casanueva (Technical University of Madrid, Spain), and my other mates at National Institute of Informatics who have been my collaborators and companies on the way pursuing my Ph.D. In addition, I thank Professor Yuji Sekiya and Dr. Marc Bruyere for their support on the experimental environment at The University of Tokyo, as well as related discussion and advice on the experimental works in the dissertation.

Last but not least, I would like to send warmest thanks to my family, especially my Parents, my Brothers and Sisters for their enduring support, spiritual encouragement during my Ph.D study, and for always being the greatest motivation for me throughout my studies, my career and my life.

Dedicated to my Family.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Network traffic monitoring

Traffic engineering is an important issue in network and security operation. It is beneficial for most networks since it enables network operators and service providers to ensure proper network performance and efficient use of network resources. Traffic engineering adapts the routing of traffic based on the network conditions and optimize traffic demand and capacity such as big flow migrations, fine-grained QoS control, anomaly elimination. Thus, it requires feedback loop between control and monitoring. For instance, in order to ensure efficient routing of network traffic through network links, traffic engineering requires detecting congestion of network traffic in the links to determine proper react to mitigate the congestion (e.g., reroute/split the traffic from congested links to other available links). Furthermore, once the congestion is detected and new routes for congestion mitigation are determined, traffic engineering requires to enforce the routes to corresponding network devices to mitigate or eliminate the traffic congestion. Such network management requires a monitoring and control capability that integrate both monitoring statistics of network traffic (e.g., bandwidth) for analysis to make react decision, and quickly enforcing policy changes to network devices to adapt the network traffic with the changes in network conditions.

However, in the past and current networks, network monitoring and control are conducted independently, and current measure techniques require separate hardware deployment or software configuration [78]. This drawback make it hard to implement traffic engineering and other network management applications. Specifically, different monitoring techniques are used for monitoring networks such as Simple Network Management Protocol (SNMP) [7], sFlow [57], NetFlow [10], and other telemetry tools. SNMP is one of the most used protocols to monitor network status. SNMP can be used to request per-interface port-counters and overall node statistics from a switch, and it is integrated in most network devices. Monitoring using SNMP is achieved by regularly polling the switch, though switch efficiency may degrade with frequent polling due to CPU overhead. Although vendors are free to implement their own SNMP counters, most switches are limited to counters that aggregate traffic for each of its interfaces, disabling insight into flow-level statistics necessary for fine-grained traffic engineering. It is worth noting that fine-grained traffic monitoring/engineering indicates monitoring/controlling network traffic with high granularity, which requires fine-grained monitoring rules with more matching fields than simple origin-destination pair (e.g., 5 tuple matching fields with source IP address, source port number, destination IP address, destination port number, protocol number). NetFlow and sFlow are a passive flow-based monitoring tool that collects samples of traffic and estimates overall flow statistics based on the samples, which is considered sufficiently accurate for long-term statistics. NetFlow and sFlow are usually used with a 1-out-of-n random sampling, and assumes the collected packets to be representative for all traffic passing through the collector. Every configurable time interval, the router sends the collected flow statistics to a centralized unit for further aggregation. Drawbacks of packet-sampling is the fact that small flows may be underrepresented, and multiple monitoring nodes along a flow path may sample exactly the same packet and therewith over-represent a certain traffic group that may decrease the monitoring accuracy. These monitoring techniques mostly require separate hardware deployment for the flow collector and they are not integrated with existing control protocol/APIs in the networks. Therefore, existing monitoring techniques introduces inflexibility and drawback to meet traffic engineering requirement.

1.2 Software-Defined Network (SDN)

As explained above, network management, especially traffic engineering, is essentially important to ensure network performance and efficient routing of traffic for computer networks. Traffic engineering has been exploited in the past and current networks (e.g., ATM, MPLS, and IP networks [2] [21]). However, these traditional networks mostly embed networking functions and protocols into hardware devices (e.g., switches, routers) that limits the ability to access and control networks (e.g., routing rules are almost fixed with pre-defined policies embedded into the routing devices and hard to change [51]). For example, to add/move any device or change a network policy, network operators may need to configure multiple switches, routers, firewalls, update ACLs, VLANs, QoS, and other mechanisms using device-level configuration tools, which may take non-trivial time and effort to enforce and ensure the consistency. Because of this static nature, these networks mostly cannot dynamically adapt to changing traffic to meet the changing demands of end-user applications and services.

In order to deal with the drawbacks of the past and current networks in terms of network management and traffic engineering, Software-Defined Network (SDN) [51] [49] [64] has been introduced recently. SDN in general, or OpenFlow as its successful implementation in particular, is an architecture that can operate networks with many flexibilities. SDN enables network operators to enforce routing rules and other network policies in real-time, where rules can be installed into network devices and enforced consistently by directly programming. This architecture decouples network control and forwarding functions that allows programmability for network control and enhances flexible manageability for various network services and applications. The decoupling enables network infrastructure to process the network traffic in line rate speed and allows network intelligence to be logically centralized in a controller with a well-defined OpenFlow protocol. SDN provides a global view over the whole network where network intelligence is logically centralized that enables network operators to observe the networking status and control their network in real time by programming. By the network control capability, SDN facilitates network management and traffic engineering. With SDN, network operators can quickly apply routing policies to adapt with changes in network conditions for bandwidth optimization, resource utilization, and quality of service for end-users.

Specifically, with SDN and its OpenFlow implementation, traffic engineering mechanisms can be implemented flexibly and intelligently as a centralized traffic engineering system compared to conventional approaches such ATM based, IP based, and MPLS based traffic engineering due to the advantages of SDN architecture [2]. This is because SDN provides: centralized visibility with global network and application information (e.g., network resource limits, QoS requirements); programmability without a need to handle individual network elements, i.e., SDN/OpenFlow switches can be programmed and dynamically reprogramed at the centralized controller to allocate network resources efficiently, avoid network congestion, or enhance network performance to ensure QoS for end-user services/applications; traffic flow based controllability with flow table pipelines in SDN/OpenFlow switches makes flow management more flexible and efficient.

With its benefits for network control, management and traffic engineering, SDN is considered as an innovative architecture for today's networks, applications and services. It is being widely used and deployed by the research communities (e.g. SDN in campus environment [53]) and industries (e.g. Google deployed B4 [33] [27], a software-defined WAN connecting Google's data centers across the planet) and being supported big hardware vendors (e.g., BigSwitch, Pica8, NTT, Cisco, HP, Brocade, Juniper, VMware, Extreme Networks,...) by integrating SDN architecture and OpenFlow protocol into their switches and routers. A variety of SDN based applications and services are developed with flows statistics acquired from network traffic monitoring [85] such as traffic engineering mechanisms (e.g., [2]), traffic matrix estimation (e.g. OpenTM [77]), network utilization (e.g. FlowSense [87]), network security (e.g. FlowGuard [32], OFX [68], anomaly detection methods [88] [23] [25], DDoS attack protection [82]), data center and cloud services [44] [84] [54] [50], and wireless and mobile cloud services [36] [35].

1.3 Monitoring in SDN

In SDN or any conventional networks, network management or traffic engineering applications require monitoring the networks to acquire necessary information for analysis and react decision making. This indicates that in order to achieve the control capability in SDN, network traffic monitoring is a key factor since it is the source of any information required to observe networking status and control the network. In SDN, particularly OpenFlow as its only implementation instance at the current stage, packets are forwarded based on the flow entries in the flow tables of switches through a matching mechanism. Each flow entry includes match fields for matching with the headers of incoming packets, actions to indicate which actions that the switch performs for the packets, and related counters (i.e., packet counts, byte counts) for monitoring the network traffic for management purpose. In each switch, for each incoming packet, the switch finds a matching flow entries based on the packet's header and the match fields of the flow entry. If a match is found, the switch performs forwarding and other actions to the packet as defined in the actions field in the flow entry, and record the packet count and byte count statistics. If no match is found, the switch sends a Packet-In message to the controller to ask for a flow entry for the packet, and then install a new flow entry to forward and monitor packet based on the instruction sent from the controller. By that mechanism, SDN/OpenFlow forwards the network traffic flows and monitor the flows based statistics through the counters in the flow entries of each switch for network management.

Some recent approaches propose deploying additional systems with sFlow [57] and NetFlow [10] tools for OpenFlow for traffic monitoring. By using sFlow and NetFlow, these approaches can achieve a low monitoring overhead for network switch/router since the monitoring tasks are conducted in separate servers. Although sFlow and NetFlow are capable tools for network monitoring, they are not integrated in SDN/OpenFlow protocol. Thus, it requires external hardware deployment and configuration to integrate monitoring functionality to SDN, which is not flexible to implement traffic engineering in SDN.

Other approaches propose extension methods to improve the monitoring performance at switch. Several approaches [9] [87] [78] propose reducing the amount of switch-controller interactions and delays in the control channel and use the flow entries of flow tables of the switch to monitor traffic flows. Although these approaches can reduce the number of switch-controller overhead, the significant monitoring overhead due to the increasing size of the flow tables with fine-grained flow entries may still remain that limit the throughput performance of the switch. Other approach [81] proposes in-band traffic monitoring method at switch which integrate monitoring modules in switch for flows monitoring. Although the approach brings flexibility for the traffic flows monitoring as the monitoring modules can be implemented independently from the flow tables of the switch, the throughput performance and monitoring overhead of these approaches are still limited. On the other hand, several approaches [70] [77] propose distributed monitoring capability for the SDN controller for traffic flows monitoring. It is worth noting that in this dissertation, the distributed monitoring term indicates monitoring network traffic where multiple switches are used for the monitoring task. These approaches suggest distributed methods and reduce monitoring overhead at the controller. However, in these approaches, the traffic flows are monitored by the default mechanism of switch, which remain the performance limitation in terms of throughput at switch. Overall, the above approaches discuss partly on either reducing monitoring overhead at switch or reducing monitoring overhead of network for distributed monitoring scenario where a systematic approach that solves both issues is essential for in-band network monitoring in SDN, especially for fine-grained monitoring of network traffic. The term, systematic, means that both the issues in SDN traffic monitoring are considered as a whole system: the issue of overhead per monitoring rule in switch, and the issue of overhead in controller and control channel for a typical SDN monitoring deployment where multiple switches and a controller are used for the monitoring task.

1.4 Problem statement

The above analysis indicates that integrating fine-grained monitoring in OpenFlow is important for traffic engineering applications that use integrated APIs. OpenFlow employs a default monitoring mechanism that record statistics of flows using forwarding flow tables. However, this naive mechanism may not ensure to achieve an efficient performance for fine-grained monitoring due to two reasons: (1) For fine-grained monitoring of traffic flows, each OpenFlow switch requires to install a larger number of flow entries (with fine-grained matching fields) than the number of flow entries required for basic forwarding. Installing fine-grained rules without coarse-grained forwarding rules increases the possibility of miss-match in the matching process of the switch, which results in additional latency as the switch must send a Packet-In message to ask the controller for a corresponding rule and wait for its response. This latency downgrades the performance (i.e., throughput) of the switch; (2) For fine-grained flows monitoring, the required number of flow entries to be installed in each switch is greater than the number of flow entries for basic packet forwarding. This makes the size of flow tables greater, and therefore increases the latency due to the lookup process for matching incoming packets with the flow entries. This factor introduces more latency for the switch processing that degrades its throughput performance, especially it may make the switch become a bottleneck when monitoring a large number of flows. Therefore, an efficient monitoring method for better monitoring performance at SDN/OpenFlow switch, especially software switch, is necessary to benefit SDN based network management. This is the first objective of the research in this dissertation.

Moreover, in most SDN based networks, a number of SDN/OpenFlow switches are selected for monitoring tasks. These switches independently monitor traffic flows and may consume huge resources (e.g., throughput, CPU, memory usage) to perform monitoring tasks, especially for network management services that require monitoring a large number of flows. In addition, when monitoring in a distributed scenario, for each flow traversing through multiple monitoring switches in the network, the switches along the flow path record almost the same statistics of the flow. This is considered as duplication of flow monitoring as a single switch in the flow path is enough to monitor the flow statistics. This duplication results in redundant flow-based monitoring rules in switches that consume significant resources of the switches (e.g., throughput, CPU, memory) and the network (i.e., communication cost in the control channel for transmitting recorded statistics of redundant flows to the controller in each flow-based statistics query, the computational cost of the controller for processing the redundant flow-based monitoring rules). This may cause serious problem to the performance of the switches and the monitoring network as the switches normally have limited resources, and especially when monitoring large number of flows. Therefore, a distributed monitoring capability that allows distributing the monitoring load, which is represented by flow-based monitoring rules, over multiple monitoring SDN/OpenFlow switches in the network and eliminate the duplication of monitoring rules is critically essential for monitoring traffic flows in distributed scenarios with multiple monitoring switches and for increasing the monitoring scalability. This is the second objective of the research in this dissertation.

1.5 Criteria in designing monitoring method for SDN

To integrate fine-grained traffic monitoring for traffic engineering in SDN/OpenFlow, a traffic monitoring method requires to embed monitoring functionality in SDN/OpenFlow switches for the monitoring task. However, due to the limitation of the resources of a SDN/OpenFlow switch (e.g., CPU, memory, bandwidth), the possibly large monitoring overhead in switches is a critical problem. Typically, monitoring overhead in a switch is caused by the computation of the monitoring process (i.e., store, lookup, and update monitoring records) and the memory usage for storing the monitoring records. This overhead directly downgrades the throughput of the switch and can be computed by the decrease of the throughput in the switch compared to the base line (i.e., without monitoring). Therefore, an integrated network monitoring method for SDN/OpenFlow switch must reduce the computation cost of the monitoring process, and the number and size of each monitoring rule in order to reduce the monitoring overhead in switches.

For distributed monitoring scenarios, due to duplication issue when multiple SDN/OpenFlow switches independently monitor traffic flows, and considering the limitation of resources in each monitoring switch, a distributed monitoring method must reduce the number of monitoring load in each switch by reducing the number of duplicated/redundant monitoring records in each switch. Moreover, to enhance the effectiveness of the collaborative monitoring in multiple monitoring switches, distributed monitoring method must assign the monitoring records into the switches in a balancing fashion, so that the monitoring load in the switches could be balanced. In addition, to reduce the overhead caused by the monitoring data and instructions exchanges in the communication channel, monitoring method/platform must minimize the size and number of monitoring records to be transmitted to the controller in each query.

In summary, the following four criteria were considered for our design of the SDN/OpenFlow based network traffic flows monitoring platform:

- Switch monitoring overhead minimization (i.e., switch throughput maximization)
- Controller processing overhead minimization (i.e., processing time of the distributed monitoring algorithm minimization)
- Communication overhead minimization
- Number of monitoring rules per switch minimization and monitoring load balance maximization

1.6 Our proposal

In this dissertation, we propose a systematic approach for traffic flows monitoring in SDN, specifically OpenFlow as its only implementation at the current stage. We aim at integrating a fine-grained traffic monitoring capability in OpenFlow for traffic engineering applications.

Different from existing approaches where partly discuss on solving either monitoring performance limitation at OpenFlow switch or overhead in distributed monitoring scenarios (i.e., monitoring with multiple monitoring SDN/OpenFlow switches in network), we focus on proposing an approach that solves both of the problems (i.e., reducing monitoring overhead for SDN switch, and reducing monitoring overhead in distributed monitoring scenarios with multiple monitoring switches) to enable a fine-grained and distributed monitoring capability for traffic engineering in SDN.

Firstly, we propose a platform for traffic flows monitoring at SDN/OpenFlow switch (named as SDN-MON). The key idea behind our proposal is to integrate into SDN switch a flows monitoring capability to enable flexible and low overhead flows monitoring in the switch that is independent from the forwarding functionality of the switch. The proposed monitoring method actively monitors network flows statistics based on monitoring match fields that can be defined by controller applications (e.g., 5-tuple match fields including source IP address, source port, destination IP address, destination port, and protocol). Traffic flows are forwarded based on flow entries of OpenFlow tables while their statistics at a fine-grained level, e.g., 5-tuple, are monitored at a separate monitoring module that is independent from the forwarding tables. The approach supports to monitor network statistics at SDN/OpenFlow switch with low overhead without affecting the forwarding functionality in the switch. As a result, the proposal decreases the monitoring overhead in the switch even for monitoring large number of active flows (e.g., thousands or hundreds thousands of active flows). We implemented the proposed method on a base software switch (i.e., Lagopus switch [60], a high performance OpenFlow software switch), and conducted a number of experiments on the based on the implementation instance to show the effectiveness of our proposal. With our proposal, new flexible monitoring functionality for SDN is introduced with a small overhead.

Secondly, we propose a distributed monitoring method to enable distributed monitoring capability for SDN. Our method eliminates the redundant flow-based monitoring rules, and distribute the monitoring load over multiple monitoring OpenFlow switches in a balancing fashion. The proposed method detects duplicated flow-based monitoring rules from the monitoring rules that the controller receive from the multiple monitoring OpenFlow switches in each query time. For each duplication, it selects a switch with the highest availability among the switches along the path to monitor the flow and eliminate the redundant monitoring rules in the other switches in a balancing fashion. The switch selection is adaptive based on the available status of the switches (represented by the existing numbers of monitoring entries in the switches), which is frequently updated in each statistics query time of the controller. The proposed method significantly decreases the number of flow-based monitoring rules per monitoring switch, therefore it decrease the monitoring load of the switches and the entire network. We implemented the proposed method and integrated into the SDN-MON. We designed and implemented a dedicated protocol for the communication between the switches and the controller for exchanging monitoring control messages and transmitting the monitoring data from the switches to the controller based on the proposed methods. We conducted a number of experiments based on the implementation instance to show the effectiveness of our proposal. The experimental results show a small processing time of the proposed distributed monitoring mechanism with around 1.6s for processing as large as a million active flows. Our proposals are integrated as a systematic approach for efficient traffic flows monitoring for SDN.

1.7 Main contributions

By integrating fine-grained traffic monitoring capability for SDN, our proposed platform brings a flexible and applicable solution for traffic engineering applications. SDN-MON provides network flow statistics at fine-grained level for network operators to implement network policies in order to enhance and optimize network performance. As SDN-MON APIs is integrated to OpenFlow as a same platform, traffic engineering applications can use both SDN-MON monitoring APIs and OpenFlow APIs at a whole for traffic control and monitoring in real time. Therefore, our proposal facilitates the implementation of traffic engineering applications and benefits network management. Main technical contributions of this dissertation consist of:

1. We propose a traffic monitoring method (i.e., SDN-MON) that can monitor flows at fine-grained level with low overhead at OpenFlow switch. The proposed method can reduces the overhead per monitoring rule in OpenFlow switch, thus reducing the total monitoring overhead when handling a specific number of flows in switch, especially for fine-grained monitoring with large number of flows.

2. Based on the traffic flow monitoring method at OpenFlow switch, we propose a distributed monitoring method for SDN. The proposal can reduces number of flow monitoring rules per switch by detect and eliminate redundant monitoring rules at switches and distribute monitoring load per switch in a balancing fashion, thus it significantly decreases monitoring overhead in switches and in the network.

3. We integrate the proposed methods as a systematic approach for traffic monitoring in SDN, including the structural designs of monitoring elements and monitoring processes for switches, controller, and a dedicated protocol for exchanging monitoring data and instructions. All of these elements and processes operate consistently as a whole monitoring solution/system for SDN/OpenFlow. We implemented the proposal on selected base software

switch and controller (i.e., Lagopus software switch [60], and Ryu controller [63]). We conduct a number of experiments based on the implementation instance in different aspects and scenarios to demonstrate the efficiency of the proposal. The experimental results on the SDN-MON switch performance demonstrate that our proposed method can achieve significantly better throughput performance than the OpenFlow default monitoring mechanism at switch. In addition, the experimental results on the distributed monitoring aspect indicate a small overhead in terms of processing time of the proposed method. We also discuss the applicability of the proposal for common networks and applications based on the experimental results of SDN-MON performance and the empirical study of requirements of the networks and applications.

1.8 Dissertation organization

This dissertation is organized as follows.

In chapter 1, we present a brief overview about SDN and network monitoring in SDN/OpenFlow. Then we analyze the problem statement, the criteria in designing a network monitoring method/platform for SDN, and introduce our proposal for the problems.

Chapter 2 introduces a background and literature review. We present a background about network traffic monitoring, SDN, an empirical study on requirements of different networks and applications for network monitoring, and discuss related proposals to the problems.

In chapter 3 and chapter 4, we describe our proposals in details. We first present our proposed method for network traffic monitoring at OpenFlow switch with small overhead in chapter 3. Based on the proposed monitoring method at switch, we propose a distributed monitoring method that enables distributed monitoring capability efficiently for SDN in chapter 4. The sections describes in details our strategies and designs of the monitoring modules and monitoring process towards the outlined design criteria. We present numerical analysis with a number of experiments based on the implementation instance of SDN-MON for both aspects, switch overhead and processing time of the distributed processing in the controller.

In chapter 5, we discuss the aspects of applications and limitations of the proposal. Chapter 6 summarizes our works and discuss the future directions.

Chapter 2

Background

2.1 Software-Defined Networking

2.1.1 Introduction

Software-Defined Network (SDN) [52] is an emerging network architecture that decouples the control plane from the forwarding plane, allow network devices to be operated in real time by direct programming at the control plane. This separation allows network infrastructure to be abstracted for network services and applications, where the network can be treated as a logical or virtual entity. SDN maintains a global view of the network in SDN controllers where network intelligence is logically centralized. With SDN, the network design and operation are largely simplified since enterprises and carriers have a control over the entire network from a single logical point. The network devices (i.e switches, routers,...) are also simplified since they only need to process the instructions from the SDN controllers without any necessity to understand and process thousands of network protocols as traditional network architecture.

SDN provides well-defined network APIs that supports implementing network services and applications including routing, bandwidth management, traffic engineering, quality of service, security, access control, storage optimization, policy management,... to meet the demand of today's business. In addition, SDN enables a network programmability that allow network operators and administrators to program any configuration of the network forwarding plane (network devices) rather than hand-code tens of thousands of configuration in thousands of devices. With a centralized intelligence at the SDN controller, the network behavior can be adjusted and operated in real-time and the deployment of new applications and network services spends much less time and effort than that in the traditional network architecture. The centralization of network state in the control plane supports network managers to flexibly manage, secure, configure, and optimize network resources through dynamic SDN programs. Additionally, SDN architecture allows network administrators to dynamically adjust network-wide traffic flows to meet the changing needs of today's data centers, carrier environments, and campuses.

SDN has been being deployed more and more widely nowadays by research communities and industrial companies (e.g. Google deployed B4 [33], a software-defined WAN connecting Google's data centers across the planet). The SDN eco-system today introduces a variety of both SDN-supported hardware switches (e.g. Cisco SDN switch, pica8, BigSwitch, Broadcom) and software switch (e.g. Open vSwitch [55]), controller implementations (e.g. NOX [28], Floodlight [20], Ryu [63], POX [59] Beacon [16]), as well as virtual deployment tools (e.g. Mininet [39]). SDN and its OpenFlow protocol is widely supported by a variety of network vendors include traditional infrastructure vendors like Cisco, HP, and IBM and startups like Pica8.

2.1.2 Architectural Overview

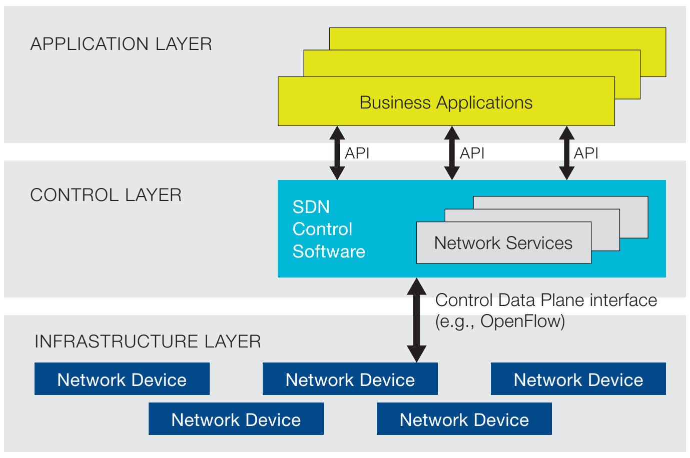


Fig. 2.1 Software-Defined Network architecture [51]

Basically, SDN architecture consists of three main layers: an infrastructure layer (or data plane) includes network devices (i.e. SDN/OpenFlow switches, routers,...); a control layer (control plane), which is a logical controller for operating and managing the network devices, and an application layer where network services and applications can be built for operating the network. The controller communicates with network devices through a standardized APIs called southbound APIs, in which the only enabler is OpenFlow protocol [42] currently. Also, the interaction between the network applications and the controller platform (control layer) is through northbound APIs.

OpenFlow controller

The network intelligence in SDN is logically centralized in the controller (network operating system) as Fig. 2.1. This controller maintains a global view of the network that is accessible through some well-defined open application programming interfaces (APIs). These APIs (typically called northbound APIs) will be used by various applications and network services that exploits the network abstraction in the controller for operating and managing the network. These APIs do not depend on proprietary software or hardware, so network administrators can write those themselves and dynamically and automatically enforce some QoS policies to manage and control a large number of network devices and traffic paths.

OpenFlow-enabled SDN switch

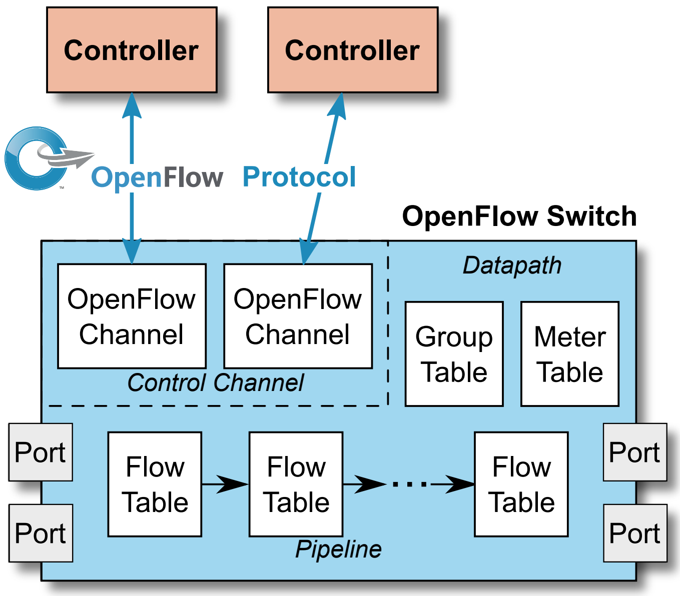


Fig. 2.2 Main components of an OpenFlow-enabled SDN switch [49]

An OpenFlow switch [49] [42] [38] consists of one or more flow tables and a group table for packet lookups and forwarding, and one or more OpenFlow channels for communication with an external controller (Fig. 2.2). The controller communicates and manages the switch via the OpenFlow protocol. Through this protocol, the controller can proactively or reactively add, update and delete flow entries in flow tables. Each flow table in the switch consists of flow entries, each consists of match fields, counters, and instructions. The match fields are used for matching with the header fields of incoming packets, the counters are for counting various network statistics related to matching flows (typically counting the number of packets and bytes passing through an OpenFlow element), and the instructions will be applied to the matching packets which informs the switch how to process the packets.

Flow matching process in OpenFlow switch

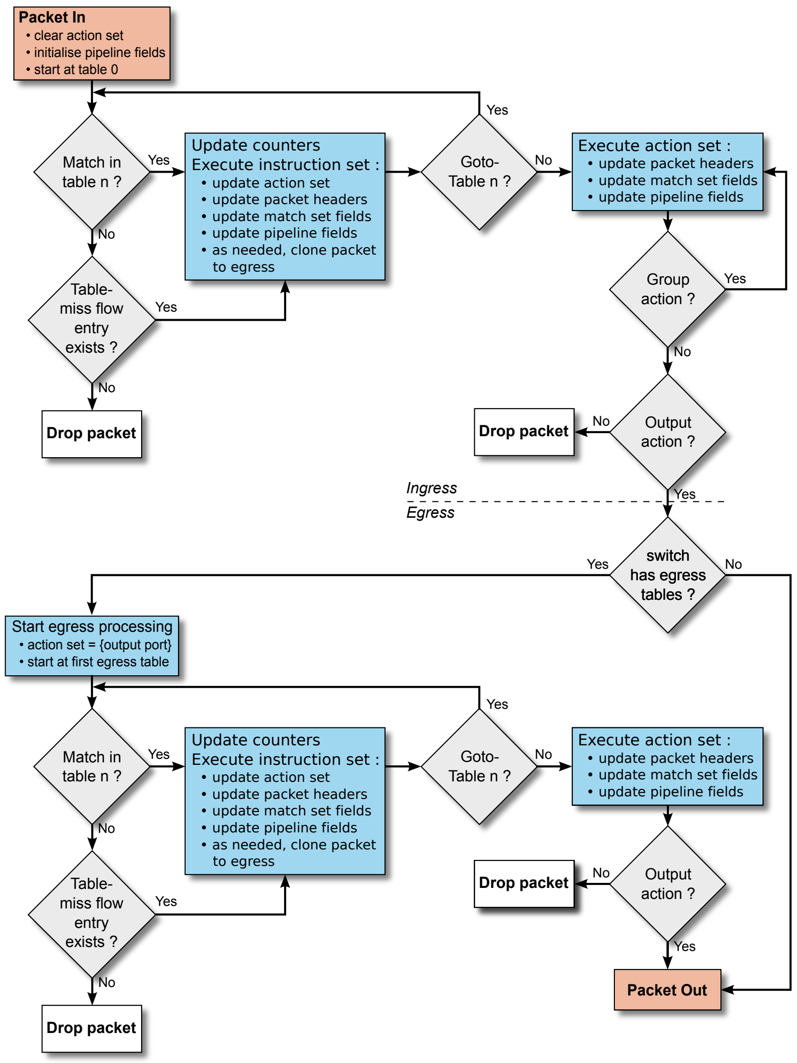


Fig. 2.3 Flow matching process in OpenFlow switch [49]

The matching process of an incoming packet starts at the first flow table and may continue to the next flow tables as showed in Fig. 2.3. A set of packet fields (e.g. source or destination MAC addresses, Ethernet type,...) will be used to match a specific flow entry in the flow tables. OpenFlow introduced multiple match fields in flow tables to perform various types of actions for the matching packets (e.g., about 40 match fields in total [49]). If a matching entry is found, the instructions in the flow entry are executed to process the packet (e.g.

direct the packet to an out-port, modify packet header fields, drop the packet,...). In case no matching entry is found, processing for the packets relies on configuration of table-miss flow entry (e.g. it may be forwarded to the controller, or dropped, or continue matching in the next flow tables). Once an OpenFlow switch receives a packet for which it does not have a matching entry, i.e., it has never seen that flow before, it sends this packet to the controller. The controller can determine how to handle this packet (e.g. drop, forward to a specific port, modify the packet contents,...) and add a flow entry to instruct the OpenFlow switches on what to process with the packets of the same flow coming in the future.

The matching decision in OpenFlow is based on priority. This means for each incoming packet, lookup will be performed for all flow tables to find all possible matches, then the match with highest priority will be selected for perform actions to process the packet. For a matching flow entry, multiple counters associated with the flow entry (e.g., per-port counters, per-queue counters, per-meter counters) are updated and the instruction set included in the selected flow entry is executed. The basis flow matching process of OpenFlow can be illustrated in Fig. 2.3, and further details on the match fields and counters can be referenced in [49] [42].

OpenFlow protocol

The communication between the network operating system (the controller) and the packet forwarding devices (i.e. SDN/OpenFlow switches) in SDN is through a standardized protocol called OpenFlow protocol [42]. This is currently the only enabler of southbound APIs. OpenFlow allows direct access to the forwarding plane of network devices (i.e. switches, routers). It is the enabler that move the network control out of the networking switches to the centralized controller. OpenFlow uses the concept of flows to identify network traffic based on matching rules (flow entries), which can be controlled by programming at the controller. It also supports to control how traffic flows should go through network devices based on parameters such as usage patterns, applications, and cloud resources. Unlike current IP-based routing, in which all flows between two endpoints must follow the same path through the network regardless of their different requirements, OpenFlow supports to program the network on per-flow basic. Thus, SDN, specifically OpenFlow as its implementation, provides a granular control, and enables the network to respond to real-time changes at the application, user, and session levels.

OpenFlow-based SDN architecture can be deployed on existing networks both physically and virtually. This means network devices can support and process both OpenFlow-based forwarding and traditional forwarding. OpenFlow is being widely supported by infrastructure vendors, in which it is implemented as a simple firmware or software upgrade. OpenFlow-based architecture can integrate with existing infrastructure of an enterprise or carrier to provide SDN functionality and benefits for those segments of the network.

OpenFlow is the only standardized SDN protocol that allows direct manipulation of the forwarding plane of network devices. It is currently being exploited and implemented by the research community and industry in various applications related to network management, traffic measurement, network and data center virtualization and wireless applications. For traffic analysis applications, OpenFlow allows for flexible, automated, fine-grained flow measurement, which makes it possible to develop innovative tools to improve traffic measurement capabilities of a switch using real-time machine learning algorithms, databases, and any other software mechanism. These innovative software mechanisms will reduce operational cost, improve network stability, and support emerging IT services.

2.1.3 Benefits of SDN for network control and management

SDN supports to address the dynamic nature of applications nowadays. It provides the flexibility and manageability for the network to adapt with the demand of today's business and significantly reduce the operations and management complexity. Its benefits for enterprises and carriers are outlined as follows.

Flexible controllability: The SDN controller can control any SDN network device including switches, routers, and virtual switches from any vendor. With SDN, network operators and managers no longer need to manage groups of devices from different vendors, which may consumes a large amount of time and effort. Instead, they can use SDN-based orchestration and management tools to quickly configure, deploy, and update devices of the whole network.

Reduced complexity: SDN offers a flexible programmability for network management and automation. It enables to develop tools that automate many management tasks that are done manually today. With the automation of management tasks, the operational overhead and network instability (caused by operator errors) can be reduced. Moreover, SDN supports cloud-based applications to be managed via intelligent orchestration and provisioning systems, thus it supports decreasing operational overhead while increasing business agility.

Increased facilitation for innovation: SDN allows network operators to directly program (and reprogram) the network in real time to meet the demands of business and user applications, thus it helps accelerating and increases facilitation for business innovation. SDN brings the ability to manage and operate the behavior of network that is critically important for proposing/developing new network services and capabilities with much less required time and effort (e.g. even in a manner of hours).

Increased network reliability and security: SDN supports to define high-level configuration and policy statements, which will be applied to the network infrastructure through

OpenFlow protocol. With SDN, there is no need to individually configure network devices each time a policy changes, or a service, application or end point is added or moved. Therefore, it decreases the likelihood of network failures caused by configuration or policy inconsistencies. With a global view and complete control over the network, SDN can ensure the quality of service, access control, traffic engineering, security, and other policies are applied consistently across the network infrastructures (including campuses, data centers, branch offices,...). Overall, SDN benefits in reducing operational costs, consistent configuration and policy enforcement, fewer errors, and more dynamic configuration capabilities.

Dynamic and granular network control: The flow-based control model of SDN supports to apply policies at a very granular level, including user, device, application and section levels, with a high abstraction and automation. This allows cloud operators to be able to support multi-tenancy while maintaining adaptive resource management, traffic isolation and security when customers share the same infrastructure.

2.1.4 SDN ecosystem and practical deployments

With various benefits for network control and management, SDN has been being supported by the research community and industrial vendors and companies. The SDN eco-system today introduces a variety of both SDN-supported hardware switches (e.g. Cisco OpenFlow switch, Pica8, BigSwitch, Broadcom) and software switch (e.g. Open vSwitch [55], Lagopus [60]), controller implementations (e.g. Floodlight [20], Ryu [63], POX [59], Beacon [16], and OpenDayLight [43]), as well as virtual network deployment tools (e.g. Mininet [39]). SDN is widely supported by a variety of network vendors include traditional infrastructure vendors like Cisco, HP, and IBM and startups like Pica8. Members of SDN consortium that represent the customer voice include Deutsche Telecom, Facebook, Google, Microsoft, Yahoo, Verizon, and so on.

SDN has been being deployed more and more widely nowadays such as deployment on campus network[53], deployment on large-scale data centers (e.g. Google B4 [33], a software-defined WAN connecting Google's data centers across the planet). The applications of SDN cover various networking fields and issues including WAN, cloud, data center, routing, big data, network virtualization, and wireless. The following section briefly outlines a part of these applications.

Network access control (NAC). Basically, the main functionality of network access control is to set appropriate privileges for users or devices accessing the networks, including access control limits, incorporation of service chains as well as appropriate quality of service. NAC generally follows the user/device as they connect from different parts of the network. An example application of SDN for network access control is campus network access control

[53]. This is the ability to control access as well as service quality to LAN and WLAN for employees, contractors and visitors based on their roles and privileges in an organization. SDN provides programmability across all access elements from campus switches to WLAN and allows assignment of the right level of privileges and quality of service for user and application flows. Comprehensive SDN controls across all devices provides ability to create end-to-end separated networks. Another example of SDN for network access control is remote office/branch (enterprise) network access control [48]. This is the ability to control access as well as service quality for remote users in disparate locations for employees, contractors and visitors based on their roles and privileges in an organization. SDN provides programmability across all access elements from remote network access devices to corporate devices, allowing control of privileges at remote branch.

Network virtualization. The basic functionality of network virtualization is creating an abstracted virtual network on top of a physical network, allowing a large number of multi-tenant networks to run over a physical network. This functionality can be spanned over multiple racks in the data center or locations, including fine-grained controls and isolation, or security services [31]. An example of SDN application for network virtualization is data center network virtualization [24]. It is the capability for creating virtualized networks that handle multiple-tenants at a time and separate traffic between different tenants. Network virtualization controllers decouple networks through direct flow control or via virtual overlay networks that can span racks or even geographic locations. Advanced SDN solutions support overlapping IP address ranges running over the same physical equipment. Virtualized gateway devices provide for communication into and out of these virtualized networks. Other SDN based network virtualization includes Network Function as a Service. It is the ability to provide existing network functions (vRouters, or virtual L4-7 functions) as an on-demand service for enterprise applications hosted within the cloud such [3] [1]. SDN is used to create dynamic paths to insert these virtual services. Virtualization of these L3 or L4-7 services enables them to be deployed on-demand and flexibly in different locations.

Data center optimization. SDN supports optimizing networks to improve application performance by detecting and taking into account affinities, orchestrating workloads with networking configuration (mice/elephant flows). One application of SDN for data center is Big Data Optimization [13] [83], which is the ability to enable improved resource utilization of hardware switches, and reduces overall computing time for faster results during Big Data analysis. SDN can be used to program the switches to provide optimal flow paths during each stage of the Big Data analysis, to enable better QoS between the servers, or dedicate more cross-links between servers based on the analysis stage. Another SDN application for data center optimization is Mice/Elephant Flow Optimization [40]. This optimization is the ability to control network infrastructure in the data center to ensure low running latency for critical business applications, while co-existing with large data set transfers for Big Data applications or video streaming. SDN can be used to set the appropriate QoS and flow rules across different ports on the network to ensure optimal use of resources based on the observing type of application flows.

Dynamic interconnect. Creation of dynamic links between locations, including between data centers, enterprise and data centers, and other enterprise locations, as well as dynamically applying appropriate QoS and bandwidth allocation to those links. An application of SDN for dynamic interconnect is dynamic enterprise VPN [79], whic is the ability to create quick connections between multiple enterprise locations to enable communications for secure conferencing or data transfer. SDN can be used to create network overlays between multiple enterprise locations to allow users or devices to communicate with each other securely for a period of time. Another SDN application is cross-domain interconnect [30] [8]. It is the ability to provide direct connections across suppliers or partners, which can be dynamically scaled-up, down or terminated. SDN can be used to create interim connections at exchange points (IXPs) between enterprises , or between enterprises and public cloud services to create instant high-speed networks for applications and devices. Other application of SDN for dynamic interconnect is bandwidth utilization [37]. It is the ability for end users to turn up bandwidth on their network links as and when they need (e.g. for dataset movement or large backups), and bring it back down when the users no longer need it. SDN is used to moderate the amount of bandwidth allowed on network links (across multiple WAN links) as well as control the QoS. SDN can also be used to instantiate instant connectivity between locations that were not connected previously.

2.1.5 Discussion on the scale of OpenFlow-based SDN

Besides various benefits for network control and management, a big concern about OpenFlow-based SDN is its scalability. This section discusses the scalability of OpenFlow-based SDN on two aspects: number of switches that a controller can manage, and number of flows a SDN switch can support.

Number of SDN switches managed by a controller.

The number of switches which can be controlled by a single OpenFlow controller is determined by the number of TCP sessions and CPU performance. Given today's server architecture, we can realistically expect a server to handle tens of thousands of TCP connections. This indicates a single server could connect with many thousands of switches. If more connectivity is needed, then additional servers can be added to form a cluster. The number of connections should not be an issue with good connection architecture.

Current generation Intel Xeon class servers have the ability to support 2 or more processor sockets with up to 16 cores per socket, providing massive amounts of processor cycles. Intel claims a single server can handle 160 million packets per second which translates to 100+ Gbps performance. This type of performance should be sufficient to cover thousands of switches. In addition, when the controller is architected and implemented correctly it will be capable of scaling up as servers are added to the network. The point is that a proper network and controller architecture is the key factor for scalability, which is not limited by the OpenFlow protocol [86]. However, due to a benchmarks on NOX (the first SDN controller) [74], it can only handle 30,000 flow initiations per second while maintaining a sub-10 ms flow install time. Therefore, although a controller can theoretically cover thousands of switches, its actual scale also depends on the how busy the network is, and the number of flows traverse through and among the network that the controller manages.

Number of flows a SDN switch can support.

The number of flows a switch can handle is basically limited by the sizes of its flow tables. OpenFlow 1.0 specifies a single flow table for the switch must match on 12 fields. Because of this requirement, most early implementations used a ternary content addressable memory (TCAM) for the flow table. These TCAM-based tables were limited to just a few thousand entries. If the table was exceeded, the packets would be handled by the switch's software. This severely limited the scalability and performance of early OpenFlow switches. Many of these scalability limitations have been resolved by using existing lookup tables when matching strictly at L2 and L3. Since OpenFlow version 1.1, the concept of multiple flow tables was introduced. It allowed each different flow table to match different fields. This flexibility provides a pathway for more flows per switch, as TCAMs could be replaced with large lower cost RAM devices. Modern switches and routers support forwarding information bases (FIB) in range of 64K to 512K entries. However, we are not able to store the OpenFlow-based flow rules in the FIB unless the flow matches on destination MAC address or destination IP prefix. Therefore, storing and processing OpenFlow's flow entries still depends mostly on the TCAM of a physical OpenFlow switch. The current SDN switches support only limited number of TCAM entries (i.e., a few thousands or a few tens of thousands entries). Thus, SDN may not support well for network applications and services that demands storing and processing a large number of fine-grained flows (i.e., fine-grained network monitoring based applications like anomaly detection, network visualization and so on).

2.2 Importance of traffic monitoring and measurement for network control and management in SDN

In order to achieve the manageability and controllability over the network infrastructure, SDN must support integration of new paradigms and services such as big data applications, cloud computing, rich multimedia content, and data centers services. Operators are responsible for configuring network policies that employ traffic monitoring mechanisms and measurement tools for detection and reaction to a wide range and dynamic network events and applications. The large scale and diversity of traffic generated from current networks results in the following problems: (1) Difficulty for network operators to effectively monitor the status and dynamics of network in short time scales; (2) Difficulty in monitoring various types of network traffic at different time scales for tasks such as congestion detection and traffic engineering to guarantee application performance; (3) Hard situations for network operators to satisfy user's expectations for delivering applications with guaranteed quality of service (QoS) that have obiquitos and accurate traffic monitoring mechanisms; (4) Requirement for interactive media applications to identify the factors that cause performance decrease along the whole end-to-end network path (e.g. to detect congestion for real-time video conferencing applications, we have to accurately monitor bandwidth changes caused by cross traffic in milliseconds to decrease the quality degradation of the video).

Traditional network devices (e.g. switches, routers) are inflexible and cannot deal with various types of network traffic due to the binding of routing rules in hardwired implementation and other obstacles. Today networking research community and industry have introduced SDN architecture and new standardized communication protocol (i.e. OpenFlow) that brings various benefits for network control and management and it is overcoming the above mentioned problems. With better capability for network management introduced by SDN, it is now easier to perform QoS control anytime and anywhere by using self-directed/adaptive mechanisms that monitors network performance and react quickly to the problems [85]. The global view of the network at the SDN controller, which is supported by the OpenFlow protocol and network monitoring and measurement mechanisms, enables a seamless network management for network infrastructure and complex applications (i.e. applications consisting of QoS and context aware components).

Besides the standard monitoring mechanism supported by the SDN architecture and OpenFlow protocol, existing monitoring techniques, which require external modules for monitoring such as sFlow [56] and NetFlow [10] [67], can be optionally used to handle the monitoring task in SDN. NetFlow is used to collect network traffic information such as source IP, destination IP, ports, protocols, bandwidth utilization, applications and more. A

NetFlow capable router or switch collects IP flow data and sends it to a server where a flow collector is installed. The collector has the ability to decipher NetFlow packets and interpret their content in a user friendly manner for further traffic analysis.

sFlow [56] [45] [57] has a similar approach of using external modules/devices to collect and analyze traffic monitoring data. However, instead of aggregating packets into flows as NetFlow, the monitoring mechanism of sFlow is based on packet sampling (i.e., capturing packet samples with a ratio of either 1:1 to capture all packet samples, or 1:N to capture a packet sample for every N incoming packets). sFlow and NetFlow can collect all packet samples by setting a sampling ratio of 1:1, though 1:N ratio is usually applied for busy traffic due to a large number of flows to be collected. A sFlow system consists of multiple devices performing two types of sampling: random sampling of packets or application layer operations, and time-based sampling of counters. The sampled packet/operation and counter information, referred to as flow samples and counter samples respectively, are sent as sFlow datagrams to a sFlow collector, which is central server running software that analyzes and reports on network traffic.

NetFlow, sFlow and other similar approaches (i.e., IPFIX [11]) may not be integrated with the OpenFlow-based SDN platform. Furthermore, the agents running inside switches and packets/flows extracting processes consume a certain amount of computing/memory resources in the switch that may affect forwarding performance of switches [26]. Although NetFlow, sFlow as similar techniques like IPFIX may not be integrated and supported in some specific SDN switches, they can be considered as optional approaches for network traffic monitoring in SDN.

With the support of network programmability, SDN-based network monitoring and measurement solutions provides consistent traffic monitoring of flow parameters (e.g. bandwidth, packet loss, latency,...) to support various requirements of today's network services and applications. The flexibility of SDN-based traffic monitoring/measurement gives the network operators and managers the capability to offer dynamic QoS to ensure service quality between endpoints. In addition, SDN-based traffic monitoring/measurement enables a statistical way to infer characteristics that cannot be monitored in traditional large networks in some cases. Overall, SDN-based network traffic monitoring and measurement is considerably a critical factor to maintain the network controllability and manageability to meet the dynamic nature and various demands of network applications and services nowadays.

2.3 Traffic flow monitoring in SDN

2.3.1 Default traffic flow monitoring support in OpenFlow-based SDN

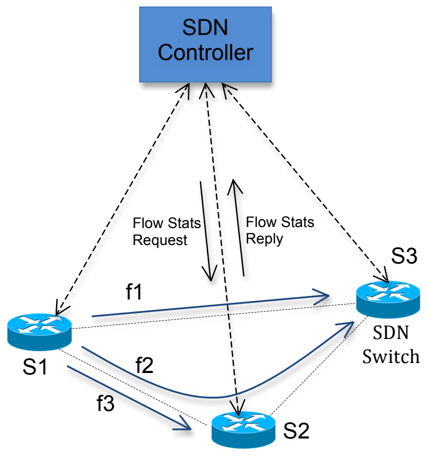
The main goal of traffic measurement in SDNs is to provide a flexible flow measurement with different granularities to satisfy a variety of applications. This task, however, is not trivial because it requires estimation of fine-grained volume of network flows with flexible settings in interconnected heterogeneous large scale networks. Currently, the OpenFlow based SDN uses counters in flow entries in switch's flow tables for monitoring functionality. Basically, current OpenFlow supports two kinds of traffic flow monitoring: individual flows monitoring and aggregate flow monitoring.

Individual flows monitoring

The individual flows monitoring is conducted by sending individual flow statistics requests (or individual flow descriptions requests) from the controller to switch to query statistics of individual flow entries at the flow tables of the switch [49]. This request is conducted by an OpenFlow-based message type called OFPMP_FLOW_STATS multipart request. Details on the format of this message can be referenced in [49].

Upon receiving an individual flow statistics request, the switch replies by sending a individual flow statistics reply to the controller. This reply contains the statistics of all individual flow entries that match with the match fields (i.e. ofp_match) defined in the request. The message body of the reply consists of an array of ofp_flow_stats (each corresponds to statistics of a matched flow entry). Details on the format of this message can be referenced in [49].

The statistics information of a flow entry basically consists of packet-count and byte-count of the flow, which are counted by the switch when packets are matched with the flow entry and are processed by the flow entry. By specifying the match fields, the controller applications can query the statistics of the flows that match the defined match fields. Fig. 2.4 illustrates an example of the flow statistics query process, in which the controller sends a flow stats request to the switch then get statistics of flow included in the flow stats reply sent from the switch. The details of the individual flows statistics request and reply can be referenced in the OpenFlow switch specification [49]. The individual flows monitoring is the fundamental method of SDN architecture for monitoring traffic flows in the SDN-based network.



Flow Table

Match fields	Counters	...
F2-1 <srcIp2, srcPort1, dstIp2, dstPort1, proto1>	...	...
F2-2 <srcIp2, srcPort1, dstIp2, dstPort1, proto2>	...	...
...		
F2-K <srcIp2, srcPortK, dstIp2, dstPortK, protoK>	...	...
F3-1 <srcIp3, srcPort1, dstIp3, dstPort1, proto1>	...	...
F3-2 <srcIp3, srcPort2, dstIp3, dstPort2, proto2>	...	...
...		
F3-M <srcIP3, srcPortM, dstIP3, dstPortM, protoM>	...	...
...		

Fig. 2.4 Basic mechanism for traffic flow monitoring in OpenFlow-enabled SDN

Aggregate flow monitoring

Besides the individual flows monitoring, SDN also supports aggregate flow monitoring. The aggregate flow monitoring is conducted by sending aggregate flow statistics requests from the controller to switch to query aggregate statistics of all flow entries at the flow tables in the switch that match the request [49]. This request is conducted by an OpenFlow-based message type called OFPMP_AGGREGATE_STATS multipart request. Similar to individual flows monitoring, once receiving an aggregate flow statistics request, the switch replies by sending an aggregate flow statistics reply to the controller (further details on the format of this message can be referenced in [49]).

The statistics information included in an aggregate flow statistics reply basically consists of a flow-count that is equal to the number of flow entries that match the request, and (optionally) the packet-count and byte-count that the switch counts all packets processed by all the flow entries matching the request. The details of the aggregate flow statistics request and reply can be referenced in the OpenFlow switch specification [49]. Basically, the individual flows monitoring is the typical method for monitoring traffic flows in SDN that is used in various network monitoring based controller applications and services.

2.3.2 Drawbacks of default traffic flow monitoring in OpenFlow-based SDN

Inflexibility since the monitoring is bounded with forwarding in the same flow tables. The current OpenFlow-based SDN architecture relies on flow entries in flow tables for monitoring task. Such dependence causes inflexibility since these tables should serve more priority the forwarding functionality. The inflexibility is due to the fact the packet header fields that controller applications want to apply for monitoring the network traffic may not be the same or overlap with OpenFlow match fields in flow entries that should be installed in flow tables to maintain usual forwarding functionality. Coarse-grained monitoring results in general flow entries will be inserted in the flow tables that affects the proper forwarding functionality of the switch (e.g. the existing finer-grained flow entries may be ignored or inactive due to the inserted general flow entries). While fine-grained monitoring results in a large number of flow entries will be inserted to the flow tables that slow down the forwarding functionality, which is basically based on matching with the flow entries in the flow tables.

Overhead in the control channel. With the current SDN architecture, every first packet of a new flow arriving at a switch will cause the switch to send a packet-in message to the controller to request forwarding rules to process the packet. Even if the packet is not the first one of the flow (it may be one of the following packets of the same flow that are ignored by the controller based on requirements of monitoring-based applications or because of an overload problem), the switch also sends a new packet-in message to the controller to request instructions to process the packet. Without any general flow entries installed in the switch (for basic forwarding functionality), the number of table-misses (when processing the matching with existing flow entries in the flow tables for a new flow) will be increased to a large number that causes a large number of packet-in messages sent from the switch to the controller (to query for a new flow entry to process that the new incoming flow). This large number of frequent queries from switch to the controller results in a lot of overhead created in the controller-switch communication channel and causes a large number of interrupts that heavily affects the performance/throughput of the switch.

Lack of scalability. Fine-grained monitoring demands a large number of flow entries installed in flow tables of the OpenFlow based SDN switch for the monitoring task. This large number of flow entries will cause the forwarding processing to be non-trivially slow down. Moreover, in fine-grained monitoring, large number of frequent packet-in messages will create a huge overhead in the control channel and interrupt the forwarding process of the switch that significantly decrease the throughput of the switch. In addition, the maximum number of flow entries that existing OpenFlow switches/routers can support is very limited (only a few thousands of flow entries depending on the specific switches/routers). With this limitation, the OpenFlow supported switches/routers cannot monitoring large number of traffic flows for fine-grained monitoring (since the current monitoring mechanism of OpenFlow totally relies on flow entries, while the number of flows in fine-grained monitoring may be much greater than the maximum number of flow entries allowed in OpenFlow supported switches/routers). These factors results in the lack of scalability for fine-grained monitoring in OpenFlow-based SDN.

Lack of support for monitoring over multiple switches. This results in the problem of duplication of flow monitoring in the switches. Since a flow may traverse through and be monitored at multiple switches in the network, multiple flow entries may be created at those switches to monitor that single flow. While in practical, only a single monitoring/flow entry can be enough to monitor a single flow. This duplication may create significant overhead on switches's computing and memory resources, computing/memory resources of the controller and the overhead on the controller-switch communication channel, to handle the duplicated monitoring rules. Moreover, current SDN architecture almost has no support to distribute and balance the monitoring load over multiple switches in the network. This results in instability/unbalance of the network, since heavy monitoring loads may be assigned to busy switches while idle switches handle only lightweight monitoring loads.

With the above mentioned drawbacks, the current SDN monitoring support is considerably not flexible, scalable, and adaptable enough to support fine-grained and network-wide monitoring tasks for a variety of controller applications.

2.4 A survey on requirements of different networks and applications

This section discusses a survey on the requirements of different networks and applications which are required to evaluate if a monitoring system is capable of serving the networks or applications. The requirements on the networks or applications are based on the main metric of number of active flows in a certain time window. Number of active flows is the main and direct factor that affect the performance of a monitoring system. A monitoring system is considered to be capable of serving a network or an application if the number of active flows supported by that system in a certain time window is over the number of active flows that may exist in the network or the application.

2.4.1 Investigation on the requirements of different networks

For the investigation about requirements of networks, we target three types of networks, i.e., local area network, campus network, and backbone network, as they are typical networks for SDN deployment. The investigation considers the number of active flows in certain time windows, which is the main factors to judge whether a network monitoring system is capable of serving the networks. The referenced information is gathered and summarized in table 2.1.

Network ID	Network type	Number of active flows	Time window
Ab-I	Backbone	37000	20 (s)
Ab-III	Backbone	62000	20 (s)
ADSL	LAN	24000	20 (s)
INRIA	Campus	38000	60 (s)
MAG	Backbone	100105	5 (s)
MAG+	Backbone	98424	5 (s)
COS	Campus	18070	5 (s)
IND	Campus	2164	5 (s)
Enterprise	Enterprise	131281	1 hour

Table 2.1 A survey on number of active flows in different networks

In the above table, the MAG, MAG+ are the traffics that were measured in a high speed backbone OC-48 backbone link, and the COS, IND are the traffic that were measured on connection points of two university campus to the internet [17]. The INRIA is the traffic measured in connection point of a campus (INRIA) to the internet [22]. The ADSL is the traffic measured on an OC3 link to several thousands of users [34]. Ab-I is the traffic measured on an OC48 link on the Abilene research network between Indianapolis and Kansas City of USA, and Ab-III is the traffic measured on an OC192 link on Abilene II between Indianapolis and Chicago [34]. Another example is the network traffic measured in an enterprise network (i.e., IBM Zurich Research Laboratory) [14]. The traffic is measured at two border routers of the network where it shows 131,281 active flows in a duration of 1 hour.

These references indicate the number of active flows that may exist on different networks (i.e., local area network, campus network, backbone network, and enterprise network) in a certain time window. These are examples to show how busy the networks are for estimating the applicability of a network traffic monitoring system. As for our proposal, the experimental results in chapter 3 shows that the monitoring platform can serve networks with over hundreds of active flows concurrently while the the distributed monitoring process at controller only spends 0.3-1.6 (s) for processing from two hundreds thousands to a million active flows. This indicates that the proposed platform is capable of serving the above mentioned types of networks.

2.4.2 Investigation on the requirements of different network applications

To study the requirements of common network applications, we investigate a few typical network applications that requires traffic monitoring for its operation, such as heavy-hitter flows detection, network troubleshooting, network forensic, bandwidth monitoring, and anomaly detection. These applications and their necessity for network operation and end-users, as well as required monitoring statistics are summarized as follows.

Heavy-hitter detection: Heavy-hitter detection is to detect traffic flows that have a large number of packets or to find the set of flows contributing significant amounts of traffic to a link. Heavy-hitter detection benefits a number of network management applications. For instance, heavy-hitter detection enables relieving link congestion [4], planning network capacity [19], or to cache forwarding table entries [62]. Further, identifying heavy hitters at small time scales can enable dynamic routing of heavy flows [12] and dynamic flow scheduling [66]. In order to detect heavy-hitters, network monitoring is important as it provides the required information of flows statistics for the analysis and detection phase. Heavy-hitter detection typically requires flow statistics (i.e., packet counts, and byte counts) based on 5 tuple (i.e., source/destination IP addresses, source/destination port numbers, protocol number) can be used to identify the heavy-hitters [46].

Bandwidth monitoring: Bandwidth monitoring is essential for planning network resources and assuring quality of services. It can be applied to detect the sources of bandwidth consumption by monitoring traffic volume of flows (i.e., numbers of packets and accumulated byte counts of the flows defined with 5-tuple). Monitoring bandwidth in fine granularity brings a global view in terms bandwidth consumption of hosts, applications and services to network operators so that they can ensure proper bandwidth consumptions of applications and services (e.g., within the limits they are allocated). Network operators can ensure bandwidth availability and resource utilization in a reasonable manner and estimate the network equipment upgrades needed to satisfy bandwidth requirements of applications and services.

Network troubleshooting: Troubleshooting and discovering root causes of some of the hard-to-debug network issues can be done with flow-based network monitoring. For instance, by correlating packet counts for a flow across the network hop-by-hop, it is able to identify the node that is dropping the packets.

Network Forensics: The capability to define flexible criterion for monitoring flows makes it easier to gather network forensic reports. Typically, 5-tuples are used to define criterion for capturing flow metadata that can define wildcards for some of the tuples. This enables gathering and analyzing reports (e.g., for a particular source and destination IP address combination).

Flow based anomaly detection and mitigation: Anomaly detection and mitigation is an important network security application to ensure proper operation of networked systems. To detect an anomaly, packet inspection approach can be applied where capturing and inspecting the payload of every packet, which is high cost and may be hard to perform in high speed networks. Therefore, flow based approach has been introduced as a supplement solution besides packet inspection approach [69]. Flow based anomaly detection analyzes communication patterns of network instead of contents of individual packets, thus it can be applied at a first stage for quick detection of anomalous traffic flows, then packet inspection may be used for further analysis of detected anomalies. Typically, for detecting anomalous flows (e.g., Denial of Service, Scans, Worms, Botnets), traffic volume statistics of flows at a fine-grained granularity (i.e., 5 tuple) are required [69]. The traffic volume is represented by number of bytes or number of packet in flows, and this is the data source for further anomaly analysis. For instance, an anomaly detector may set an empirical threshold for traffic volume changes in flows through query times, then when the traffic volume change of a flow exceeds the threshold, the flow is marked as anomalous [65]. Reaction to anomalous/attacked traffic to mitigate the affect of attacks requires a control capability that can drop, reroute or modify traffic pattern quickly or even in real time. Thus, a fine-grained traffic monitoring platform with low overhead that can be integrated to a capable network control architecture is required for anomaly detection and mitigation applications.

Since the main factors to evaluate whether a network monitoring system is applicable for a network application are the number of active flows and corresponding time windows where the flows are generated in the network, we consider these factors in this investigation. The referenced information is gathered and summarized in table 2.2.

Application	Number of active flows	Time window
Heavy-hitter flows identification [46]	206,299	124 seconds
Heavy-hitters detection [14]	133,281	1 hour
Diagnostic of troubleshooting [73] (FREEnet 1)	64778	30 min
Diagnostic of troubleshooting [73] (FREEnet 2)	15495	30 min
Anomaly detection ([41] [71])	54,912	15 seconds
Anomaly detection ([76])	324,608	15 seconds

Table 2.2 A survey on number of active flows in different applications

In Table 2.2, the heavy-hitter flows identification application [46] detects heavy-hitter flows in a high speed backbone network. The traffic is measured in a high speed OC-48c backbone link by the PMA project at NLANR where it shows 206,299 flows (i.e., 5-tuple flows) in a duration of 124 seconds of the detection. As another reference, the heavy-hitter detection application [14] detects heavy-hitter flows from the network of the IBM Zurich Research Laboratory. The network hosts approximately 400 employees and at least as many networked workstations. The network traffic is measured at the two border routers of the network where it shows 133,281 flows in a period of 1 hour. The diagnostic of troubleshooting application [73] works on a backbone network. Traffic monitoring were conducted on two border gateway routers from FREEnet (i.e., FREEnet 1, FREEnet 2), which has several internal and external links. Measurements from Gigabit links were taken, where network statistics were recorded at 30 minutes intervals, twenty-four hours a day for a week to discover network behavior with different loading levels. The maximum number of active flows observed through the measurement period is 64,778 and 15495 for the border gateway routers FREEnet 1 and FREEnet 2 with a time window of 30 minutes, with traffic rate of 215.4 Mbps and 28.05 Mbps correspondingly.

For anomaly detection and mitigation application, the accumulated number of flows that an anomaly detection requires to monitor can reach a number of millions. For instance, the number of flows in a duration of 15 minutes of MAWI traffic repository [41] (i.e., a traffic trace collection measured at a transit link) is 3,299,166 flows [71]. Similar accumulated number of flows for CAIDA [76] (i.e., another well-known traffic trace) is 2,353,413 flows, and RedIRIS [71] (i.e., a Spanish Research and Education network) is 2,972,880 flows. As these numbers are the accumulated number of flows during the monitoring lifetime, active number of flows required to be monitored is a lower number corresponding to query time window. For instance, the number of new flows arrived per second in those traces are 3,665 flows in MAWI, 21,672 flows in CAIDA, and 9,916 flows in RedIRIS. Furthermore, average number of required monitoring rules with a timeout of 15 seconds (i.e., number of active flows for a time window of 15 seconds) for a flow sampling ratio of 1/128 is 429 active flows for MAWI, 1,162 active flows for RedIRIS, and 2,540 active flows for CAIDA. Thus, the number of active flows in a time window of 15 seconds is about 54,912 active flows for MAWI, 148,736 active flows for RedIRIS, and 324,608 active flows for CAIDA. Those are the numbers of active flows in an example time window of 15 seconds in the above traces. A certain sampling ratio may be applied in case of monitoring in busy network, so as to reduce the number of active flows to be monitored in a certain time window. For a smaller time window, the number of active flows requires to be monitored will be smaller with a possibly corresponding ratio.

The above mentioned applications typically require 5-tuple flow statistics for their analysis and operation. As the number of active flows in a certain time window is the main factor that affects the performance of a network monitoring system, a network monitoring system is considered as being capable of serving such applications if it can handle monitoring active flows in the applied networks of the applications. Therefore, the condition to judge the applicability of a network monitoring system for such applications is whether it is capable of monitoring specific numbers of active flows that may exist in the target deploying networks of the applications (e.g., LAN, Campus network, Backbone network, etc).

2.5 Existing approaches for SDN monitoring

Realizing the importance of network traffic monitoring for traffic engineering and network management, recent researches exploit SDN and proposed approaches for enhancing traffic monitoring in SDN. The works proposed methods for in-band traffic monitoring SDN using OpenFlow switches and controller. The main goal of the approaches is to reduce monitoring overhead either for switches, controller, and the network. The existing works can be categorized into 4 main approaches: overhead per monitoring rule reduction approach, sampling based approach, switch-selection based approach, and rule aggregation based approach. The details of the approaches are discussed as follows.

2.5.1 Switch overhead reduction approach

This approach refers to the works that proposed monitoring methods for OpenFlow based SDN switch (i.e., Open vSwitch) to reduce monitoring overhead. Realizing the important of fine-grained traffic monitoring for network management, and the resource limitation in a switch, the authors in [81] proposes UMON that can support fine-grained monitoring of traffic flows at switch. UMON extends OpenVSwitch and embed additional monitoring module to support traffic monitoring based on non-routing fields and subflow monitoring. It restructures a flow table in the OpenFlow pipelined flow tables into a monitoring table with an extra pointer pointing to a subflow table that can monitor subflows (Fig. 2.5). The monitoring mechanism is that after a packet passes through the pipelined flow tables, it is passed through the monitoring table where flow statistics will be collected. Packets of a flow can also be passed to the subflow table (if the extra pointer to the subflow table was specified in the entry that monitors that flow in the monitoring table) where finer-grained statistics will be recorded based on its specified subflows. By this way, a flow can be divided into subflows and monitored with finer-grained statistics. In the kernel space of OpenVSwitch, in order to collect flow statistics, the authors suggest to adjust kernel flow rule into a finer one by mapping kernel forwarding rules to a monitoring rule. For instance, assume there are two existing forwarding rules DstIP=A, Output(0) and DstIP=B, Output(1), and we would like to monitor packet counts that are sent to port 80. This can be achieved by unwildcarding the DstPort field and converting the original two forwarding rules into following rules (assuming that all packets are either to port 80 or port 22): DstIP=A, DstPort=80, Output(0) (rule 1), DstIP=A, DstPort=22, Output(0) (rule 2), DstIP=B, DstPort=80, Output(1) (rule 3), DstIP=B, DstPort=22, Output(1) (rule 4) [81]. From those converted rules, the packet counts that are sent to port 80 can be inferred as the sum of packet counts of rule 1 and rule 3. Using kernel space of OpenVSwitch may reduce the monitoring overhead in user space (i.e., reduce overhead per monitoring rule), however, converting forwarding rules into finer rules for monitoring is complicated as it depends on the overlap of match fields in forwarding rules to infer them into sub-rules, and it may increase the number of rules in the kernel routing tables than necessary. Although the approach of UMON supports fine-grained monitoring, the current design mostly fit for only Open vSwitch with kernel and user spaces. In addition, the performance of UMON as observed is still limited (e.g., throughput of UMON is about 2 - 2.4 Gbps, and the packet rate is about 0.3 - 0.4 Mpps for a 10G NICs experimental network).

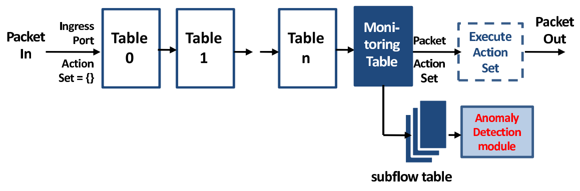


Fig. 2.5 UMON design which restructures a flow table and subflow table for monitoring [81]

Similarly, another work [29] proposes extension for Open vSwitch that injects telemetry functionality for network monitoring. The proposal extends kernel module of Open vSwitch that modifies port functionality in the switch, and manually insert additional actions to monitor some traffic metrics. The mechanism is that at the ingress port, some telemetry metadata is allocated and associated with incoming packets, and an ingress timestamp is stored in reserved memory of the packets. At the egress port, packets with attached telemetry flags and ingress timestamp is processed where telemetry metrics are computed. With this mechanism, switch performance is not affected by the extension when not using the telemetry functionality and overhead is reduced by implementing the monitoring process in kernel space of Open vSwitch. However, the current implementation of the proposal mostly focus on hop latency metric where other important metrics for traffic monitoring such as number of packets/bytes per second are not implemented, leaving the default monitoring mechanism in the OpenFlow switch to handle monitoring these traffic metrics. Thus, the current design may not benefit for fine-grained traffic engineering applications.

2.5.2 Sampling based approach

This approach indicates the works that deploy external system using tools such as sFlow [57], NetFlow [10], IPFIX [11], or JFlow [47], to collect packet samples for monitoring purpose. The sampling based schemes in this approach are considered as non-OpenFlow monitoring schemes as they mostly requires external deployment for sFlow/NetFlow/IPFIX system for sampling and analyzing packets/flows. A typical scheme in this approach is OpenSample [72], which uses sFlow deployed in external hardware servers to collect flow samples for monitoring. OpenSample uses sFlow agents running on switches to collect packet samples and transmit them to a sFlow collector. The samples are then aggregated into snapshots and sent to the controller via a communication API (Fig. 2.6). The external collector in OpenSample gathers samples from all switches in the network for monitoring and analysis.

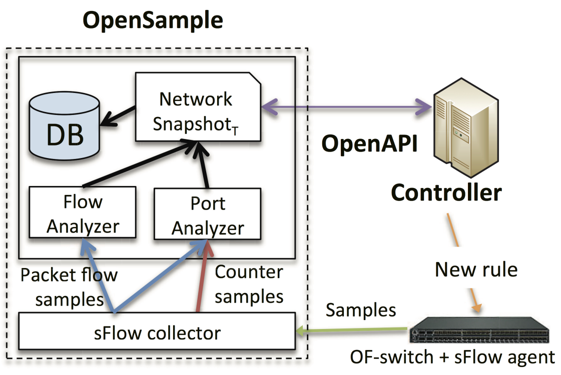


Fig. 2.6 OpenSample monitoring scheme [72]

Similarly, another work [25] proposed combining OpenFlow and sFlow for anomaly detection in SDN. The proposal deploys a similar network statistics collection system that uses sFlow tool for collecting packet samples from the SDN network, then use the statistics as the input for further analysis and detecting anomalies. By sampling packets and delegating further monitoring processes to an external system, the works in [72] and [25] introduce low overhead at switch and low monitoring latency. Similar to OpenSample, this proposal requires external deployment for sFlow collector, and monitoring APIs are not integrated well to OpenFlow control APIs, which may not benefit for facilitating implemention of general traffic monitoring applications.

2.5.3 Rule aggregation based approach

The rule aggregation based approach infers to the works that reduce monitoring overhead in switches and in the network by suggesting flow monitoring algorithms that can adaptively adjust the granularity of flow monitoring. The authors in [88] proposed a monitoring mechanism, i.e., OpenWatch, that can adjust aggregation of flows for anomaly detection. The mechanism bases on an algorithm to predict the possibility of anomaly to adjust the monitoring granularity. For instance, if the predicted possibility of anomaly is low, the algorithm can aggregate flows to reduce total number of required rules in switch. By adaptively aggregate flows, the mechanism balances between monitoring overhead and accuracy detection and reduce monitoring overhead when the predicted possibility of anomaly is low. However, as the scheme bases on aggregating flows, it may not be applicable for applications that requires fine-grained statistics of flows (e.g., traffic classification application, where fine-grained statistics is required to identify traffic of different applications or groups of applications). In addition, as the scheme uses forwarding flow entries for monitoring, it may not be flexible for other applications as matching rules for monitoring purpose may be different from forwarding purpose (e.g., traffic classification may requires fine-grained monitoring rules with 5-tuple based matching, while basic forwarding may require much coarser grained rules such as only ingress and egress port numbers). Moreover, the overhead per rule is not reduced in the proposed scheme, which can be considered as a drawback for fine-grained monitoring as fine-grained monitoring may require a large number of rules installed in switch.

2.5.4 Switch selection based approach

The switch selection based approach represents for the works that reduce monitoring overhead in the network by selecting switches in the network for polling flow statistics. A typical scheme in this approach is FlowCover [70], which proposed a switch selection scheme for polling flow statistics (Fig. 2.7. FlowCover selects a subset of switches that cover all existing flows (i.e., existing flow entries in switches) in the SDN network and only poll the statistics from only these switches instead of polling all switches. By this scheme, statistics of all existing flow entries can be gathered at the controller with smaller number of statistics queries is required, as the controller only polls statistics from selected switches in each query time. Thus, the communication cost in the SDN control channel can be reduced accordingly. However, the overhead per monitoring rule is not reduced in this scheme where default forwarding flow entries are used for traffic monitoring, which is inflexible and can be considered as a drawback to be applicable for applications that require fine-grained monitoring (e.g., traffic classification, which may require 5-tuple based monitoring), as fine-grained monitoring may require installing large number of rules in switches. In addition, as switches independently monitor flows where the scheme do not present any mechanism to detect and eliminate rule duplication, redundant rules may still remain in switches which cause non-trivial overhead for switches.

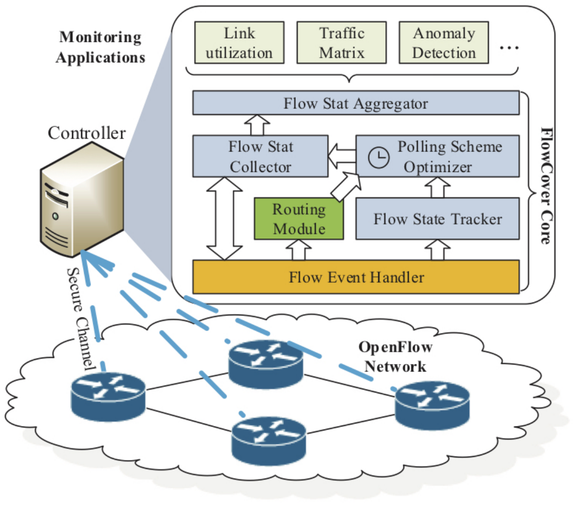


Fig. 2.7 FlowCover monitoring scheme [70]

The authors in [77] introduces OpenTM, a switch selection scheme for traffic matrix estimation (i.e., estimation of traffic volume between origin-destination pairs in a network). The idea is that for each flow, the mechanism selects switches in the flow path to poll flow statistics. It keeps track of all active flows in the network, get topology information from the controller, discover flow paths, then periodically polls flow statistics from switches on the flow path. The switch selection strategies for statistics query are also discussed, e.g., querying the last switch, querying the switches on the flow path uniformly or non-uniformly random, round-robin querying, querying the least load switch, for traffic matrix estimation. The switch selection reduces overhead in the communication channel as the controller can query only selected switches to get flow statistics instead of querying all switches. As the proposal does not discuss neither reducing overhead per monitoring rule nor removing redundant rules in switches in the switch selection mechanism, if the number of active flows is large, the overhead in switches and the communication cost become large accordingly.

2.5.5 Time window based approach

The time window based approach indicates the proposals that adjust the frequency and schedule of polling flow statistics to reduce monitoring overhead in the network. The authors in [9] proposed Payless, an adaptive scheduling mechanism for flow statistics collection which targets a link utilization application. The key idea of the proposed mechanism is that it maintains a higher polling frequency for flows that significantly contribute to link utilization, and a lower polling frequency for flows that do not significantly contribute to link utilization at a certain moment. Payless uses FlowRemoved and FlowStatisticsRequest of OpenFlow protocol, and initializes a statistics collection time out τ (i.e., polling time window) to collect flow statistics. Specifically, when the controller receives a PacketIn message from a switch, it will add a new flow entry to the switch with flow expiration time out is set as the Payless suggested time window τ. If the flow is expired within τ, the controller will receive statistics of the flow in a FlowRemoved message. Otherwise, the controller sends a FlowStatisticsRequest message to poll statistics of the flow. If the packet count in that flow does not significantly change within τ, Payless will multiply the τ for that flow with a small constant α, and in the opposite case, the τ will be divided by another small constant β. By this mechanism, low polling frequency can be applied for flows with low traffic volume, thus it reduces communication overhead on the OpenFlow control channel. This proposal mainly balances between overhead and monitoring accuracy for link utilization application. As the proposal does not discuss reducing overhead per monitoring rule, it may not suit applications that require fine-grained monitoring of a possibly large number of traffic flows.

Other works propose monitoring methods that target specific scenarios/applications. The authors in [87] proposed a mechanism (i.e., FlowSense) that exploits OpenFlow based events to get flow statistics for network utilization. The mechanism acquires flow statistics by passively capturing and analyzing the flow arrival and flow expiration messages. The proposed mechanism may introduce low overhead for traffic matrix estimation as it mostly captures OpenFlow based events/messages for the estimation. However, it calculates the link utilization at discrete points in time after the flows expire, thus, it may not be applicable for applications that require real time monitoring statistics (e.g., traffic classification). OpenNetMon [78] proposes a similar mechanism for monitoring specific per-flow metrics such as throughput, delay, and packet loss. The idea of the mechanism is poll switches with an adaptive rate (i.e., polling time window) that increases when flow rates differ between samples (i.e., traffic volume change significantly between the current and nearest queries) and increases when flows stabilized. By using adaptive polling rate, network overhead can be reduced as low polling rate can be applied when traffic volume change in a flow is insignificant, and it is basically a tradeoff between monitoring accuracy and network overhead. However, the mechanism simply polls edge switches in each flow path without any switch selection mechanism, thus the monitoring load in switches and in the network may not be balanced as there is no mechanism to control it. In addition, as the proposal does not discuss reducing overhead per monitoring rule, the monitoring overhead is large when the number of active flows in the network is large (i.e., it may not suit for applications that require fine-grained

Method	Features	Overhead per rule reduction	Distributed monitoring overhead reduction	Network component	Aggregated flow monitoring	Pros and cons
UMON [81]	Inject monitoring module into switch for monitoring	Yes	No	Switch (Open Vswitch)	Yes	Reduced overhead per rule
OpenSample [72]	Deploy sFlow system to collect packet samples for monitoring (sampling based)	No	No	Use external deployment	Yes	Reduced overhead by sampling. High accuracy, low latency. May require external deployment for sFlow collector
K. Giotis et.al [25]	Deploy sFlow for anomaly detection (sampling based)	No	No	External deployment	Yes	Reduced overhead by sampling. High monitoring accuracy, low latency. Required external hardware deployment for sFlow collector
OpenWatch [88]	Prediction for flow counting to detect anomaly (rule aggregation based)	No	Yes	Controller	Yes	Overhead of switch and network reduced by rule aggregation. Overhead per rule not reduced
FlowCover [70]	Switch selection based, load-balanced	No	Yes	Controller	No	Overhead of switch and network reduced by distributing load over switches. Overhead per rule not reduced
OpenTM [77]	Select switches on flow path for polling statistics (switch selection based). For traffic matrix estimation	No	Yes	Controller	No	Overhead on controller and network reduced. Overhead in switch not reduced (redundant rules in switch not removed)
Payless [9]	Adjust frequency of polling statistics in switch (time window based)	No	Yes	Controller	No	Reduced network overhead (tradeoff between monitoring accuracy and overhead). Overhead per rule not reduced
FlowSense [87]	Capture flow arrival and flow expiration messages for monitoring	No	Yes	Controller	No	Reduced overhead by only capturing OpenFlow events. Not capable of real-time monitoring
OpenNetMon [78]	Adapt statistics polling time window to reduce overhead (higher time window for busy traffic and lower one for less busy traffic). Simple polling edge switches in each flow path	No	Yes	Controller	No	Reduced overhead for less busy traffic. Monitoring load in switches not balanced. Overhead per rule not reduced

Table 2.3 Pros and cons of existing SDN traffic monitoring methods

monitoring of traffic flows). The main features, advantages, and disadvantage of the related works are summarized in table 2.3.

Chapter 3

Monitoring method for SDN switch

This chapter presents our monitoring method for OpenFlow-based SDN that reduces monitoring overhead at switch and the network for fine-grained monitoring of network traffic. We aims at designing a dedicated method for monitoring that can reduce overhead per monitoring rule in switch. As outlined in chapter 1, using flow entries for monitoring is expensive and may introduce high overhead, especially when the number of active flows in the network is large, we design our proposed monitoring method with monitoring components and process that are independent from forwarding functionality of OpenFlow based SDN switch. In this chapter, we firstly present in section 3.1 an empirical discussion on requirements of different networks and applications. Section 3.2 presents an overview about our method. In section 3.3, we describe the details about the proposed method including the proposed architecture, monitoring process, monitoring APIs. The implementation of the proposal are described in section 3.4. Section 3.5 describes the evaluation of our proposed method in terms of monitoring overhead at switch and other aspects to validate the efficiency of the proposal. In section 3.6, we discuss the monitoring performance of Lagopus switch and our proposed method, the impact of sampling on the monitoring overhead in the proposal, and a comparison of our proposed method with existing works. A summary of the chapter is presented in section 3.7.

3.1 Design requirements

In OpenFlow-based SDN, a main controller typically manages a number of switches in the network. Therefore, in the proposal of this chapter, we assume to use a controller to control monitoring switch. As discussed in chapter 2, we target our proposal to serve monitoring for common networks (i.e., LAN, backbone network, campus network, and smaller scaled networks) and traffic engineering applications (e.g., heavy-hitter detection, traffic classification, bandwidth monitoring, routing optimization). As these networks and applications may require monitoring fine-grained statistics of thousands or a hundred thousands of active flows, we set a first requirement for our proposal in this chapter is being capable of monitoring thousands or a hundred thousands of active flows. Furthermore, discussed in chapter 2, the targeted networks and applications require query time windows of seconds, minutes, or an hour. Therefore, we set a second requirement for our proposal is that the processing time (i.e., the elapsed time of the monitoring process) of the proposed monitoring method must be under a threshold of required query time window, i.e., in an order of seconds. We target our design for fine-grained and real-time monitoring that does not rely on capturing OpenFlow events passively (e.g., existing works of time window based approach that base on OpenFlow Packet-In and Flow-Expiration messages to monitor traffic, such as FlowSense [87], OpenNetMon [78]), thus we reasonably do not require setting short flow expiration period or frequently change of routing flow entries at switch in our design.

Furthermore, as discussed in chapter 1, the target networks and applications may require monitoring a large number of active flows (e.g., thousands or a hundred thousands active flows), while a switch typically has limited resource (e.g., bandwidth, CPU, memory). Therefore, to meet the above mentioned requirements, a monitoring method must be able to reduce monitoring overhead at switch to enable fine-grained monitoring capability for switch and the network. For monitoring a large number of active flows, the required number of monitoring rules is large accordingly, which results in a large monitoring overhead in switch. Therefore, a monitoring method must be able to reduce monitoring overhead per monitoring rule in switch to ensure a low/acceptable monitoring overhead for serving such networks and applications. This is considered as a critical requirement for proposing a monitoring method in OpenFlow-based SDN. In summary, main requirements for our proposed monitoring method in this chapter consists of: (1) The method must be able to reduce overhead per monitoring rule in switch; (2) The method must be capable of monitoring thousands or a hundred of active flows; (3) The processing time of the monitoring method must be under a threshold of a few seconds for monitoring such number of active flows in the network.

3.2 Method overview

In order to meet the above mentioned requirements, we propose a monitoring method for OpenFlow based SDN where switch can monitor traffic flows with reduced overhead. Our strategies for reducing overhead per monitoring rule in OpenFlow-based SDN switch and network consists of:

(1) Designing lightweight monitoring modules and monitoring process in OpenFlow-based SDN switch which is independent from the forwarding functionality of the switch. In our design, incoming packets are forwarded by matching with the flow entries of the switch, while fine-grained statistics of the flows, i.e., packet counts and byte counts, are recorded or updated using separated monitoring rules (i.e., monitoring entries). With this design, incoming packets are forwarded with no additional delay and the matching is with a small number of flow entries (which are installed for basic packet forwarding functionality of the switch), while the monitoring process record statistics automatically and concurrently. This design strategy conserves the throughput for the switch.

(2) Monitoring entry in the proposed method is designed with smaller number of fields than a flow entry, and a fast lookup/matching data structure is also applied (i.e., hash table with hash based matching, 5 tuple exact match) to reduce monitoring overhead. Number of fields and number of required bytes in each monitoring entry is minimized with necessary fields and bytes that dedicated only for flows monitoring purpose.

(3) Designing a sampling process to support a flow-based sampling capability. For this sampling process, a lightweight data structure (i.e., Bloom filter) is used for checking if a flow is sampled or not. This sampling method support reducing number of monitoring rules with a tradeoff with monitoring accuracy. Thus, it may enhances throughput in the switch (i.e., decrease the monitoring overhead) in case the switch is about to be overloaded.

We design monitoring modules and process integrated as a monitoring platform for OpenFlow-based SDN called SDN-MON. The platform is integrated into OpenFlow protocol to improve monitoring capability for network management. Our monitoring method can be simplified with 2 phases:

- When a packet traverse through an OpenFlow switch, the switch forwards the packet using flow entries in its flow tables (it may sends a PacketIn message to ask for a flow entry if the incoming packet is a first packet of a new flow). Concurrently, SDN-MON monitoring module in switch monitor the flow with fine-grained statistics using monitoring entry or Bloom filter element.

- Applications at the controller use SDN-MON monitoring APIs to query SDN-MON monitoring statistics in switch with an adjustable query time window for their operations.

The details of the architectural design, structures of monitoring modules, and the monitoring process are described in the following sections.

3.3 SDN-MON Architecture

Our SDN based monitoring platform (SDN-MON) is a platform that supports a fine-grained and flexible monitoring for controller applications. In SDN-MON, monitoring is conducted independently from forwarding. SDN-MON monitors packet flows at fine-grained level (i.e., based on 5 tuple match fields) using a separated and integrated monitoring module in switch, while the flows can be forwarded by a small number of coarse-grained flow entries in flow tables. SDN-MON supports controller applications to monitor flows based on 5 tuple match fields. These are common match fields of various flow based control applications. The 5 tuple match fields consist of source Ip address, source port number, destination Ip address, destination port number, and protocol number. SDN-MON supports an efficient sampling mechanism that makes it more scalable to be adaptable with larger-scale networks than current mechanisms. With SDN-MON support, the SDN controller can determine forwarding logics in a more proper and scalable way with less flow entries in the switches' flow tables to avoid a possible overflow problem. Moreover, the number of controller-switch messages required for monitoring are smaller, this helps reduce the monitoring-based overhead in the SDN communication channel. SDN-MON leverages the switches' processing power to process its monitoring functionality and is designed with a high priority for low processing overhead in a switch. In this section, we describe the architecture of SDN-MON, how it operates alongside other processing of SDN switches as well as its APIs to support monitoring applications on SDN.

3.3.1 Architecture Overview

As shown in Fig. 3.1, SDN-MON is composed of a controller-side module and a switch-side module. The controller-side module has SDN-MON monitoring APIs that enable a flexible monitoring in the SDN controller. These APIs operate on top of the SDN controller platform to support monitoring purposes of controller applications. The switch-side module consists of three components: the SDN-MON local control app, SDN-MON monitoring APIs, SDN-MON monitoring database. These components operate together to handle the monitoring functionality in switches. The support of these components together with the SDN-MON monitoring APIs on the controller-side enables a more fine grained and flexible monitoring with a set of arbitrary monitoring match fields defined by controller applications.

SDN-MON Monitoring APIs at controller side

The controller-side SDN-MON monitoring APIs are programming APIs to support controller applications for monitoring. With these APIs, controller applications can request the switches

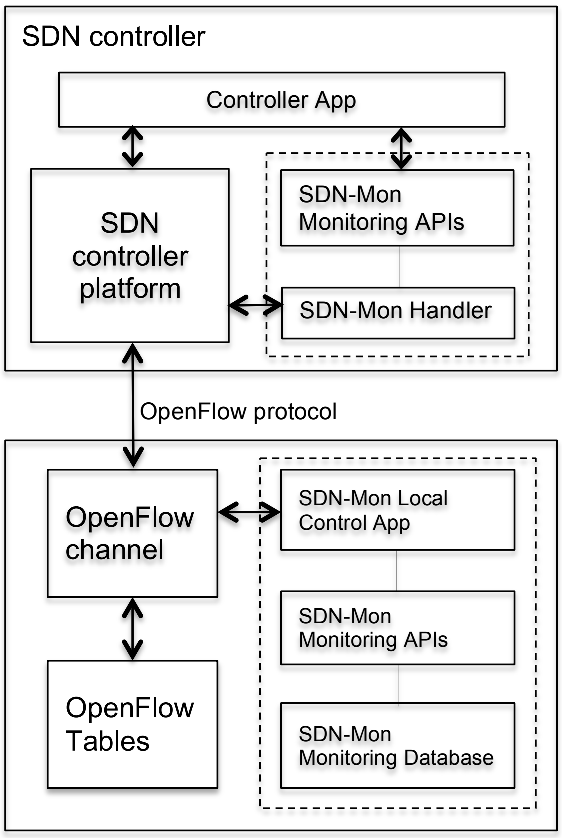


Fig. 3.1 SDN-MON Architecture

to insert monitoring entries with a certain user-defined set of monitoring match fields, remove monitoring entries from the monitoring database in switches, query network statistics of monitoring entries in switches, as well as other monitoring control functions, as described in Section 2.3. These APIs serve a variety of monitoring purposes of controller applications independently from the forwarding functionality.

SDN-MON Handler

The SDN-MON handler is a module at the controller-side that is responsible for handling SDN-MON-related communication with switches. This module supports encapsulating the controller's monitoring request parameters into SDN-MON messages and sending the messages to the switches. For SDN-MON-related replies from the switches, the SDN-MON handler supports parsing the received messages and extracting the monitoring data or switch notification information from the messages for controller applications.

SDN-MON Local Control Application

The SDN-MON local control app is a lightweight processing application that is responsible for handling SDN-MON-related requests from the controller as well as local switch-management processing to ensure the proper operation of the SDN-MON switch-side module. This application analyzes the monitoring-related messages delegated by the SDN communication channel between the control and data planes, which is enabled by the well defined OpenFlow protocol [49], and leverages the SDN-MON monitoring APIs to process and manage the monitoring database based on the controller requests. For network-statistics requests from the controller, the SDN-MON local control app queries the statistics from the monitoring database, encapsulates the statistical information, and delegates it to the controller-switch communication channel to send to the controller. The SDN-MON local control app also reserves room for local pre-processing in switches for experimenters' applications (e.g. pre-checking flows in a switch for anomaly detection, and pre-checking flow volumes to detect large flows in the networks).

SDN-MON Monitoring APIs at switch

The switch-side SDN-MON monitoring APIs are programming APIs for the SDN-MON local control app to process the monitoring tasks delegated by the controller through the monitoring requests. Basically, the programming functions provided by these APIs correspond to the functions of the controller-side SDN-MON monitoring APIs. These APIs support the SDN-MON local control app to manage the SDN-MON monitoring database and query monitoring statistics from the database to respond to the controller requests. Details of these APIs are described in Section 2.3.

SDN-MON Monitoring Database

The SDN-MON Monitoring Database includes a monitoring table and a Bloom filter. The monitoring table has a set of monitoring entries. Each monitoring entry consists of monitoring match fields (i.e., 5 tuple), counters, and a hash value. The structure of a monitoring entry consists of the following fields: hash value of the entry (used as an identification of the entry that hashed based on the 5 tuple), source Ip address, source port number, destination Ip address, destination port number, protocol number, packet count, byte count, and last update time stamp.

Since OpenFlow protocol [49] is basically the only protocol that enables SDN communication channel between the control and data planes, in our current design, monitoring match fields are a set of fields that correspond to the match fields of a flow entry defined in the OpenFlow protocol. These fields consist of a subset or all OpenFlow match fields. Controller applications define monitoring match fields based on their monitoring purposes at the initial stage of the monitoring process. Counters consist of a list of packet counts and one of byte counts. These lists have an equal size (which is the number of packet count in the lists, called counters buffer size S_b) that is defined by controller applications at the initial stage of the monitoring process based on the applications' requirements. The packet and byte count lists are buffers that record the historical packet and byte counts of a monitoring entry at every time interval ΔT_u. The counter-buffer-update time interval ΔT_u is calculated by the local control application based on S_b and a query time interval ΔT_q, which is also set by the controller at the initial stage of monitoring, with the formula: $\Delta T_u = \Delta T_q / S_b$. These counter buffers support fine-grained monitoring and help decrease the number of necessary data exchanges between switches and controller for monitoring statistics. The hash value in a monitoring entry is calculated from the monitoring match fields of the entry. This hash value together with the hash table data structure implementation of the monitoring table support accelerating the lookup process in the monitoring table.

Due to the need of sampling in a variety of existing network monitoring applications, SDN-MON provides a sampling mechanism in its monitoring process. This sampling mechanism is processed with the support of a Bloom filter [6], a lightweight data structure for checking the existence of certain entries. In SDN-MON, the Bloom filter is used to mark specific flows corresponding to an incoming packet as non-monitoring to ignore them for monitoring based on the result of the sampling mechanism. This sampling mechanism allows the controller to control the total numbers of monitoring entries in a switch based on monitoring-based application requirements or to avoid possible overflow/overloading problems, by setting the sampling ratio in the SDN-MON monitoring database via the SDN-MON monitoring APIs. The sampling ratio is defined as the ratio of sample size to population size, or the ratio of number of flows to be monitored by the monitoring table of SDN-MON to the number of existing flows in the network, i.e., $\Delta R_s = n_f / N_f$, where n_f is the number of monitored flows, and N_f is the number of existing flows in the network. Figure 3.2 illustrates in detail the components in the SDN-MON switch-side module and how packets are processed in the module for monitoring.

3.3.2 Monitoring process

For monitoring traffic at a switch, the controller firstly sends a message to the switch turns on the monitoring mode. The monitoring process in SDN-MON is initiated accordingly. When a packet arrives at a switch (where the monitoring mode is on), the pipeline packet processing of OpenFlow is performed. Concurrently, monitoring match fields are extracted from the

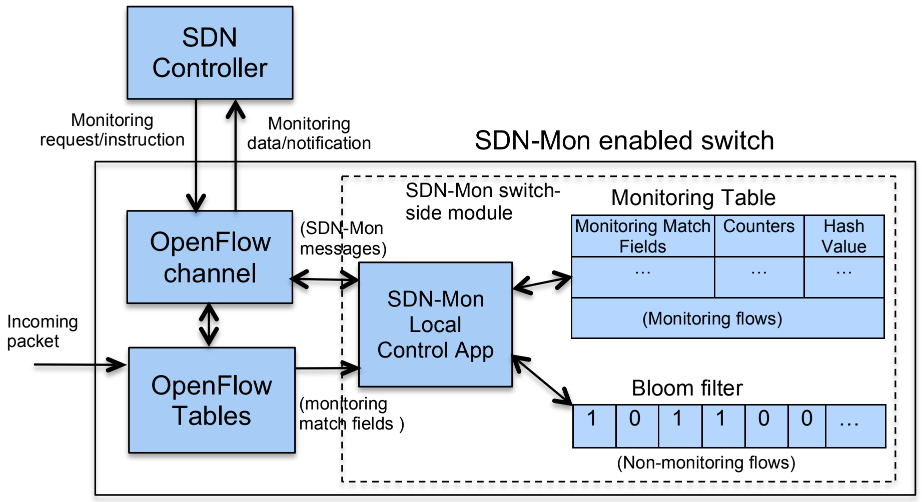


Fig. 3.2 Packet processing for monitoring at SDN-MON enabled switch

packet-header fields and passed into the SDN-MON local control app for monitoring. Based on the monitoring match fields, this application conducts a lookup process in the Bloom filter to verify if a corresponding entry exists in the Bloom filter. If the verification result is positive (this means that the entry based on these monitoring match fields is a non-monitoring entry), the SDN-MON local control app ignores this entry, and all following packets matching these monitoring match fields will not be monitored. If the verifying result is negative, the application conducts the lookup process in the monitoring table to find a monitoring entry whose monitoring match fields match the corresponding match fields in the packet.

If a match occurs, the counters in the matching monitoring-entry will be updated by increasing the packet count by one, and increasing the byte count by the number of bytes of the packet. If no matching entry is found, the SDN-MON local control app determines whether to monitor this flow based on the sampling ratio set by the controller. If the flow is determined for monitoring, a new monitoring entry, in which the values of monitoring match fields are set by those of corresponding match fields of the flow, will be created and inserted into the monitoring table. If the flow is not determined for monitoring, a new entry/element, in which the values of monitoring match fields are set by the same way, will be created and inserted into the Bloom filter to mark the flow as non-monitoring. The workflow of the monitoring process is illustrated in Fig. 3.3. In SDN-MON, the monitoring match fields are flexible and set by the controller based on the requirements of the controller applications. The controller can also insert specific monitoring entries for monitoring by using the SDN-MON monitoring APIs to send monitoring entry modification requests to a switch to insert the entries.

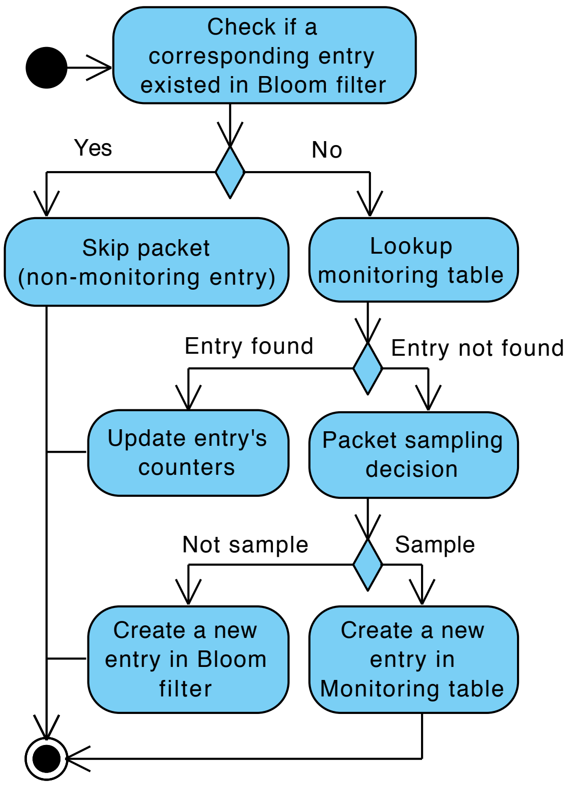

Check if a corresponding entry existed in Bloom filter
Yes
No
Skip packet (non-monitoring entry)
Lookup monitoring table
Entry found
Entry not found
Update entry's counters
Packet sampling decision
Not sample
Sample
Create a new entry in Bloom filter
Create a new entry in Monitoring table

Fig. 3.3 Workflow of monitoring process in SDN-MON switch-side module

3.3.3 SDN-MON Monitoring APIs

The SDN-MON monitoring APIs provide functions that the SDN-MON controller-side and switch-side modules can use to exchange messages containing monitoring control instructions or monitoring data between each other in the OpenFlow communication channel. These functions allow the controller to define the set of monitoring match fields for the monitoring table in the switches, manage the monitoring table by setting a sampling ratio based on its requirements or monitoring table status, and query statistics of monitoring entries in switch-side monitoring tables. For the switches and the controller, SDN-MON provides functions for adding a new monitoring entry into the monitoring table of a switch and removing an entry from the monitoring table. The functions of the SDN-MON monitoring APIs are as follows.

sendToController (SDN-MON message): This allows the switch-side module to send SDN-MON messages to its controller. These messages include the Monitoring Statistics Reply Message and Overflow Notification Message.

sendToSwitch (SDN-MON message, Switch ID): This allows the controller-side module to send SDN-MON messages to a specific switch. These messages include the Monitoring Statistics Request Message, Set Monitoring Match Fields Message, Set Sampling Ratio Message, Add Monitoring Entry Message, and Remove Monitoring Entry Message.

setMonitoringModeOn(Monitoring Match Fields, Counter Buffer Size, Query Time Interval, Switch ID): This function turns on the monitoring mode on a switch. It initiates the SDN-MON switch-side module with the specified monitoring match fields (i.e., 5 tuple match fields in the current specification of SDN-MON), counter buffer size S_b, and query time interval ΔT_q. The query time interval ΔT_q can be flexibly adjusted at a later time based on the controller's requirements, by using the setQueryTimeInterval function as described below.

setMonitoringModeOff(Switch ID): This function turns off the monitoring mode on a switch. All data structures of SDN-MON switch-side module will be cleaned up accordingly.

addMonitoringEntry (Monitoring Match Fields Pattern, Switch ID): This function adds a new monitoring entry into the switch-side monitoring table. For monitoring a specific flow, the controller uses this function to insert a new monitoring entry into the switch-side monitoring table for monitoring. The controller-side module encapsulates the monitoring match fields pattern into an Add Monitoring Entry Message and sends it to the switch specified by its switch ID.

removeMonitoringEntry (Monitoring Match Fields Pattern, Switch ID): This function removes an existing monitoring entry from the switch-side monitoring table, when the controller determines not to keep monitoring a specific flow/entry. The switch processes this command by removing the corresponding monitoring entry from the monitoring table and by concurrently creating a new Bloom filter element corresponding to the entry to add into the Bloom filter for marking the flow as non-monitoring.

resetMonitoringTable (Monitoring Match Fields, Counter Buffer Size, Query Time Interval, Switch ID): This function cleans up the existing monitoring table and resets it with the specified monitoring match fields, counter buffer size S_b, and query time interval ΔT_q.

setSamplingRatio (Sampling Ratio Value, Switch ID): This function sets a sampling ratio for the switch-side monitoring database. It can be reused subsequently to adjust the sampling ratio and control the monitoring table size to avoid the overflow or overload problem.

setQueryTimeInterval(Query Time Interval, Switch ID): This function sets a new query time interval ΔT_q. The controller uses this function to notify the monitoring switch each time it changes the query time interval. The switch then calculates and updates the counter-buffer-update time interval ΔT_u based on this new query time interval. The counter buffers on the switch will be updated based on the new value of ΔT_u accordingly.

setOverflowNotificationThreshold (Threshold Value, Switch ID): This function supports setting a threshold for the monitoring switch to pre-alert the controller concerning the overflow of the monitoring table. When the number of existing monitoring entries exceeds this threshold, which can be set based on the switch capacity, the switch will send an Overflow Notification Message to its controller to notify the controller. The controller can reuse this function at subsequent times to adjust the threshold to avoid the overflow problem.

3.3.4 SDN-MON communication protocol

For communication and exchanging the monitoring control instructions and monitoring data between switches and controller in the network, we design a communication protocol, called SDN-MON protocol. This protocol is an extension of OpenFlow protocol and it is integrated into OpenFlow protocol. SDN-MON protocol consists of messages that are implemented in experimenter message space of OpenFlow. The messages in our designed protocol can be categorized into two groups: monitoring control messages, and monitoring data query messages.

To integrate our monitoring protocol to SDN/OpenFlow, we design SDN-MON messages following the format of the OpenFlow messages [49] (i.e., using experimenter message types defined in OpenFlow). This make it easy for deployment and avoid any possible conflict or incompatibility in the communication channel. Experimenter extensions provide a standard way for OpenFlow switches to implement additional functionality within the OpenFlow message type space. This is a staging area for features meant for future OpenFlow revisions. A typical experimenter message is composed of an experimenter ID, an experimenter type, and experimenter arbitrary data. In SDN-MON messages, the experimenter ID is set to a unique 32-bit constant, which allows the SDN-MON modules to differentiate between SDN-MON messages and other experimenter messages that may exist in the communication channel. The Experimenter Type field in SDN-MON is set with the 32-bit ID of the module sending the message (note that both of the SDN-MON switch-side and controller-side modules are assigned with a unique ID for management purposes). With these settings, each SDN-MON message includes: an OpenFlow header, a SDN-MON experimenter ID (32-bit), a SDN-MON module ID (32-bit), and a message content. The detail design of SDN-MON messages are as follows.

Monitoring control messages

The monitoring control messages are designed for user applications at the controller to control monitoring parameters and modules in switches as the specified monitoring process in the proposed monitoring method. SDN-MON control messages are implemented where its header fields follow structure of an OpenFlow message header [42]. The general structure of a SDN-MON message is as follows.

- *ofp-header*: OpenFlow header.
- *SDN-MON Identifier* (4 bytes): Identifier of SDN-MON. This field is to identify if a message received at switch or controller is SDN-MON message or OpenFlow message so as to process the message based on SDN-MON defined process or OpenFlow default process. This field is set as with a hexadecimal value of *0xEFFE* in the our implementation.
- *Switch/Controller ID* (4 bytes): This field indicate the identifier of the destined receiver of the message. If the message is from controller to a switch, then this field is set as the ID of the switch. In the other hand, if the message is a response message from a switch to a controller, then this field is set as the ID of the controller.
- *SDN-MON message type* (4 bytes): This field is the code of the message. Upon receiving a SDN-MON message, switches or controller will base on this code to recognize the message in order to process it according to the defined monitoring process of the proposed method. The message types are set with codes from 1 to 14 corresponding to 14 messages of SDN-MON protocol.
- *SDN-MON control data* (8 bytes): The data included in the message to send from controller to a switch for control purpose, or from a switch to controller for response to a controller message.

Monitoring mode on/off message This SDN-MON message is used to initialize or terminate monitoring process in switch. The structure of this message is implemented as follows.

- *ofp-header*: OpenFlow header.
- *SDN-MON Identifier* (4 bytes): set with SDN-MON identifier, i.e., *0xEFFE.*
- *Switch ID* (4 bytes): set with the ID of the destined switch.
- *SDN-MON message type* (4 bytes): set with a 4-bytes hexadecimal value of 0x0001.
- *SDN-MON control data* (8 bytes): set with a hexadecimal value of 1 for starting monitoring mode, or 0 for terminating monitoring mode in switch.

A user or operator at the controller can start or terminate monitoring mode in switch by sending this message to switch. When a switch receive this message, if the control data field is set as 1, it starts monitoring traffic flows using its SDN-MON local monitoring modules. Similarly, if control data field in this message is set as 0, the switch reset all monitoring data storage and terminate the monitoring process.

Reset monitoring database message This message is used to reset the monitoring database in switch. Upon receiving this message, a switch reset its local monitoring database including monitoring table and Bloom filter. The structure of this message is as follows.

- *ofp-header*: OpenFlow header.
- *SDN-MON Identifier* (4 bytes): set with SDN-MON identifier, i.e., *0xEFFE*
- *Switch ID* (4 bytes): set with the ID of the destined switch.
- *SDN-MON message type* (4 bytes): set with a 4-bytes hexadecimal value of 0x0002.
- *SDN-MON control data* (8 bytes): This field is not required in this message. It is set as 0 by default.

Set sampling ratio message A user or network operator at the controller can use this message to set or adjust a sampling ratio for a switch. Upon receiving this message, the switch set or adjust the sampling ratio in its local monitoring process. The structure of this message is implemented as follows.

- *ofp-header*: OpenFlow header.
- *SDN-MON Identifier* (4 bytes): set with SDN-MON identifier, i.e., *0xEFFE*
- *Switch ID* (4 bytes): set with the ID of the destined switch.
- *SDN-MON message type* (4 bytes): set with a 4-bytes hexadecimal value of 0x0003.
- *SDN-MON control data* (8 bytes): set with a sampling ratio for switch. If this field is not specified, 1 is set by default (i.e., monitor all flows in the network).

Set query time window message This message is used to set or adjust a query time window in switch for polling monitoring entries in the switch. Upon receiving this message, the switch set or adjust the query time window in its local monitoring module. If the time window is defined in a switch, it can periodically send new or updated monitoring entries to the controller without any request from controller. This message can be used to automate the monitoring process, and reduce control overhead as the controller is not required to send a statistics query message in each period of the time window. The structure of this message is implemented as follows.

- *ofp-header*: OpenFlow header.
- *SDN-MON Identifier* (4 bytes): set with SDN-MON identifier, i.e., *0xEFFE*
- *Switch ID* (4 bytes): set with the ID of the destined switch.
- *SDN-MON message type* (4 bytes): set with a 4-bytes hexadecimal value of 0x0004.
- *SDN-MON control data* (8 bytes): set with a query time window for switch. If this field is not specified, 10 is set by default (i.e., 10 seconds is set for the query time window).

Set overflow notification threshold This message is optional and can be used to set or adjust an overflow threshold for switch. This threshold is defined by the number of entries (i.e., monitoring entries and flow entries) and can be set by the controller to get notification from switch if the existing number of entries in switch is about to overcome the threshold. The structure of this message is implemented as follows.

- *ofp-header*: OpenFlow header.
- *SDN-MON Identifier* (4 bytes): set with SDN-MON identifier, i.e., *0xEFFE*
- *Switch ID* (4 bytes): set with the ID of the destined switch.
- *SDN-MON message type* (4 bytes): set with a 4-bytes hexadecimal value of 0x0005.
- *SDN-MON control data* (8 bytes): set with a notification threshold for switch.

Response message This message is used by switch to response to a controller's control message. Upon receiving a control message from controller, switch will reply with a response message including a success code of 1 if defined instruction is executed successfully, or an error code of 0 if any error occurs. For instance, if a switch receive a set sampling ratio message, it update the sampling ratio in its monitoring module. Then it replies the controller with a response message with a success code of 1 if the ratio is updated successfully, or with an error code of 0 if any error. The structure of this message is implemented as follows.

- *ofp-header*: OpenFlow header.
- *SDN-MON Identifier* (4 bytes): set with SDN-MON identifier, i.e., *0xEFFE*
- *Controller ID* (4 bytes): set with the ID of the controller.
- *SDN-MON message type* (4 bytes): set with the message type to response to the controller.
- *SDN-MON control data* (8 bytes): set as 1 if success, or 0 if any error.

Monitoring data query messages

Monitoring data query messages are designed for users and applications at the controller to poll the monitoring entries in the switch. This message group consists of two messages: monitoring data request message and monitoring data reply message. The header fields in SDN-MON monitoring data query messages follow structure of an OpenFlow message header [42]. The details about these messages are defined as follows.

SDN-MON monitoring data request This message is used to request a switch to send monitoring data (i.e., monitoring entries) to controller for user applications. The structure of this message is as follows.

- *ofp-header*: OpenFlow header.
- *SDN-MON Identifier* (4 bytes): set with SDN-MON identifier, i.e., *0xEFFE*
- *Switch ID* (4 bytes): set with the ID of the switch from which to query monitoring data.
- *SDN-MON message type* (4 bytes): set with 0x0006.

SDN-MON monitoring data reply This message is used as response from switch to the controller when receiving a SDN-MON monitoring data request. This message is implemented as an experimenter multipart message type of OpenFlow. It is worth noting that for an OpenFlow multipart message, when the data to send to the controller is larger than a maximum number of bytes of a single message, the data will be transmitted as a sequence of continuous messages with the same header fields. Once receiving a multipart message, the controller parses the receiving data sequentially to get the data in a correct order. The implemented format of a SDN-MON monitoring data request message consists a number of fields as follows.

- *ofp-header*: OpenFlow header.

- *SDN-MON Identifier* (4 bytes): set with SDN-MON identifier, i.e., *0xEFFE*
- *Controller ID* (4 bytes): set with the ID of the controller to which to send monitoring data.
- *SDN-MON message type* (4 bytes): set with 0x0007.
- N_{fe} (4 bytes): set with the number of existing flow entries in switch.
- N_{me} (4 bytes): set with the number of existing monitoring entries in switch.
- *SDN-MON monitoring data*: encoded bytes of the monitoring entries to send to controller.

Once receiving a SDN-MON monitoring data request from the controller, a switch collects new or updated monitoring entries, encode these entries into a SDN-MON monitoring data reply message, then send the message to the controller.

3.4 Implementation

Our implementation of SDN-MON consists of the switch-side and controller-side modules. We implement the switch-side module on a base switch (i.e., Lagopus software switch [60]) and the controller-side module on a base controller (i.e., Ryu controller [63]). The Lagopus switch is a software OpenFlow switch that exhibits a high performance, is easy to deploy, and has been used widely in the commercial and research community recently. The switch-side module is implemented using C programming language and the controller-side module is implemented using Python. In our implementation, about 1500 lines of C code is added into the Lagopus switch for the SDN-MON switch-side module and only a few hundred lines of Python code is added to the Ryu controller for the SDN-MON controller-side module, which requires only a limited amount of processing power for the SDN-MON monitoring mechanism. Although SDN-MON extends switches, it remains deployable since it runs on the general-purpose processors of the switches.

Based on the design of the local data storage at switch as mentioned above, the monitoring data in the local monitoring module at switch consists of a monitoring table, and a Bloom filter. The global monitoring table is the main data storage component as they store the monitoring statistics of flows in the network. We implementation the monitoring table using a fast lookup data structure, i.e., hash table. With hash table implementation, the computational complexity for lookup process is O(1), and for update process is also O(1). The monitoring table contains a number of monitoring entries which are used to records statistics of flows in network. A monitoring entry is used to monitor a flow that consists of a number of fields and defined number of bytes as described below:

- *Entry hash*: 8 bytes (Unique hash value of a monitoring entry, this hash value is unchanged during the lifetime of the monitoring entry).
- *SrcIp*: 4 bytes (Source IP address of the flow)
- *SrcPort*: 2 bytes (Source port number of the flow)
- *DstIp*: 4 bytes (Destination IP address of the flow)
- *DstPort*: 2 bytes (Destination port number of the flow)
- *Proto*: 1 byte (Protocol number of the flow)
- *Packet count*: 8 bytes (Number of packets recorded for the flow)
- *Byte count*: 8 bytes (Number of bytes recorded for the flow)
- *Last update*: 8 bytes (Timestamp since the last update of the packet count and byte count of the flow)

As the resources in a switch (e.g., throughput, CPU, memory) are limited, it may become overloaded when the number of flows to be monitored become larger than its capacity. Therefore, our proposed monitoring method support sampling capability to reduce number of monitoring rules with a tradeoff with monitoring accuracy in case a switch is about to overload. The sampling is flow based with checks if an incoming flow is sampled or not. For this goal, we use a data structure that can quickly check the presence of a certain element with low memory consumption, i.e., Bloom filter. We implement the Bloom filter as a bit vector with filter size of 20 bits. This means the memory size of the implemented Bloom filter is 2^{20}, i.e., 128 KB. Initially, all bits in the bit vector are set to 0 when the SDN-MON monitoring functionality is initialized. To add an element to the Bloom filter, we hash the 5 tuple values of the flow (i.e., srcIp, srcPort, dstIp, dstPort, proto) a few times and set the bits in the bit vector at the index of those hashes to 1. To test for the presence of an element (i.e., checking if a flow was monitored or not), we hash the 5 tuple of the flow with the same hash functions, then check if those values are set in the bit vector. For resetting a Bloom filter in case receiving a monitoring reset message from the controller, the Bloom filter is clean up by setting all bits in the implemented bits vector to 0. As the Bloom filter can determine if an element is definitely not in the set and consume low memory (i.e., only 128 KB in our implementation), it suits well for sampling purpose of checking if a flow is sampled or not in our design.

When a packet come to a switch, the packet will be proceeded by the processing pipeline of OpenFlow tables. Concurrently, we extract the 5 tuple values from the packet header and pass it to the SDN-MON monitoring module in switch. The packet flow will be checked with the Bloom filter for sampling decision and will be saved to the monitoring table if a sampling is determined following the workflow in Fig. 3.3. For monitoring data storage at the controller, we implement a similar hash table for storing and updating monitoring entries received from switches. The monitoring hash table in controller has the same number of fields and required bytes as the switch's monitoring table as described above to make the monitoring data storage and update consistent.

For the communication between controller and switch in SDN-MON, we implement the SDN-MON messages. Each message has a structure, number and of bytes as described the SDN-MON protocol design above. In our implementation, message parsers are also implemented at switch and controller as extension of corresponding default parsers at switch and controller, to parse the byte arrays of SDN-MON messages in the right order for monitoring process at switch and the controller.

3.5 Performance evaluation

3.5.1 Experiment environment

We conducted experiments with a testbed SDN network as illustrated in Fig. 3.4. This network is composed of an SDN software switch running in a physical computer (bare metal setup), a controller and a host connecting to the switch in a loopback topology with 2 NICs in each host and switch. The software switch is a SDN-MON capable switch (i.e., SDN-MON switch-side extension modules integrated in a base Lagopus software switch version v0.2.0), and the controller is the SDN-MON capable controller (i.e., SDN-MON controller-side extension modules integrated in a base Ryu controller version v3.26). We ran a SDN-MON capable switch and a default Lagopus switch to evaluate the performance overhead.

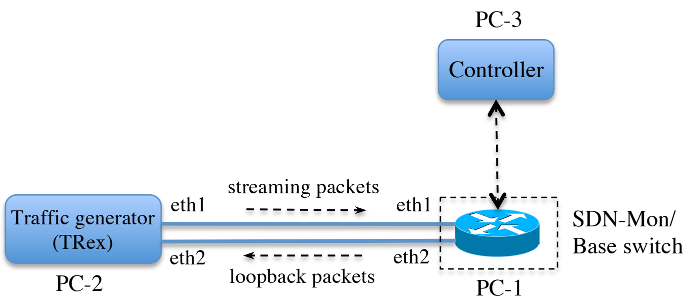


Fig. 3.4 Experimental setup

The hardware configuration of the experimental network is summarized in Table 3.1. The software switches in our experiments run with DPDK v16.11 [15] for increasing the speed of packet processing. The switch in each experiment worked on the server PC-1, a physical computer with a 2.6-GHz CPU (16 cores) with 32GB RAM. The Ryu controller worked on a separate computer (PC-3) and connected to a switch through a LAN. Two Intel Ethernet 10-Gigabit 2P X520 NICs were installed in PC-1 to handle switch ports (represented by interfaces 'eth1' and 'eth2' of the switch, as shown in Fig. 3.4). An two 82599ES 10-Gb SFI/SFP+ NICs were installed in the host running TRex for experimenting the SDN-MON switch, each NIC was connected to the corresponding NIC of the switch in a loopback topology for reliable experimenting the switch as a device-under-test (DUT).

	CPU	Memory size	NIC type
PC-1 (SDN-MON switch)	Intel Xeon E5-2650 v2 2.6 GHz (16 cores)	32 GB	Ethernet 10-Gb 2P X520 (eth1, eth2), NetEtreme II BCM57800 1/10 Gigabit Ethernet (eth0)
PC-2 (TRex)	Intel Xeon E5-2650 v2 2.6 GHz (16 cores)	32 GB	82599ES 10-Gb SFI/SFP+ (eth1, eth2)
PC-3 (SDN-MON controller)	Intel Xeon E5-2650 v2 2.6 GHz (16 cores)	32 GB	NetEtreme II BCM57800 1/10 Gigabit Ethernet

Table 3.1 Hardware configuration of experimental network

We ran the SDN-MON-supported Lagopus switch and the Ryu controller for the experiments. We set monitoring match fields to 5-tuple consisting of a source IP address, source port, destination IP address, destination port, and protocol, which are popular matching fields used in a variety of monitoring applications. The S_b was set to 5 and the ΔT_q was set to 10 seconds, which results in packet and byte counters being updated in every 2 seconds. We injected a stream of packets from host 1 to host 2. We measured the throughput, then repeated the same experiments for the default Lagopus switch to evaluate the performance overhead of the SDN-MON modules. We used TRex for experimenting the performance of the SDN-MON switch, where treaming packets with a defined configuration are sent from an interface of TRex device to the switch and the output traffic from the switch comes to the other interface of TRex device. Statistics including traffic rate of input and output traffic, packet drop-rate, and number of lost packets are recorded for each experiment.

3.5.2 SDN-MON switch overhead evaluation

Firstly, we conduct experiments on the throughput performance of SDN-MON, the base switch (i.e., Lagopus software switch) and other switches with 1 flow in the injected traffic in the experimental environment with 10G NICs. The throughputs are measured in Mpps and Gbps. The experimental results (Fig. 3.5 and Fig. 3.6) showed that Lagopus switch has a throughput of around 8 Mpps for basic forwarding functionality, which could be considered as a competitive performance among the available well-known SDN switches. This explains the reason why Lagopus switch was chosen as a base switch for the prototype implementation of SDN-MON. The figures also showed the throughput of SDN-MON as about 7.5 Mpps for the basic forwarding case, which show a small overhead as the monitoring process of SDN-MON was functioned besides the forwarding functionality in such case.

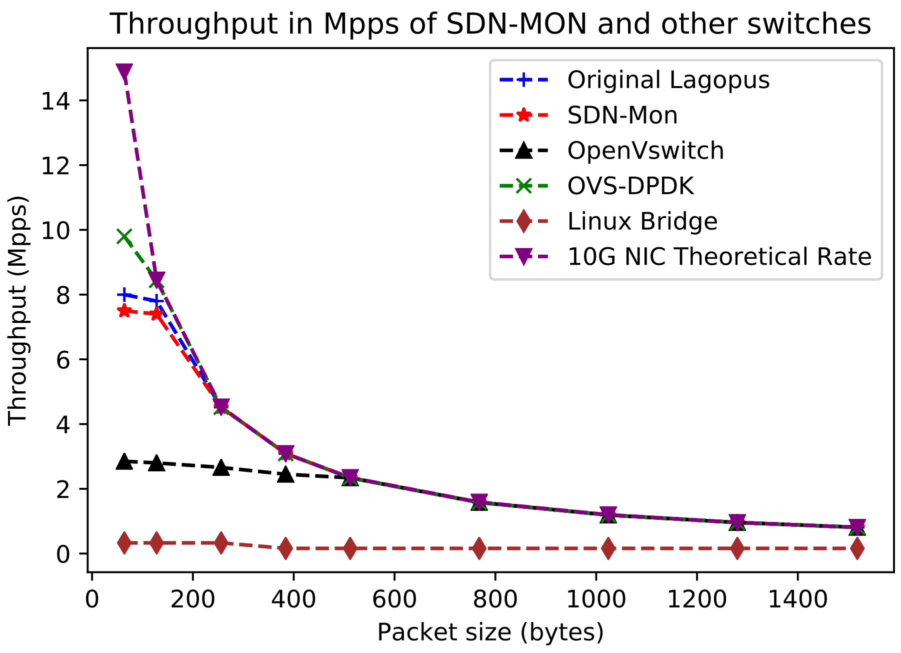


Fig. 3.5 Throughput in Mpps of SDN-MON and other switches

We evaluated the performance of the SDN-MON-supported Lagopus switch when monitoring 18000 5-tuple flows (created when monitoring traffic of the experimental pcap trace). The original (unmodified) Lagopus switch is set with 1 flow entry installed for basic forwarding from the source interface to the destination interface in the switch (i.e. the original Lagopus does not handle 5-tuple based monitoring task in this case). The packet size of each packet in the experimental traffic is of 64 bytes in each experiment. In the experiments for SDN-MON overhead, we set the sampling ratio of SDN-MON to 0.5 (this results in 9000 monitoring entries will created at the monitoring table of SDN-MON and 9000 Bloom filter elements will be created at the Bloom filter). We measured the packet output rates in Mpps through different packet-injecting rates from 0 to 10 Mpps. The input rates R_is and the output rates R_os measured by TRex were recorded.

Figure 3.7 shows the R_os of SDN-MON switch (with one flow entry for forwarding, and 9000 monitoring entries and 9000 Bloom filter elements for monitoring) and the default

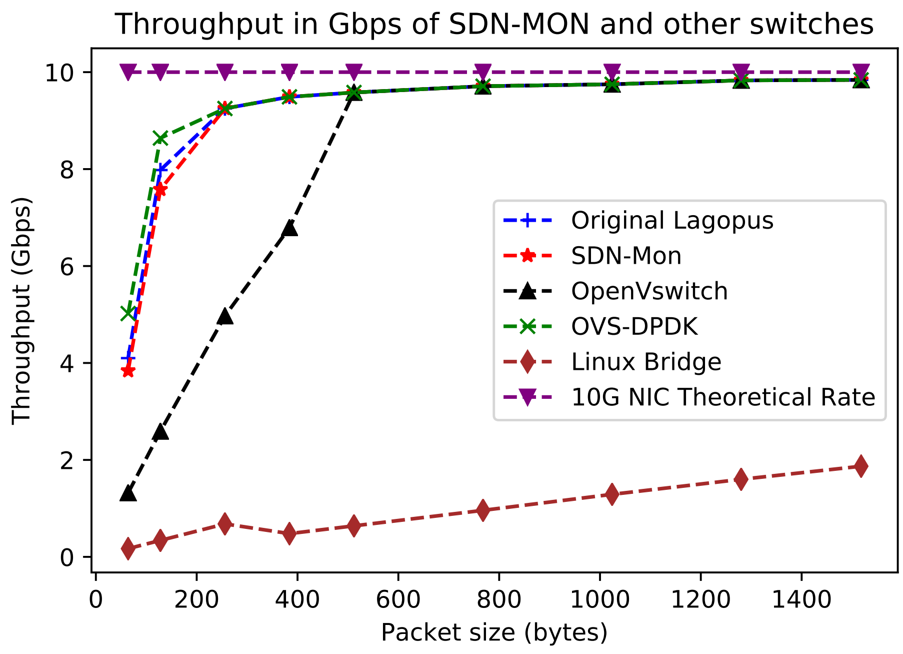


Fig. 3.6 Throughput in Gbps of SDN-MON and other switches

Lagopus switch (with only one flow entry). The experimental results show a small monitoring overhead of SDN-MON, which is calculated as only 11% for monitoring the rather large number of flows in the network traffic (i.e., 18000 flows).

Moreover, we conducted experiments on the throughputs of SDN-MON and other monitoring framework/switch to show its throughput among with the base switch and others (i.e., OVS, OVS-DPDK) running on the same deployed network environment. In the experiments, SDN-MON handle monitoring task with about 10000 monitoring entries with sampling ratio 1.0 and other switches were set with only 1 rule for basic forwarding. The throughputs were measured in Mpps and Gbps. The experimental results in Mpps (Fig. 3.5) and in Gbps (Fig. 3.6) show that SDN-MON has small monitoring overhead a competitive throughput compared to compared to the base Lagopus switch and other switches.

3.5.3 Impact of the sampling capability on enhancing the throughput of the proposed architecture

In order to demonstrate the positive impact of the sampling capability of the proposed architecture on enhancing its performance, we conducted additional experiments to measure the throughput of SDN-Mom on different sampling ratios. We set the sampling ratio from 0 to 1.0. The experimenting traffic includes 18000 active flows. We injected traffic with packet

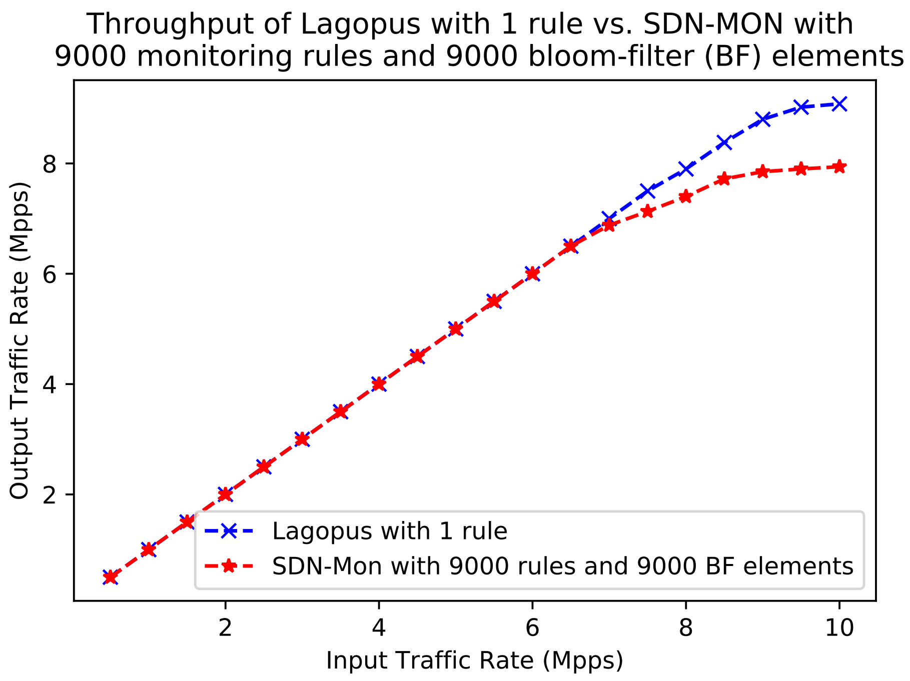


Fig. 3.7 Throughput of default Lagopus with 1 rule vs. SDN-MON with 9000 rules and 9000 bloom filter elements

input rate from 6 Mpps to 8 Mpps and measured the output traffic in Mpps. The experimental results (Fig. 3.8 showed the positive impact of the sampling capability which enhanced the throughput the monitoring switch when the traffic input rate overcome the processing limit of the switch (i.e., over 6.5 Mpps as in Fig. 3.8). These results demonstrated the validity of our proposal which integrated sampling capability with a Bloom filter data structure.

3.5.4 Validating the efficiency of the proposed architecture

In addition, to demonstrate the validity of the proposal on enhancing the performance of the default SDN monitoring mechanism, we have conducted additional experiments on the performance of SDN-MON compared with the default monitoring mechanism of the base SDN switch (i.e., Lagopus software switch) at different numbers of active flows. In these experiments, traffic with different numbers of active flows from 1 to 100000 of 64-byte packets were injected and maximum throughputs of SDN-MON and the base switch are measured. The experimental results (Fig. 3.9) demonstrated the higher throughput performance of SDN-MON compared to the based switch when the number of active flows is around or above1500 flows (i.e., the crossover point in Fig. 3.9). The source of the crossover point is because SDN-MON monitoring modules and process are operated besides

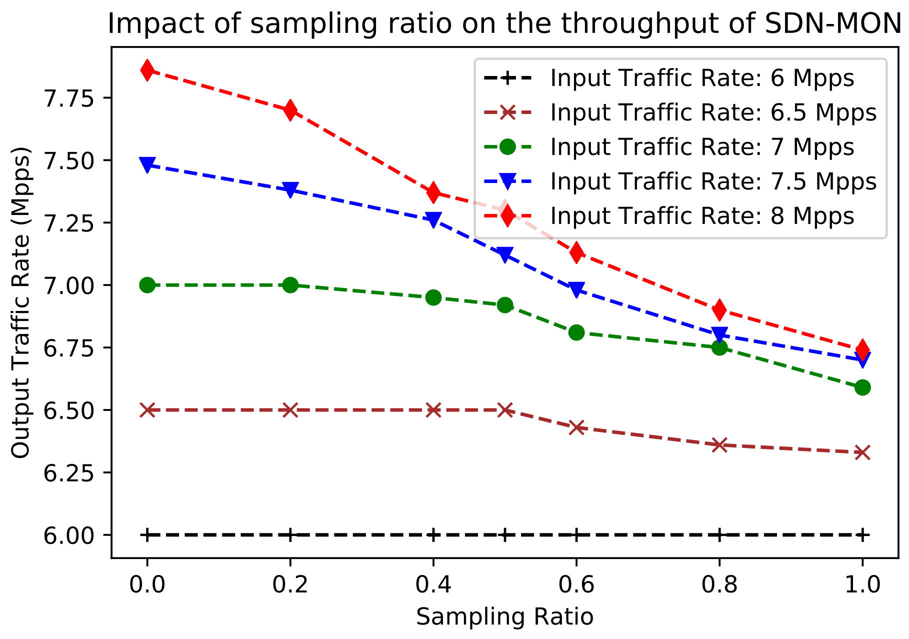


Fig. 3.8 Impact of sampling ratio on throughput of the proposed architecture (SDN-MON)

the forwarding functionality in such case resulting in a certain overhead is introduced in SDN-MON compared to the default Lagopus with operating only forwarding functionality, and the general but heavy flow matching in pure SDN/Lagopus switch (details about main factors that affect the performance difference between the default SDN/Lagopus switch and our proposal are described in section 3.6 of this chapter). As showed in Fig. 3.9, the throughputs of SDN-MON and the base switch for monitoring 20000 active flows are 5.8 Mpps and 3.4 Mpps correspondingly, and for 100000 active flows, the throughputs are about 3.5 Mpps for SDN-MON and 0.8 Mpps for the base switch. These results demonstrated that the proposal enhances the performance of switch for fine-grained monitoring of network traffic.

3.5.5 SDN-MON system overhead evaluation

We evaluated the system overhead of SDN-MON from two aspects: (1) the system overhead in various monitoring-table sizes (the number of active monitoring entries in the monitoring table) without background traffic, and (2) the system overhead at various R_i of background traffic. To evaluate the system overhead of SDN-MON, we built a controller program that leverages the SDN-MON monitoring APIs to query monitoring statistics in the SDN-MON-supported Lagopus switch from the controller. The system overhead is represented by the elapsed time of a round trip since the controller sends a monitoring-statistics request to

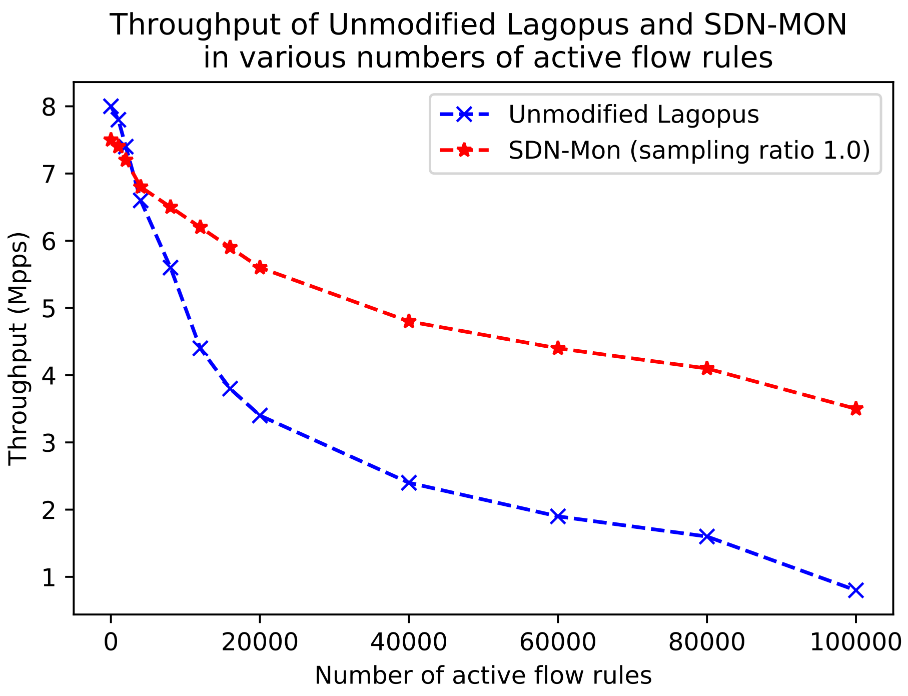


Fig. 3.9 Throughput of SDN-MON vs. default Lagopus in different number of active flow rules

the switch until it completes receiving all statistical data from the switch. This elapsed time consists of the time for the monitoring-statistics request to reach the controller since it was sent, the time for processing in the switch to collect the requested data and create a monitoring statistics reply message, and the time since the reply message was sent until it reaches the controller. In this experiment, the sampling ratio of SDN-MON was set to 0.5. The controller program in our experiment sent a monitoring-statistics request to the switch every 10 seconds periodically, and the timestamps were marked to observe the exact time when the controller sent the request and when it received reply data from the switch.

For the performance evaluation (1), we use TRex to generate experimenting traffic containing different numbers of active flows from 0 to 20000, which resulted in 0 to 10000 monitoring entries in the monitoring table of the SDN-MON switch (sampling ratio 0.5 was set in these experiments). In each experiment, we injected the packets in the pcap file and started to observe the system elapsed time when the sending host completed injecting the packets. We observed the system elapsed time 5 times and calculated the sample means and standard deviations. Figure 3.10 shows the observed system elapsed time. We observed that the elapsed time increased from 1.6 to 180 ms when the number of entries in the monitoring table increased from 0 to 10000. This is reasonable because for a larger number of monitoring

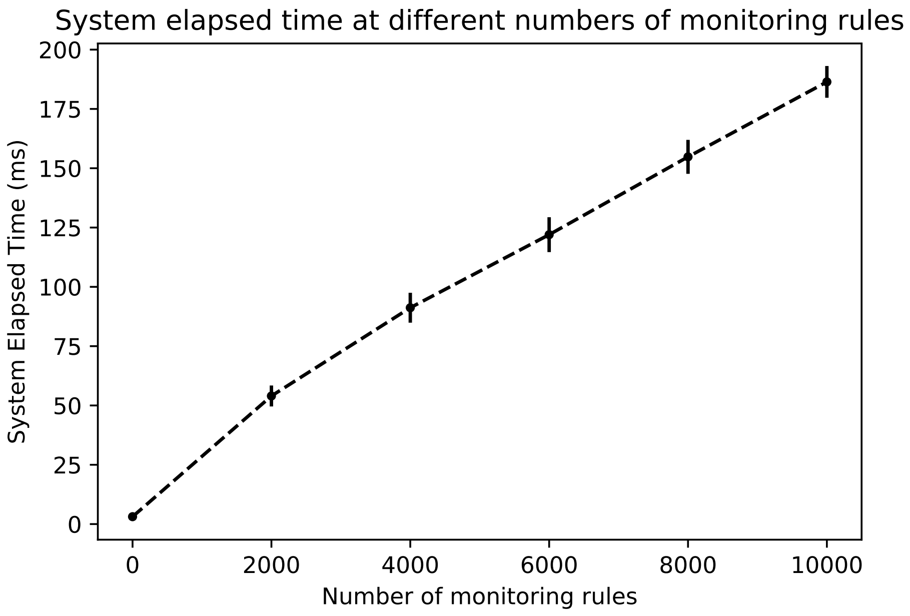


Fig. 3.10 System elapsed time at different numbers of monitoring rules

entries, the switch consumes more time for collecting the statistics from all entries, and transferring a larger amount of statistical data also results in a greater delay time in the communication channel between the switch and controller. The experimental results show that the system elapsed time of SDN-MON-based monitoring is small, even for pulling statistical data of thousands of monitoring entries.

We conducted the performance evaluation (2) with experimental network traffic containing 20000 active flows, which creates 10000 monitoring entries in the monitoring table of the SDN-MON switch accordingly at a sampling ratio of 0.5. We injected the packets at various R_i from 0 to 6 Mpps to create the background traffic in the experiments. We observed the elapsed time of the system 5 times and calculated the sample means and standard deviations. Figure 3.11 shows the elapsed time of the monitoring system in different R_i from 0 to 6 Mpps. The experimental results show that the overhead caused by the background traffic was negligible compared with the cases of no background traffic.

3.6 Discussion

Monitoring performance of SDN/Lagopus switch and SDN-MON

As presented in chapter 2 about the flow matching mechanism of OpenFlow-based SDN switch [49] and its implementation instance (e.g., Lagopus software switch [60]), a large

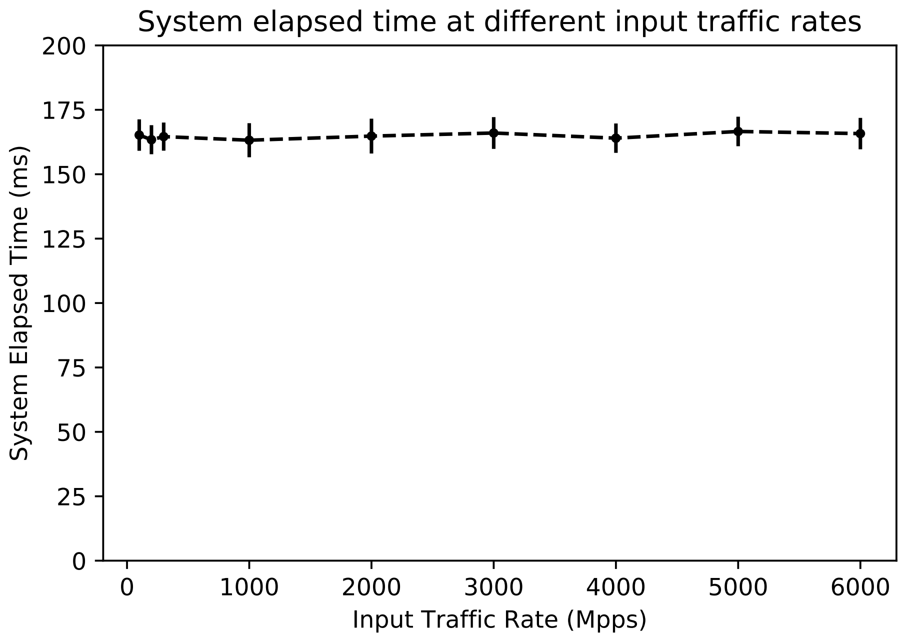


Fig. 3.11 System elapsed time at different input traffic rates

number of fields is required to be checked for each incoming packet in the flow entry matching mechanism (i.e., about 40 different match fields, which requires 1261 bits to implement match fields of a flow entry). These are the fields required for various actions to packets forwarding and controlling of SDN. In SDN-MON, for monitoring purpose, only 5 different match fields (i.e., 5 tuple, which requires 104 bits to implement the monitoring match fields) are required to be checked to find a matching monitoring entry. The large number of fields required to be checked for each incoming packets is a factor that makes the flow entry matching in default OpenFlow-based SDN switch is more complicated and costly than the monitoring entry matching in SDN-MON. In addition, a typical OpenFlow switch (e.g., Lagopus switch) must implement wildcard matching, which is more costly than exact matching. Although wildcard matching supports controller to aggregate traffic for routing, it is not required for fine-grained flow monitoring. Thus, it is reasonable that we design our monitoring mechanism with exact match that suits the target monitoring purpose to reduce processing overhead.

Moreover, multiple actions (e.g., forward to an output port, update packet headers, drop packet, group the entry), and counters (e.g., per-port counters, per-queue counters, per-group counter, per-meter counters) are defined for each flow entry in OpenFlow switch [49] [60], which requires multiple actions and counters to be executed and updated for each match, resulting in non-trivial processing for the actions execution and counters update. For monitoring purpose, as updating counters and corresponding time stamp are main required actions, in our design and implementation of SDN-MON, no action is required, and only three updates are performed for each flow match: updating packet count and byte count of the matched flow, and update the last update timestamp. Thus, the cost for processing a monitoring entry match in SDN-MON is lower than the cost of processing a flow entry match in SDN flow tables. This is considered as a second factor that contributes to the lower overhead of SDN-MON compared to the default monitoring mechanism of SDN switch and its implementation instance.

In addition, OpenFlow switch introduced a priority matching scheme where each incoming packet will be matched with all existing flow entries in the flow tables to find out all matches, then the match with highest priority will be selected to process the packet. With this matching scheme, the switch must check all flow entries to find out all possible matches instead of concluding the matching process when a match is found. For monitoring purpose, such priority mechanism is basically not required. Thus, in our design and implementation of SDN-MON, matching process for a flow is concluded when a match is found, which results in less number of lookups and less computing cost in average compared to the case of lookup for the whole of all tables in SDN flow matching process. The above characters are considerably the main factors that contribute to the lower overhead of monitoring entry matching in SDN-MON compared to the flow entry matching of SDN switch and Lagopus switch as its implementation instance. The main characteristics of the two frameworks are summarized in Table 3.2.

OpenFlow switch (Lagopus)	SDN-MON
40 different match fields required to be checked for each incoming packet	5 different match fields (i.e., 5 tuple) to be checked for monitoring traffic flows
1261 bits required for match fields implementation	104 bits required (13 bytes for implementation of 5 tuple)
Tree matching in Lagopus implementation	Hash based matching
Multiple flow tables are required	A single monitoring table and a lightweight Bloom filter are required for monitoring process
All flow entries required to be checked to find all possible matches (match priority)	Matching process finishes when a hash match is found
Wild card enabled	Exact match (fine-grained rule match)
Multiple actions and counters executed and updated for each flow entry match	No action required and only three counter updates performed for each monitoring entry match

Table 3.2 Main characters of flow table lookup in Lagopus switch and monitoring table lookup in SDN-MON

On the other hand, an OpenFlow switch may implement a flow cache that store a limited number of frequently accessed flow table entries (e.g., Lagopus implementation [60]) for speed up the forwarding process. In the implementation of SDN-MON, we do not implement any cache for our monitoring module in switch as limited cache in switch should be reserved for forwarding functionality of the switch as a higher priority. Therefore, the original Lagopus may perform better than SDN-MON for small number of rules with the support of flow cache (e.g., the number of flows about less than 1500 flows corresponding to the crossover point in Fig. 3.9), as the monitoring module and related process are operated besides forwarding functionality in SDN-MON compared to only forwarding functionality operated in default OpenFlow switch/Lagopus. As SDN-MON performance is higher than the performance of default SDN monitoring mechanism for larger number of flows (e.g., over 1500 flows for the case of using Lagopus as a base switch), while the number of flows required to be monitored in fine-grained traffic engineering applications mostly over a few thousands or even hundreds of thousands, SDN-MON considerably performs monitoring better than default SDN monitoring mechanism for such traffic engineering applications.

The affect of flow sampling in our proposal

In the design of SDN-MON, we use Bloom filter for sampling mechanism in order to reduce monitoring overhead in case a switch is about to be overload, with a tradeoff with monitoring accuracy. The Bloom filter used in our design is a lightweight data structure to check the presence of an element, thus it suits properly for our purpose of checking if a flow is sampled or not for sampling support in our proposed monitoring process. We implement the Bloom filter as a bit array with low memory consumption (i.e., only 128 KB in our implementation), where a presence of a member/entry can be quickly checked with hashing the 5 tuple with a few hash functions, and check if all the bits in hashed positions are 1 (i.e., representing that the checking member is existed in the Bloom filter). Bloom filter is used for sampling mechanism to reduce the number of monitoring entries in case a switch is about to be overloaded. As the number of monitoring rules are reduced, and a Bloom filter member lookup is basically faster than a monitoring entry lookup, the monitoring overhead in switch is reduced accordingly with a tradeoff with monitoring accuracy. We consider supporting reliably removing existing Bloom filter elements with the Cuckoo filter [18] in the next version of our implementation.

Comparison with existing traffic monitoring methods

As our goal in this dissertation is in-band traffic monitoring in SDN that integrate monitoring capability into SDN APIs for fine-grained traffic engineering applications, we discuss the previous works with this target. As this chapter focuses on the aspect of reducing monitoring overhead in switch, we outline in this section the pros and cons of existing works in comparison with our proposal. The closet work to ours is UMON [81] that reduces monitoring overhead in SDN switch (i.e. Open VSwitch) for fine-grained monitoring. The overhead in switch is reduced by injecting subflow table for fine-grained monitoring and using kernel space of Open VSwitch. Although UMON extends OpenFlow based SDN switch (i.e., Open vSwitch) for fine-grained monitoring similar to our approach, its performance is still limited. Specifically, for monitoring 4432 fine-grained flows with a network deployment using 10G NICs, the throughput in switch is about 0.4 Mpps. While SDN-MON achieved over 6 Mpps in a similar network deployment of using 10G NICs as demonstrated by the experimental results presented above. The work in [29] reduces overhead at switch by implementing extended monitoring functionality with some telemetry metrics in kernel space of Open vSwitch. Although the monitoring overhead in switch is reduced by the support of kernel space for speeding up the monitoring process, the current design only introduces hop latency measurement. Other important monitoring metrics such as packet and byte counts are not implemented in this proposal, leaving the default monitoring mechanism of the OpenFlow switch to handle monitoring these traffic volume metrics. As our proposed monitoring method introduces lower overhead than default OpenFlow based SDN monitoring mechanism for monitoring a large number of flows (e.g., a few thousands flows or above) as showed in the above experiments, it is reasonable to infer that our monitoring method will achieve lower monitoring overhead than that the proposal [29] in terms of monitoring overhead reduction in switch for fine-grained traffic monitoring.

The aggregation based approach (i.e., OpenWatch [88]) reduced overhead in network based on adjusting flow aggregation level. However, this approach does not discuss reducing overhead per monitoring rule, as it uses flow entries for monitoring. Thus it may not fit well for fine-grained traffic engineering applications as these application requires fine granularity of flow statistics resulting in a large number of rules to be installed in switch for monitoring. In fact, using flow entries introduce high overhead as we demonstrated in our experiments in this chapter. Therefore, in terms of overhead in switch, our proposal achieve lower overhead per monitoring rule, resulting in lower overhead in switch when monitoring the same number of flows compared to OpenWatch.

The switch selection based approach introduced in FlowCover [70] and OpenTM [77] reduced overhead in network by adaptively select switches to poll flow statistics. Network overhead is reduced since the approach only poll statistics in selected switches instead of all switches. As this approach uses flow entries for as monitoring rules and does not discuss removing redundant rules in non-selected switches, the overhead in switch for monitoring the same number of flows will be higher than our proposed method. This indicates that for fine-grained traffic engineering, our proposed monitoring method will archive better performance in switch (i.e., lower overhead) than the approach.

For the time window based approach presented in Payless [9], the overhead in controller and network is reduced by adjusting statistics polling frequency (i.e., query time window) adaptively. With this approach, the business is each flow is predicted where low polling frequency is applied for less busy flows, thus the overhead in network is reduced. As this approach uses flow entries for monitoring, the overhead in switch will be higher than SDN-MON for monitoring a same number of flows. The approach in FlowSense [87] and OpenNetMon [78] reduced overhead by passively capturing OpenFlow events (i.e., PacketIn, and FlowRemoved messages) without injecting any monitoring message in the control channel. As the approach relied on OpenFlow events passively, it may not suit for neither real-time or fine-grained monitoring. Furthermore, the approach do not discuss reducing overhead per monitoring rules, thus overhead in switch will be higher than SDN-MON for monitoring same number of flows.

Overall, our monitoring method outperforms the discussed existing works in terms of overhead at switch for monitoring same number of flows. Table 3.3 summarizes a comparison between the closet work and our proposal. As the monitoring is integrated to a capable control platform (i.e., OpenFlow based SDN), our proposed method benefits network management, especially traffic engineering applications that requires monitoring a possibly large number of flows and flexibly controlling network traffic.

Method	Switch overhead reduction	Distributed monitoring overhead reduction	Network component	Integrated API	Fine-grained monitoring
UMON [81]	Yes	No	Switch (Open vSwitch)	Yes	Yes
A. Gulenko, et. al. [29]	Yes	No	Switch (Open vSwitch)	Yes	Yes
sFlow/NetFlow [57] [10]	Yes	Yes	External deployment	No	Yes
SDN-MON (ours)	Yes	Yes	Switch and Controller	Yes	Yes

Table 3.3 Comparison of our proposal with existing traffic monitoring methods in terms of monitoring overhead reduction in switch

3.7 Summary

This chapter presented our proposed monitoring method that reduces monitoring overhead in switch for fine-grained traffic monitoring in OpenFlow based SDN. The method, with its monitoring modules, data structure, and monitoring process are designed as a monitoring platform for OpenFlow based SDN, which is integrated to OpenFlow protocol. Our proposed method reduces overhead per monitoring rule in OpenFlow switch, thus it reduces the monitoring overhead for the switch. The experimental results show a low monitoring overhead at switch and small system elapsed time. Specifically, the experimental results as showed in Fig. 3.9 demonstrated that SDN-MON introduce lower monitoring overhead (i.e., higher throughput) for monitoring different number of active flows compared to the default SDN monitoring mechanism in the base switch. For instance, the throughput of SDN-MON and the base switch for monitoring 20000 active flows are 5.8 Mpps and 3.4 Mpps correspondingly, and for 100000 active flows, the throughputs are about 3.5 Mpps for SDN-MON and 0.8 Mpps for the base switch. This indicates that SDN-MON satisfies the requirement in terms of reducing overhead per monitoring rule for serving target networks and applications as specified in section 3.1. Moreover, as SDN-MON can monitoring over a hundred thousands of active flows as showed by experimental results above, the requirement in terms of number of active flows of target networks and application is also satisfied. In addition, the elapsed time of the monitoring system was also proved to be small, i.e., in an order of hundreds of mili seconds for monitoring thousands of active flows, and about a second for monitoring a hundred thousands of active flows). This indicates that the proposal satisfies the requirement in terms of processing time of the monitoring system (i.e., in order of seconds). Since our proposed monitoring method satisfied the requirements of the targeted networks and applications as outlined in section 3.1 of this chapter, it is reasonable to infer that the proposal is capable of serving monitoring task for the networks and applications.

Chapter 4

Distributed monitoring method for SDN

In this chapter, we present our monitoring method for SDN monitoring in distributed scenarios with multiple monitoring switches. In SDN, switches independently monitor flows that results in duplication of flows monitoring and produce overhead in switches and the network as analyzed in chapter 1. Specifically, when the number of flows traversing through the multiple monitoring switches is large, the duplication ratio may become large accordingly that results in significant overhead in switches. As the resources (e.g., throughput, memory, CPU) in switches are limited, reducing overhead caused by the monitoring duplication is important to enable monitoring in distributed scenarios with reduced overhead. Therefore, in our proposal, we aim at designing a monitoring method that can detect and eliminate the duplicated monitoring rules and distribute monitoring load over multiple monitoring switches in a balancing fashion. In this chapter, we firstly present in section 4.1 an empirical discussion on requirements of different networks and applications. Section 4.2 describes an overview of our proposed method. The architectural design of the method including monitoring components and modules are described in section 4.3. Section 4.4 presents the details of our proposed algorithm for detecting, removing duplicated monitoring rules and assigning non-duplicated rules into switches in a balancing manner. In section 4.5, we describes the evaluation of our proposed method in different aspects: the reduction of monitoring rules per switch, the balance of monitoring load in switches, algorithm processing time, and the system elapsed time.

4.1 Design requirements

Typically, in SDN, a main controller manages switches in the network. Therefore, in the proposal of this chapter, we assume that one controller manages monitoring switches in a distributed monitoring scenario where each monitoring switch is a monitoring point in the network. As the monitoring functionality is operated independently from forwarding functionality and does not rely on flow tables of SDN switch in our proposed monitoring platform, a small number of flow entries for basic forwarding in switch may be required for basic forwarding (e.g., a switch may install forwarding rules where matching fields are only source ip address, ingress port, and egress port, which results in a small number of rules required corresponding to number of physical links and hosts in the network). We target our proposed monitoring platform to serve a number of common networks including LAN, campus network, backbone network and smaller scaled networks, and common applications including heavy-hitter detection, traffic classification, routing optimization, bandwidth monitoring, and other traffic engineering applications. As discussed in chapter 2, the number of active flows required to be monitored in these networks and applications may reach thousands or a hundred thousands of active flows with a query time window in seconds, minutes, or hours. Therefore, we set a first requirement for our design that it must be capable of serving thousands or a hundred of active flows, and a second requirement is that processing time of the monitoring method must be under a threshold of the required query time windows of the target networks and applications (i.e., in an order of seconds). In addition, the monitoring tasks are mostly conducted in a few monitoring points. Therefore, we set a third requirement for our design in this chapter that is a capability to serve at least ten or even a few tens of monitoring switches. In summary, the requirements for our design in this chapter consists of: (1) Capability of serving a few thousands or a hundred thousands of active flows; (2) Processing time of the monitoring method must be in an order of seconds for such number of active flows; (3) Capability to serve ten or a few tens of monitoring switches in the network for distributed monitoring scenarios.

As outlined above, distributed monitoring scenarios with multiple independently operated monitoring switches introduce the duplication issue, while the resources in switches and in network are limited. Therefore, for distributed monitoring, it is important to solve the monitoring duplication problem and maintain a balance of monitoring load assigned in switches. This means that a distributed monitoring method must be able to eliminate redundant/duplicated monitoring rules to reduce monitoring overhead per switch and in network, and ensure monitoring rules are assigned to switches in a balancing fashion. As the process of such monitoring rules reduction and balance introduces a certain processing overhead at the controller, a distributed monitoring method must consider this factor in its design so as to limit the processing overhead at controller under the required time window. Our proposed monitoring method for a switch in chapter 3 demonstrates that it can monitor over a hundred of active flows with reduced overhead compared to the default SDN monitoring implementation in a base switch. Therefore, for distributed monitoring aspect, to meet the above mention requirements, we focus on the following main goals for our distributed monitoring method: (1) the method must eliminate redundant rules caused by monitoring duplication in switches; (2) the method must ensure assigning monitoring rules to switches in a balancing fashion; (3) the processing overhead at the controller, i.e., processing time of the distributed monitoring functionality, must be under a threshold of required query time windows of the networks and applications (i.e., in an order of seconds or minutes); (4) the method must be capable of serving ten or a few tens of monitoring switches in the network.

4.2 Method overview

To meet the above mentioned requirements, we propose a distributed monitoring mechanism where multiple switches can monitor traffic flows in the network in a coordinated way. This method is integrated with the proposed monitoring platform in chapter 3 as a consistent monitoring platform for fine-grained and distributed monitoring platform (SDN-MON). With our distributed monitoring method, network applications can define a specific time window to periodically collect monitoring data from the switches. To reduce monitoring overhead in network and processing overhead at the controller, we introduce the following strategies in the proposed method:

(1) To reduce monitoring overhead caused by redundant/duplicated monitoring rules, we detect and eliminate duplicated monitoring rules in switches and balance monitoring load assigned to the switches. The key factor of the proposed method is a distributed monitoring algorithm that filters monitoring records collected from the switches in each query. For each flow, only one monitoring rule is kept for monitoring the flow and remove other duplicated ones from the correspondent switches. The switch selection is determined based on the available status of the switches (i.e. represented by the number of existing monitoring and forwarding entries in the switch) at the current time and the most available switch is select. This selection strategy ensures the monitoring rules assigned to the switches are balanced based on the available status of the switches. The available status of the switches is updated periodically where the information of number of flow entries and monitoring entries are injected into the reply message sent to the controller in each query without requiring any additional message.

(2) Furthermore, in order to reduce the communication cost caused by monitoring messages in the control channel, we design the format of a monitoring rule with a minimum number of fields and required bytes which are dedicated for monitoring purpose as the proposed switch monitoring method introduced in chapter 3. On the other hand, we observe

that only new monitoring rules or updated ones (i.e., monitoring rules that have their statistics updated during the time window since the previous query) are necessary to be sent to the controller for statistics update at the controller. Our distributed monitoring method integrates a strategy that allow a switch to quickly check the update status of monitoring rules and only send new or updated monitoring rules to the controller in each query instead of all existing rules in the switch. With this strategy, the number of rules required to be sent to the controller is reduced, and thus additionally reduce communication overhead in the communication channel and processing overhead at the controller.

(3) In addition, although the controller is mostly a server where its resources (CPU, memory,...) is not as limited as the switches, our distributed monitoring algorithm was designed with targeting for small overhead in terms of processing time of the algorithm.

Our distributed monitoring method can be summarized with four main phases as follows.

1. SDN-MON monitoring module in switches actively monitor traffic flows in a network (i.e., create a new monitoring entry to monitor the flow) independently from the flow tables of the switch.
2. To get monitoring statistics, the SDN-MON monitoring module at the controller periodically sends query messages to monitoring switches. Upon receiving new/updated monitoring entries from switches, the proposed algorithm detects duplicated monitoring entries and select a switch with lowest load to keep monitoring the flow. Duplicated entries are marked for later removal.
3. When the process of duplication detection and storing monitoring entries is finished, the proposed algorithm sends messages to switches that are storing duplicated monitoring entries to remove the entries.
4. Upon receiving the monitoring entry removing message, each switch removes specified entries in its monitoring table.

The details of the architectural design, structures of monitoring data storage, and the distributed monitoring algorithm are described in the following sections.

4.3 Architectural Design

This section describes the architecture design of our distributed monitoring method. As the target of this method is to enable efficient monitoring in distributed scenarios with multiple switches, we design and integrate the components of this method as an extension on the

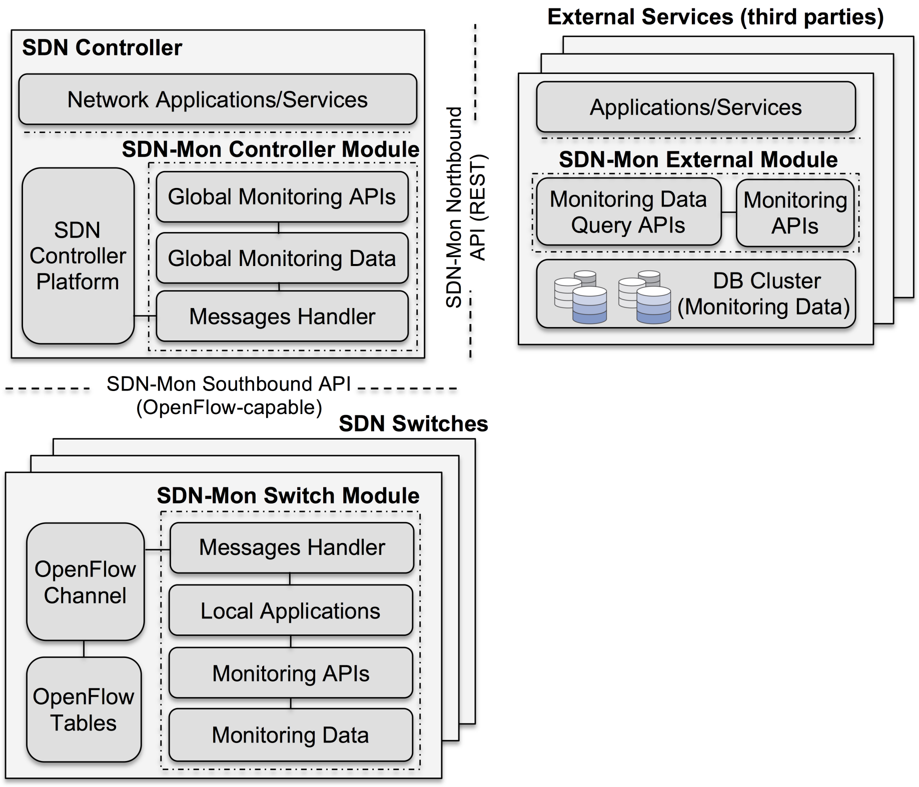


Fig. 4.1 Extended SDN-MON architecture.

SDN-MON platform proposed in chapter 3 (and call the platform with a consistent name as SDN-MON). Fig. 4.1 illustrates the architecture of extended SDN-MON which support monitoring with multiple switches and external applications and services of third parties.

The extended SDN-MON consists of three major modules: the switch module, the controller module, and the external module. The switch module handles the monitoring functionality at a switch [58], while the controller module provides global monitoring APIs and global monitoring data for controller applications and services. The external module provides monitoring data query APIs that support third parties to remotely extract the global monitoring data from the controller, and monitoring APIs that correspond to the monitoring APIs at the controller module for external applications and services. Each flow that traverses through a switch is monitored by a *monitoring entry* (or *m-entry*), which consists of monitoring match fields (e.g. 5-tuple), counters, and the updating timestamp, in the switch's monitoring database. The proposed mechanism in this section enables SDN-MON to automatically assign the monitoring load to multiple monitoring switches in a balanced way. This allows network operators to leverage SDN-MON APIs and the global monitoring data for their applications without any further effort for managing the multiple monitoring switches.

4.4 Monitoring Mechanism

4.4.1 Organization of Global Monitoring Data

In the proposed mechanism, we organize the global monitoring data at the controller into *Global Monitoring Tables (GMTs)*; each corresponds to each monitoring switch in the network. Each table stores, updates, and manages the monitoring data (i.e., m-entries) in a switch. Each m-entry includes following fields: *Monitoring match fields*, *Counters*, *Last update*, and *Hash*. The hash value of each entry is unique, and a corresponding hash-based data structure is used for fast lookups and other data based processes of *GMTs*. The m-entries in each *GMT* is kept updated by the controller through frequently polling of m-entries at the corresponding switch.

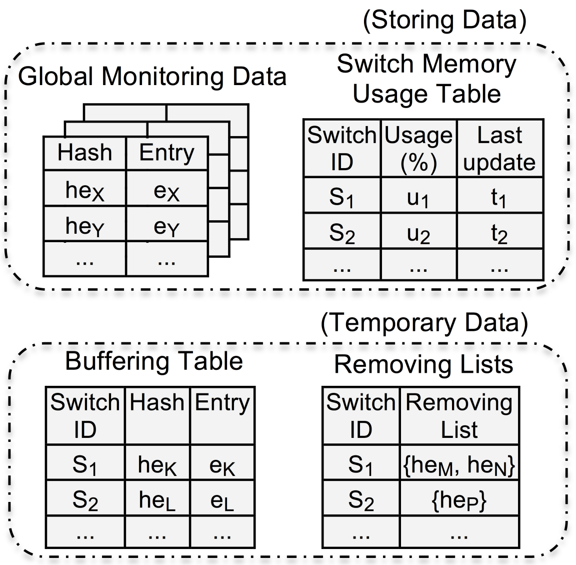


Fig. 4.2 Organization of the global data.

A lightweight table called *Switch Memory Usage Table* (Fig. 4.2) holds frequently updated information of the memory usages of switches for monitoring load balancing purpose. Each entry in this table holds the memory usage information of a certain switch, including the following fields: *<switch ID, Usage, Last update>*. The *switch ID* is the identification number of the switch. The *Usage* represents the percentage of used memory: *Usage* $= (N_{fe} + N_{me})/Capacity$, where *Capacity* is approximately estimated by the maximum number of flows and m-entries that the switch can handle (based on the switch configuration), N_{fe} and N_{me} are the current numbers of flow entries and m-entries at the switch. The *Last update* is the timestamp of the latest update of N_{fe}, N_{me} and *Usage*.

Besides the *GMTs*, the controller also contains temporary data structures consisting of a *Buffering Table* and *Removing Lists*. The buffering table holds selected m-entries from lists of m-entries that the controller receives for each monitoring data query. Removing lists (*RLs*), in which each one corresponds to each monitoring switch, hold hashes of rejected m-entries. These m-entries are duplicated ones that are not chosen from the switch selection process. When the controller completes processing the received monitoring data of each querying time, the m-entries in the removing lists will be physically removed from that switch, and the buffering table and the removing lists will be cleared for processing the received data of the next querying time.

4.4.2 Monitoring workflow

For every query-time-interval, the controller sends *SDN-MON Data Request* messages to all monitoring switches to query the new and updated m-entries at switches. Each switch responds such request by sending a *SDN-MON Data Reply* message including a list of new m-entries and updated ones (the m-entries whose counters have been updated since the previous query of the controller) to the controller (Fig. 4.3). A *SDN-MON Data Reply* also includes N_{fe} and N_{me}, which are counted by the SDN-MON switch module, for updating the memory usage information of that switch in the *Switch Memory Usage Table* at the controller. For a flow that traverses through multiple switches in network, such switches may actively install m-entries to sample/monitor that flow (the probability of sampling the flow is based on the sampling ratio), resulting in the duplicated m-entries stored at the switches.

Upon receiving the m-entries from switches, the controller filters the m-entries by: selecting a switch with smallest memory usage to keep monitoring a m-entry for each flow, putting the hashes of the duplicated m-entries to *RLs* for later removal at switches. For switches whose removing list is not empty, the controller sends instructions to them to physically remove the rejected entries after processing all received m-entries of a querying time. Algorithm 1 illustrates the detailed steps for processing the received monitoring data of a querying time at the controller.

4.4.3 Distributed monitoring algorithm

In the proposed monitoring algorithm (algorithm 1), the controller checks each m-entry e_X of each m-entries list L_i received from a switch S_I. If the entry is an updated version of an existing monitoring entry (i.e. the entry that arrived at the controller in a previous query), the controller updates the corresponding m-entry in the *GMT* of that switch. If not (i.e., the m-entry is a new one that arrives at the controller for the first time), the controller check if

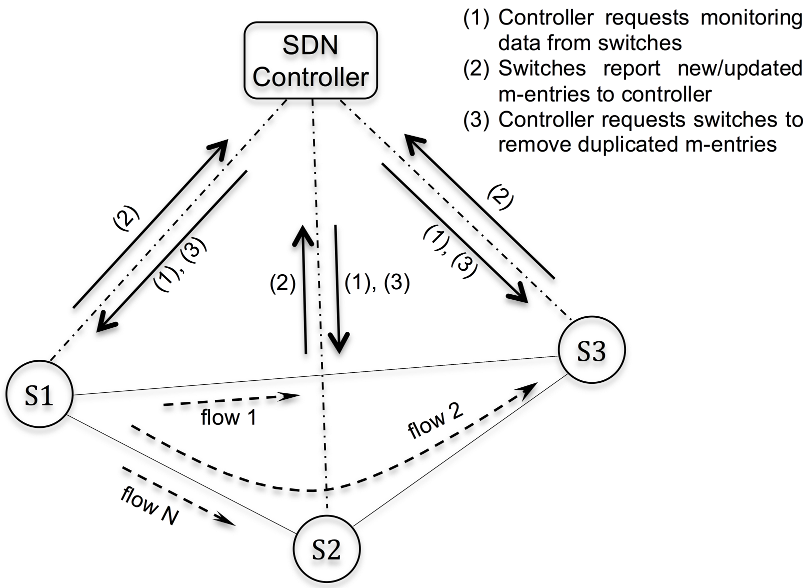

SDN Controller
(1) Controller requests monitoring data from switches
(2) Switches report new/updated m-entries to controller
(3) Controller requests switches to remove duplicated m-entries
(2)
(1), (3)
(2)
(1), (3)
(2)
(1), (3)
S1
S2
S3
flow 1
flow 2
flow N

Fig. 4.3 Controller-switches communication.

the buffering table has a duplicated m-entry (i.e., an entry whose hash is the same as the hash of e_X). If not, the controller inserts the entry e_X into the buffering table. If the buffering table has a duplicated m-entry e_Y (whose hash is the same as the hash of e_X), the controller compares the memory usages of switch S_I of e_X and S_{e_Y} of e_Y. If the switch memory usage of S_I is less than the switch memory usage of S_{e_Y}, the controller inserts e_X to the buffering table, and puts the hash of e_Y to the *RL* of S_{e_Y}. Else, the controller puts the hash of e_X to the *RL* of S_I.

After the above mentioned processing, non-duplicated entries from all arriving m-entries of a querying time (including the originally non-duplicated entries, and the chosen m-entries among duplicated ones through the switch selection processes) are put to the buffering table (task *(1)* in Fig. 4.4). The hashes of non-chosen duplicated m-entries are put to the *RLs* of correponding switches for later removing process (task *(2)* in Fig. 4.4). After each time interval ΔT since the controller receives the first *SDN-MON Multipart Reply* message of a querying time, the controller puts the m-entries in the buffering table to the *GMTs* of the corresponding switches, and clears/resets the buffering table to be ready for processing the arriving m-entries of the next querying time (task *(3)* in Fig. 4.4). Then for each *RL*, the controller sends an instruction containing the entry-hashes in *RL* to the corresponding switch to remove those m-entries from the local monitoring module of the switch, and clears/resets the *RL* of that switch. The time interval ΔT can be set equal to the query-time-interval (the time interval between two continuous queries), or it can be flexibly set by a time amount that is enough for the controller to complete the above mentioned processings for all arriving

```
Data: Lists of m-entries received from switches
for Each list of m-entries $L_i$ from a switch $S_I$ do
    for Each m-entry $e_X$ in $L_i$ do
        if $e_X$ is an updated m-entry then
            Update the corresponding m-entry in Global Monitoring Table of $S_I$;
        else
            if $e_X$ is not existed in Buffering Table then
                Insert $e_X$ into Buffering Table;
            else
                Assume $e_Y$ is the duplicated m-entry, $S_{e_Y}$ is corresponding switch of $e_Y$;
                if switch-usage($S_I$) < switch-usage($S_{e_Y}$) then
                    Insert $e_X$ into Buffering Table;
                    Insert $e_Y$ into Removing List of $S_{e_Y}$;
                    Update Switch Memory Usage Table;
                else
                    Insert $e_X$ into Removing List of $S_I$;
                end
            end
        end
    end
end
```

Algorithm 1: Pseudocode for processing the received m-entries of a querying time at the controller.

SDN-MON Multipart Reply messages of a query (and should be less than or equal to the query-time-interval). By default, ΔT is set equal to value of the query-time-interval.

Fig. 4.5 shows an example of how arriving m-entries are distributed to *GMTs* and *RLs* after the duplicated m-entries detecting and switch-selecting phases of algorithm 1. The monitoring entry *e1* is monitored in switches *S1* and *S3* (represented by *e1-S1* and *e1-S3* in Fig. 4.5). Since *S1* has memory usage of 42%, greater than the memory usage 31% of *S3*, after processing the algorithm with the support of the buffering table, the hash of the entry *e1* is put in the *RL* of *S1*, and the entry *e1* is inserted to the *GMT* of *S3*. Similarly, the monitoring entry *e2* appears in *S1*, *S2*, and *S3*. Since *S3* has the smallest memory usage among the three switches, it is chosen for monitoring *e2*. The result is that the entry *e2* is inserted to *GMT* of *S3*, and the hash of *e2* is put to the *RL* of *S1* and *S2* correspondingly. Similar process occurs for the entry *e3*. The result of the overall process (including actual removing of m-entries in *RLs* from corresponding switches) is that the monitoring load for *e1*, *e2*, and *e3* is distributed on the idlest switches among *S1*, *S2*, and *S3* in a balancing way based on the switch memory usage of the switches at the moment when the processing is performed, without any duplication.

The proposed algorithm detects and eliminate the duplication in the received monitoring data and adaptively selects polling-switches with lowest load (i.e., lowest number of monitoring entries and flow entries) for monitoring flows. With the proposed algorithm, mon-

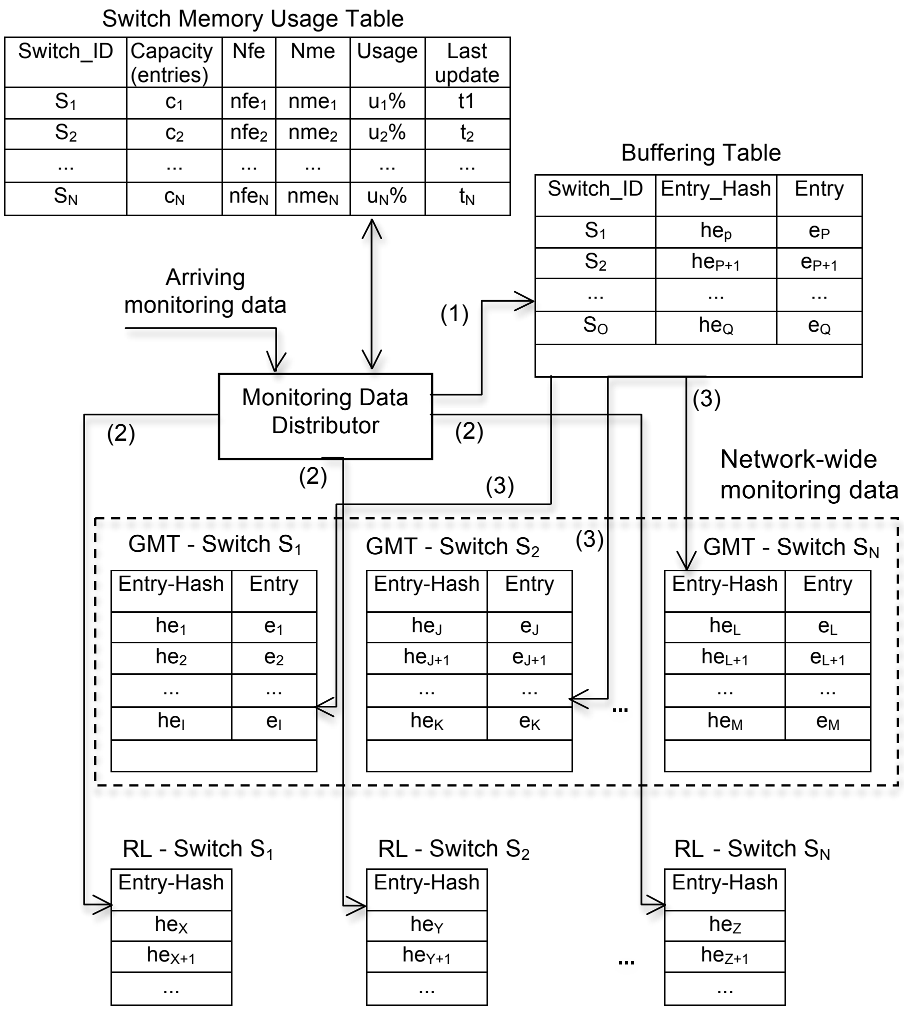
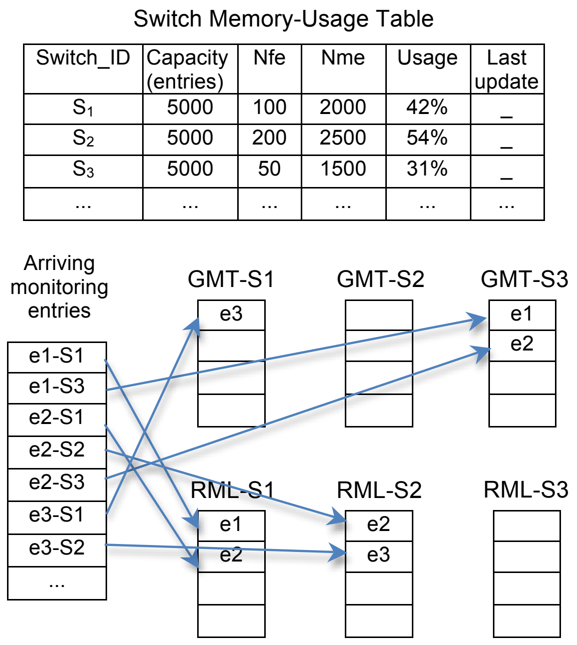


Fig. 4.4 An illustration on duplicated monitoring entries detection and assignment of monitoring entries to global monitoring tables of switches

itoring overhead per switch is reduced since the overhead caused by duplicated monitoring is rejected, and the monitoring load is distributed over switches in a balancing manner.

4.5 Implementation

We implement our proposed distributed monitoring method in this chapter as an extension of the monitoring platform proposed in chapter 3. The extensions mostly focus on implementation of global monitoring data structures at the controller, the distributed monitoring algorithm, and communication messages for monitoring control and monitoring data query exchange between a controller and switches in the network. We use C programming language



Switch Memory-Usage Table

Switch_ID	Capacity (entries)	Nfe	Nme	Usage	Last update
S_1	5000	100	2000	42%	_
S_2	5000	200	2500	54%	_
S_3	5000	50	1500	31%	_
...	...	...	...	...	...

Fig. 4.5 An illustration of global monitoring data processing and the switch selection scheme.

for implementing extensions in switch, and Python for implementing extensions for the controller as these programming languages are the ones used in the base switch (i.e., Lagopus software switch) and the base controller (i.e., Ryu controller) of SDN-MON implementation. The implementation for SDN-MON switch is about 1800 lines of C codes and for SDN-MON controller and the monitoring messages are about a few hundreds lines of Python codes. Further details of the implementation of the proposed distributed monitoring method in this chapter is presented in the following subsections.

According to the design of the global data storage at the controller as mentioned above, global monitoring data consists of: Global monitoring tables, Switch memory usage table, Buffering table, and removing lists. The global monitoring tables is the main data storage component as they store the monitoring data of monitoring switches in the network. Each table represents for monitoring data of a switch, and the table is identified by the identification number of the switch (i.e., switch ID value). We implementation these tables using a fast lookup data structure, i.e., hash table. With hash tables implementation, the computational complexity for lookup process is O(1), and for update process is also O(1). Each table contain a number of monitoring entries. A monitoring entry is used to monitor a flow that consists of a number of fields and defined number of bytes as described below:

- *Entry hash* (8 bytes): Unique hash value of a monitoring entry, this hash value is unchanged during the lifetime of the monitoring entry.

- *SrcIp* (4 bytes): Source IP address of the flow
- *SrcPort* (2 bytes): Source port number of the flow
- *DstIp* (4 bytes): Destination IP address of the flow
- *DstPort* (2 bytes): Destination port number of the flow
- *Proto* (1 byte): Protocol number of the flow
- *Packet count* (8 bytes): Number of packets traversed though the flow
- *Byte count* (8 bytes): Number of bytes traversed through the flow
- *Last update* (8 bytes): Timestamp since the last update of the packet count and byte count of the flow

For maintaining available status of switches in the network for switch selection when a duplication occur based on the proposed algorithm (i.e., algorithm 1), we implement the Switch Memory Usage table using hash table data structure. With hash table data structure, a number of entries that it can store can be tens of thousands or even millions of entries. However, as the number of monitoring switches in a network is typically not too large (e.g., a few or a few tens of switches), this table is lightweight table which may store a few or a few tens of entries depending on the number of switches used for monitoring task in the network. Each entry in this table contains information about the availability of a switch, which consists of a number of fields as defined below:

- *Switch ID* (4 bytes): Unique identification number of a switch in the network. For monitoring in a distributed scenario with multiple monitoring switches, this ID is used to identify a switch at the controller.
- *Capacity* (4 bytes): An estimated capacity threshold of a switch (i.e., maximum number of entries that the switch can store and process), which can be approximately estimated and set by the controller based on capacity support of the switch.
- N_{fe} (4 bytes): Existing number of flow entries in the switch
- N_{me} (4 bytes): Existing number of monitoring entries in the switch
- *Usage* (2 bytes): Percentage of resource usage in the switch. This percentage is calculated by the formula $Usage = (N_{fe} + N_{me})/Capacity$ as specified in the above section.
- *Last update* (8 bytes): Timestamp since the last update of the packet count and byte count of a flow

For filtering and detecting duplicated monitoring entries sent from multiple monitoring switches in the network, a buffering table is implemented as the design of global data specified in the above section. As this table may be large depending on the number of monitoring entries received from multiple switches in a query, we implement it using hash table data structure. A buffering table is a temporary data structure designed for the duplication filtering and detecting process and will be reset when the proposed algorithm finish duplication detection and elimination process in each query time. A buffering table temporarily storing a number of monitoring entries of the detecting phase. Each entries in this table consists of a number of fields representing for a monitoring entry, and a *Switch ID* to indicate which switch the monitoring entry is sent from, as specified below:

- *Switch ID* (4 bytes): Unique identification number of a switch in the network.
- *Entry hash* (8 bytes): Unique hash value of monitoring entry.
- *SrcIp* (4 bytes): Source IP address of the monitoring entry
- *SrcPort* (2 bytes): Source port number of the monitoring entry
- *DstIp* (4 bytes): Destination IP address of the monitoring entry
- *DstPort* (2 bytes): Destination port number of the monitoring entry
- *Proto* (1 byte): Protocol number of the monitoring entry
- *Packet count* (8 bytes): Number of packets traversed though the monitoring entry
- *Byte count* (8 bytes): Number of bytes traversed through the monitoring entry
- *Last update* (8 bytes): Timestamp since the last update of the packet count and byte count of the monitoring entry

In the proposed distributed monitoring algorithm, duplicated monitoring entries that are detected and filtered out from the duplication detecting phase will be rejected and the related instructions will be sent to corresponding switches to remove the duplicated entries in their local monitoring table. To remove a monitoring entry, a hash value of the entry is enough to lookup and delete the entry. Therefore, in our implementation, for saving memory for the controller, we simply keep the hash values of the duplicated monitoring entries and temporarily save them into lists, i.e., removing lists, for later removal of the duplicated monitoring entries. Each removing list contains a list of hashes of duplicated monitoring entries of a switch. Each hash value in a removing list has the same number of bytes as an entry hash of a monitoring entry (i.e., 8 bytes). Since only a list of hash values are temporarily stored in each query, we use a list data structure for the implementation of the removing list.

4.6 Evaluation

We evaluate the effectiveness of the proposed mechanism in three aspects: the reduction of monitoring rules the per switch, the balance of monitoring rules assigned to monitoring switches, and the overheads (i.e., elapsed times) of the proposed distributed monitoring algorithm and the monitoring system.

4.6.1 Experiment environment

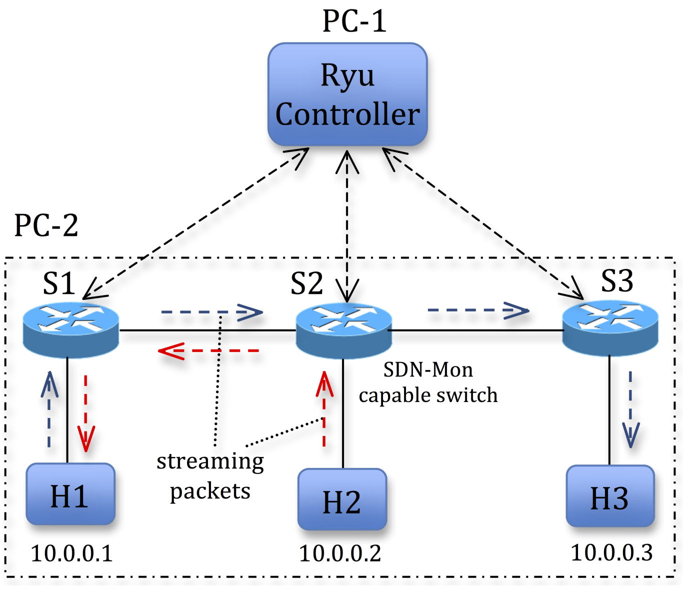


Fig. 4.6 Experiment network.

We conducted experiments with a virtual SDN deployment using VNX [80], as illustrated in Fig. 4.6. The controller is run on PC-1, a physical computer with a 2.66 GHz CPU Intel core 2 Duo E6750 (2 cores) and 4 GiB RAM. Three switches and three hosts are run on PC-2, another physical computer with a 3.4 GHz CPU Intel core i7 (8 cores) and 8 GiB RAM. PC-1 and PC-2 are connected via a LAN network. The switches are run with DPDK v16.11 [15] to enhance its processing speed.

4.6.2 Evaluation on the reduction of monitoring load per switch and the monitoring load balance among multiple switches

The monitoring match fields in SDN-MON switch module are set with 5-tuple consisting of source IP address, source port, destination IP address, destination port, and protocol. The query time interval is set to 10 seconds, which means the controller queries new and updated m-entries from switches in every 10 seconds. Sampling ratio is set to 1.0. We conduct the experiments for two cases: monitoring with the support of our proposed mechanism, and monitoring without the support of our proposed mechanism.

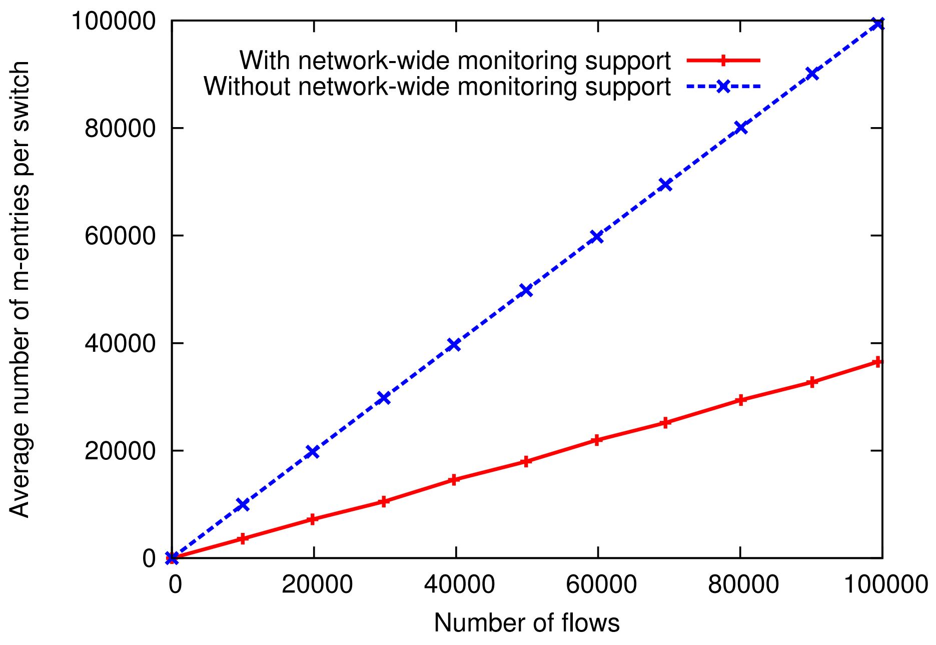


Fig. 4.7 Average number of m-entries per switch in various number of flows.

For evaluating the reduction of monitoring load per switch, we inject a stream of packets from *H1* to *H3* using *tcpreplay* [75]. The dataset for packet injecting is a pcap file from MAWI traffic repository [41], which is a 6.6 GB file of network traffic captured from a real backbone network in Feb. 26, 2017. We measure the number of m-entries installed in each switch in various numbers of 5-tuple flows (from 0 to 100,000 flows). The evaluation results (Fig. 4.7) shows that the average number of m-entries per switch is reduced as over 63%.

For evaluating the monitoring load balance among the switches, we inject two streams of packets from *H1* to *H3*, and from *H2* to *H1* concurrently with two pcap traces (of the same traffic volume), with the same packet injecting speed. We measure the numbers of m-entries in switches and calculate the standard deviation of these numbers. A small standard deviation means that the monitoring load balancing functionality works efficiently. The evaluation results (Fig. 4.8) shows that with the support of the proposed mechanism, the standard deviations are small for all different numbers of flows in the traffic. Thus, the switches are assigned with nearly equal or equivalent numbers of m-entries.

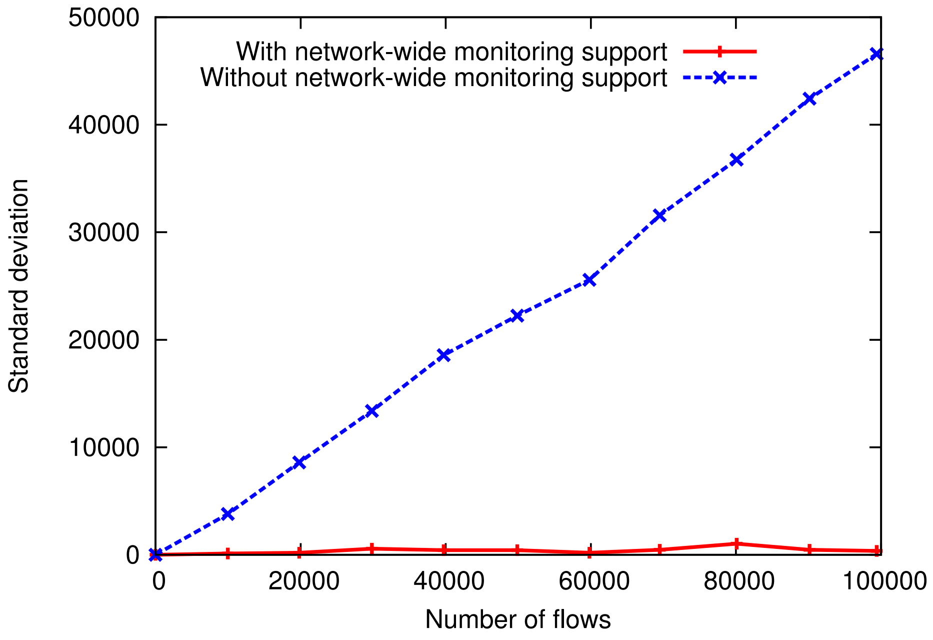


Fig. 4.8 Standard deviation of numbers of m-entries in switches.

4.6.3 Elapsed times of the algorithm and the system

We evaluate the overhead of the proposed mechanism in two aspects: the elapsed time of the processing of the proposed algorithm (called the *algorithm elapsed time*), and the elapsed time of all the processes of a querying time (called the *system elapsed time*). The algorithm elapsed time is calculated by the amount of time for processing all arriving reply messages and the amount of time for putting the m-entries in the buffering table into the *GMTs* and removing the corresponding m-entries in the *RLs*. Let $T1_i$ be the time when the controller receives an SDN-MON multipart message from a monitoring switch (an SDN-MON multipart reply from a switch consists of one multipart message or a sequence of multiple ones depending on the number of m-entries that the switch send to the controller in a querying time), $T2_i$ be the time when the controller completes processing that message, and ΔT be the processing time for putting the m-entries in the buffering table into *GMTs*, removing the corresponding m-entries in *RLs* from related switches, and clearing the buffering table and the *RLs*: $\textit{Algorithm elapsed time} = \sum_{i=1}^{N}(T2_i - T1_i) + \Delta T$, where N is the total number of multipart messages that the controller receives from all monitoring switches in a querying time; The system elapsed time is evaluated by the time interval since the controller sends the first m-entries request message to a switch until it completes processing all reply messages from switches including the algorithm processing.

Number of monitoring entries per query	20,000	40,000	60,000	80,000	100,000
Algorithm elapsed time (s)	0.237	0.436	0.742	0.939	1.181
System elapsed time (s)	0.371	0.868	1.361	1.966	2.692

Table 4.1 Algorithm elapsed time and system elapsed time at various numbers of monitoring entries per query.

We experiment for various numbers of m-entries per query (the total number of m-entries sent from all monitoring switches that the controller receives and processes in a querying time) to observe the algorithm and system elapsed times. We inject a pcap trace containing 634,500 5-tuple flows, which creates 232,950, 234,860, and 234,390 m-entries stored in the switches *S1*, *S2* and *S3* respectively (and corresponding numbers of m-entries stored in the global monitoring tables *GMT-S1*, *GMT-S2*, and *GMT-S3* at the controller). We inject the packets at various injecting rates so that the total numbers of m-entries sent from all monitoring switches to the controller are at various values from 20,000 to 100,000 m-entries. We measure the algorithm and system elapsed times for 5 times in each experiment and calculate average values of them for final results. Table 4.1 shows the elapsed times for processing the algorithm and the system elapsed time in different numbers of m-entries per query from 20,000 to 100,000. The experimental results show that both of the elapsed times of the proposed network-wide monitoring mechanism and the monitoring system are negligible, with maximum elapsed times of the algorithm and the system are small as 1.181 and 2.692 seconds correspondingly even for processing a large number of m-entries as 100,000 m-entries per query, for a large number of 5-tuple flows as 634,500 flows in the injecting traffic.

4.6.4 Evaluation on the scale of the proposed distributed monitoring mechanism

For evaluating the scale of the proposed distributed monitoring algorithm, as the existing tools have limited support for SDN and mostly no support for deploying modified switches, we have built up a simulation module which simulate the input data that the algorithm processes in each query time. The simulation module generate the monitoring entries sent from switches to mimic the input data that the proposed distributed monitoring algorithm received from switches, with three parameters: number of active switch (N), number of active flows sent per switch (M), and duplication ratio (D). Based on the built simulation module, we conducted simulation experiments on different numbers of switches, different numbers of active flows and the different duplication ratios.

We conducted experiments with different numbers of monitoring entries of 1000 to 5000 m-entries for the cases of 10 switches, 100 switches, 500 switches, and 1000 switches. The experimental results (Fig. 4.9 showed that the processing time of the proposed algorithm is trivial for the case of 10 switches with 5000 monitoring active flows monitored in the switches. For the cases of 100 switches, the elapsed time is about 0.3s for 5000 active flows in each switches, which means the algorithm spends 0.3s for processing as large as 500000 active flows per query.

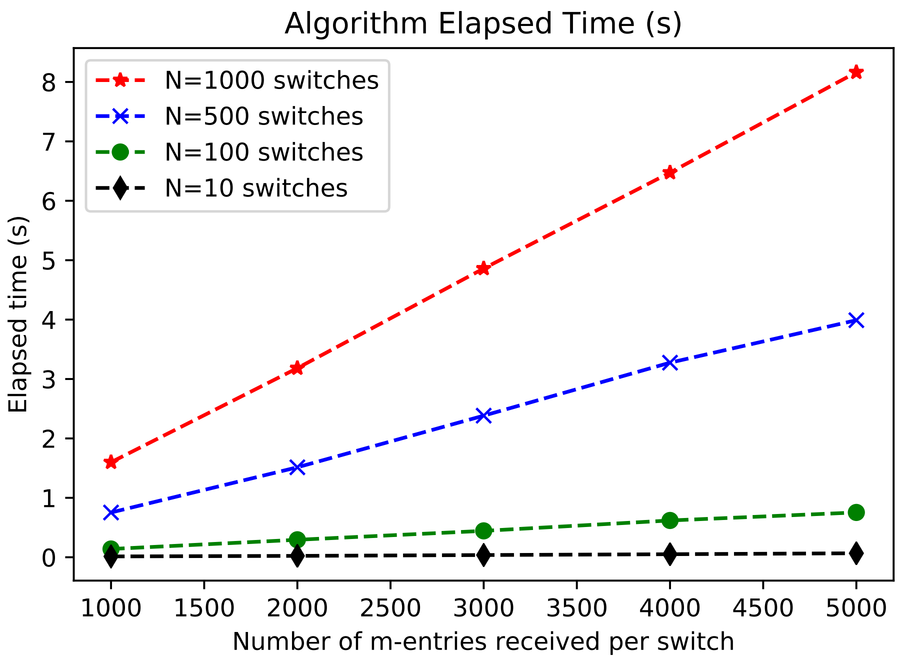


Fig. 4.9 Processing time of the proposed distributed monitoring algorithm at different numbers of monitoring entries per switch

Moreover, we evaluated the elapsed time of the proposed distributed monitoring algorithm in different number of switches. The experimental results (Fig. 4.10 showed a small elapsed time of 1.8s for handling 200 switches where each switch monitors 5000 active flows. Further, we evaluated the elapsed time of the algorithm based on the numbers of active flows with the parameter of duplication ratio. The duplication ratio is set from 0 to 0.8 and the elapsed times for different number of active flows from 200000 to 1000000 active flows were measured. The experimental results (Fig. 4.11 showed a small elapsed time of 1.6s for processing a big number of active flows as 1 million active flows, which is enough to serve different types of networks (i.e., backbone, local area network, campus network, and smaller scaled networks).

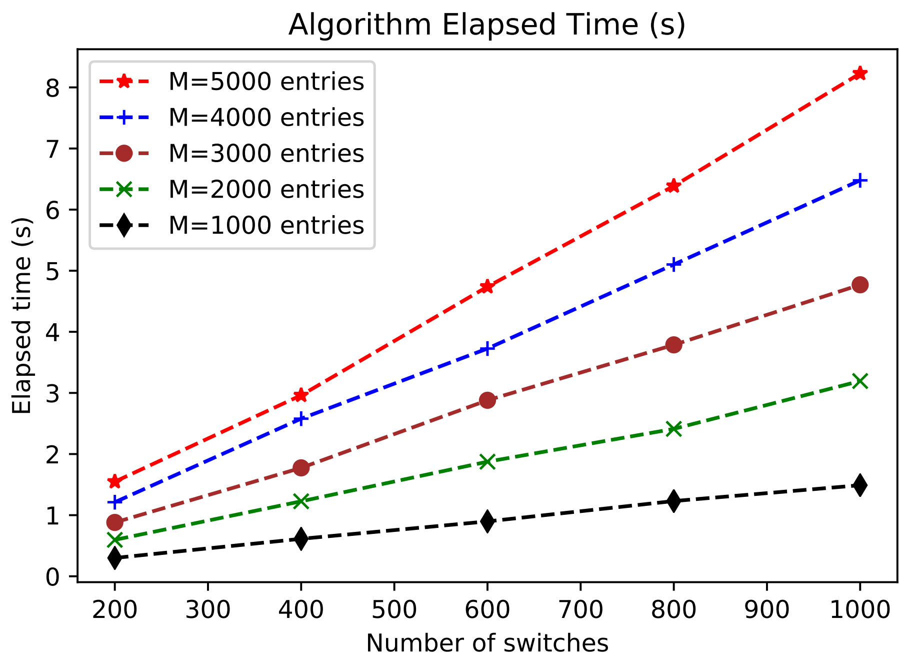


Fig. 4.10 Processing time of the proposed distributed monitoring algorithm at different numbers of switches

In addition, we evaluated the system elapsed time in the network measurement through an estimated calculation from measurement results on data transmission time and simulation results on the processing time of the proposed algorithm. The system elapsed time is estimated from the overall elapsed time of all processing steps including switch processing for gathering data and data transmission time (i.e., τ_1), data parsing time processed by the default controller (i.e., τ_2), and the algorithm elapsed time (i.e., τ_3). Given τ_1, τ_2, τ_3, the system elapsed time is calculated as: $\tau_s = \tau_1 + \tau_2 + \tau_3$.

Let N_s be the number of monitoring switches, M_s be the number of m-entries existed in each switch. As the specified in the proposed method, when a monitoring switch receives a monitoring statistics request, its collect monitoring entries and sends to the controller. This process is conducted concurrently and independently in each of the monitoring switches. In addition, as the controller sends monitoring statistics requests to all monitoring switches at the same time in each query (e.g., through a Python loop), and assuming that each request spends a similar amount of time to reach its destined switch, the switch processing and data transmitting time of N_s switches, i.e. τ_1, can be estimated as the maximum of $\tau_{1,i}$: $\tau_1 = max(\tau_{1,i})$, for i in $[1, N_s]$, where $\tau_{1,i}$ is the switch processing and data transmitting time of a monitoring switch S_i.

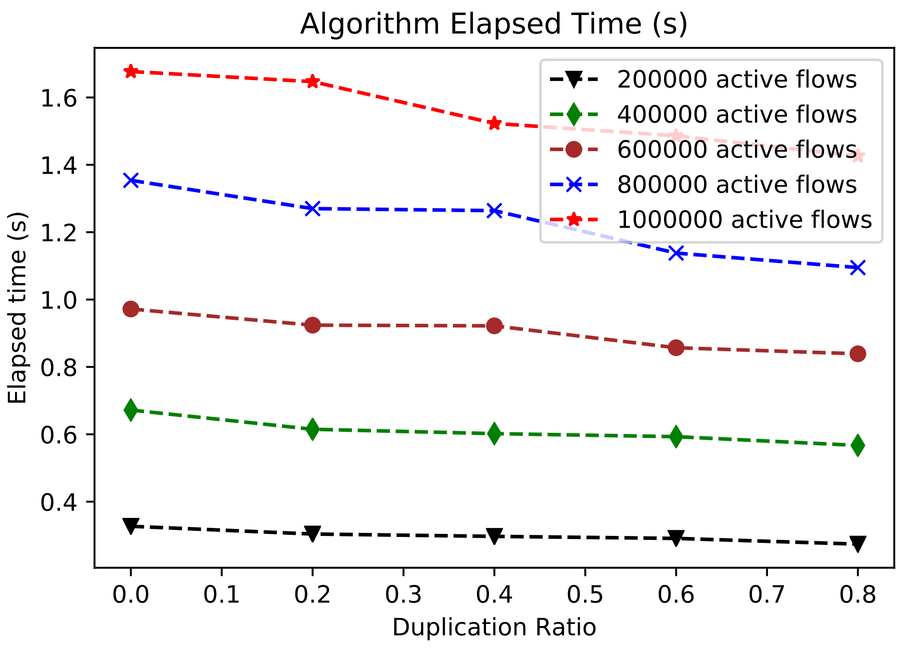


Fig. 4.11 Processing time of the proposed distributed monitoring algorithm in different flow duplication ratios

When a reply message reaches the controller, a default data parsing module in the controller parses the message from byte arrays into data with a defined format of the controller (e.g., Python data object in a Ryu controller). For worse case evaluation, we assume that the data parsing times for parsing multiple reply messages from monitoring switches are sequential. This means that in the worse case, the data parsing time for all reply messages is estimated as: $\tau_2 = \sum_j^K \tau_{2,j}$, where $\tau_{2,j}$ is the data parsing time of a monitoring entry, K is the total number of monitoring entries received from switches, i.e., $K = N_s$ x M_s. Therefore, the system elapsed time is estimated as: $\tau_s = max(\tau_{1,i}) + \sum_j^K \tau_{2,j} + \tau_3$, for i in [1, N_s], j in [1, K], $K = N_s$ x M_s, where: $\tau_{1,i}$ is the switch processing and data transmitting time of each monitoring switch S_i, $\tau_{2,j}$ is the data parsing time of a monitoring entry among all monitoring entries that the controller received from switches, and τ_3 is the processing time of the algorithm for all received monitoring entries.

As each monitoring entry has the same structure and number of bytes, it is reasonable to assume that the controller spend an equivalent amount of time for parsing each monitoring entry. Therefore, the data parsing time for all monitoring entries received from N_s switches, where each switch sends M_s monitoring entries, can be estimated as: $\tau_s = max(\tau_{1,i}) + N_s$ x $\tau_{2,i} + \tau_3$, for i in [1, N_s], where: $\tau_{1,i}$ is the switch processing and data transmitting time of each monitoring switch S_i, $\tau_{2,i}$ is the data parsing time of all monitoring entries that the controller received from a monitoring switch, and τ_3 is the processing time of the algorithm for all received monitoring entries.

We conducted experiments to measure: (1) the elapsed time for data gathering at switch and data transmitting from switches to controller (i.e., $\tau_{1,i}$), and (2) the elapsed time for data parsing (i.e., $\tau_{2,i}$). The processing time of the algorithm for all received monitoring entries, τ_3 is measured through simulation as indicated above. Fig. 4.13 and Fig. 4.12 showed the elapsed time of corresponding processing steps. The evaluations results (as showed in Fig. 4.9, Fig. 4.10, Fig. 4.11, Fig. 4.12, and Fig. 4.13) indicated that the architecture can support a few tens of switches for busy networks with a number of active flows up to a hundred thousands active flows, and it can support a few hundreds of switches for less busy networks with a number of active flows about a few thousands or a few tens of thousands of active flows. The system elapsed time can be calculated with the above mentioned formula. For instance, for monitoring a million active flows, τ_3 is about 1.6s, the τ_1 is about 0.45s (as the data transmission and switch processing time is about 45ms as showed in Fig. 4.13), τ_2 is about 5.15s (as the data transmission and switch processing time is about 515ms as showed in Fig. 4.12), the system elapsed time will be about 7.2s.

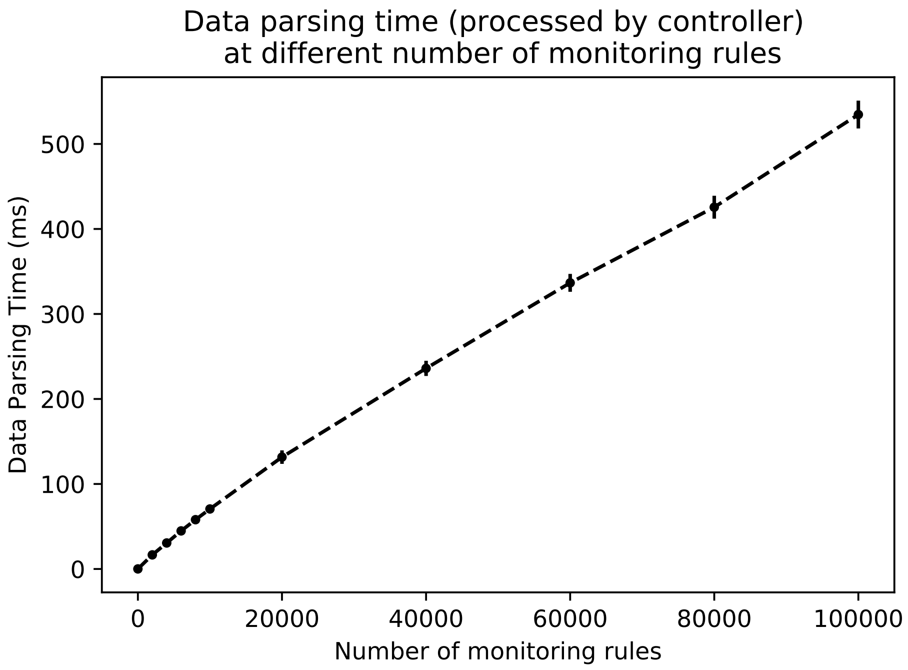


Fig. 4.12 Data parsing time (processed by default controller) at different number of monitoring rules

With the conducted experimental results, the scale of network of the architecture is validated where overall elapsed time (including switch processing for gathering data, data

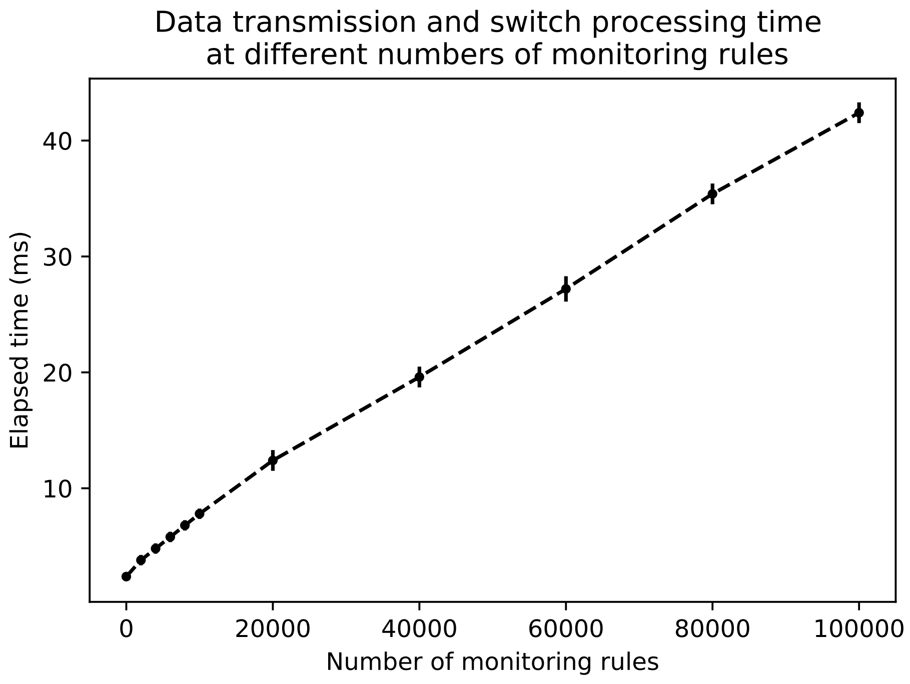


Fig. 4.13 Data transmission and switch processing time at different number of monitoring rules

transmission to the controller, data parsing by the default data parsing step of controller, and the processing time of the proposed algorithm) is about a second for monitoring a hundred thousands of active flows int the network. The processing time of the proposed distributed algorithm is proved to be small (e.g., about 1.6 second for processing a million active flows as in Fig. 4.11). The results showed its capability of serving the target networks with over few tens of switches for busy networks with up to a hundred thousands of active flows (e.g., backbone network), and a few hundreds thousands of switches for less busy networks with up to a few thousands of active flows in the network (i.e., local area network, campus network, and smaller scaled networks).

4.7 Discussion

As the target of this dissertation is proposing traffic monitoring method for OpenFlow-based SDN that reduces monitoring overhead for all network elements including switches, controller and control channel, we discuss the previous pros and cons of previous works as whole monitoring systems for all related aspects, and compare to our proposed platform in this section. UMON [81] reduces monitoring overhead in switch and support fine-grained monitoring. However, it does not discuss any mechanism for monitoring in a distributed scenario with multiple switches. Thus, redundant/duplicated monitoring rules are not eliminated from switches that results in significant overhead in switches caused by the redundant rules. Thus, for distributed monitoring with multiple switches, the overhead in UMON system will be higher than our proposed SDN-MON platform.

The aggregation based approach in OpenWatch [88] reduced overhead by adapting flow aggregation level (i.e., adjusting monitoring granularity). However, aggregation mechanism may not benefit for network applications that requires fine-grained statistics of all flows (e.g., traffic classification, anomaly detection). Furthermore, as the approach does not discuss reducing overhead per monitoring rule in switch, the overhead in switch for monitoring a same number of flows in OpenWatch will be higher than our proposed platform. In FlowCover [70] and OpenTM [77], the overhead in control channel is reduced with different proposed switch selection schemes. However, these proposals do not discuss reducing overhead per monitoring rule, and the redundant/duplicated rules are not eliminated in switches. Thus, overhead in these proposals will be higher than SDN-MON for monitoring same number of flows. In terms of system processing overhead, the system elapsed time of FlowCover is about 2.5 seconds for monitoring 100000 active flows, while the system elapsed time in SDN-MON for monitoring the same number of active flows is about a second as showed by experimented results in this chapter. Thus, the system overhead of SDN-MON is lower than FlowCover. In Payless [9] and OpenNetMon [78], the monitoring overhead is reduced by adapting frequency of statistics polling with traffic busy status. These works do not discuss reducing overhead per monitoring rule and simply reply on OpenFlow events for monitoring. Thus, overhead for monitoring same number of flows will be higher than our proposed platform. In addition, by passively capturing OpenFlow events, it may not benefit for fine-grained or real time monitoring requirements. Table 4.2 summarizes a comparison between the closest work and our proposal in the aspect of reducing monitoring overhead in distributed scenarios.

4.8 Summary

This chapter presents our proposed mechanism to support SDN to monitor over multiple switches network in a distributed fashion (called network-wide monitoring mechanism). The mechanism is designed and implemented on the proposed SDN-MON platform, to enable a monitoring capability for OpenFlow-based SDN for fine-grained and distributed monitoring. The mechanism distributes the monitoring tasks, which are represented by monitoring entries, over multiple monitoring switches in the network and balance these tasks among switches.

Method	Overhead per rule reduction	Distributed monitoring overhead reduction	Network component	Integrated API	Fine-grained monitoring
FlowCover [70]	No	Yes	Controller	Yes	Yes
OpenWatch [88]	No	Yes	Controller	Yes	Not well supported
OpenTM [77]	No	Yes	Controller	Yes	Yes
Payless [9]	No	Yes	Controller	Yes	Not well supported
OpenSample [72]	Yes	Yes	External deployment	No	Yes
FlowSense [87]	No	Yes	Controller	Yes	Not well supported
OpenNetMon [78]	No	Yes	Controller	Yes	Not well supported
SDN-MON (ours)	Yes	Yes	Switch and Controller	Yes	Yes

Table 4.2 Comparison of our proposal with existing traffic monitoring methods in both aspects: monitoring overhead reduction at switch and overhead reduction in distributed monitoring scenario

This mechanism benefits in intelligently leveraging idle computing/memory resources at switches for monitoring, and allows much less monitoring load/overhead at each single switch. The experimental results, which are conducted with real traffic traces, validate the reduction of monitoring load per switch corresponding to the duplication ratio (e.g., over 63% for a three switches network scenario). Through the experiments, the balance of monitoring load among the switches is also validated where the results show small standard deviations on the numbers of monitoring entries assigned to switches. The elapsed times of the proposed mechanism and the monitoring system are also proved to be small (e.g., about a second and 2.7 seconds respectively for monitoring a large number active flows as a hundred thousands active flows, where the number of flows in the experimental traffic is as large as over a half million flows.

In addition, the effectiveness in terms of the scale of the proposed distributed monitoring proposal was also evaluated to validate its capability to serve the targeted networks and applications. With the conducted experimental results, the scale of network of the architecture is validated where the processing time of the proposed distributed algorithm and system elapsed time are proved to be small. For instance, the overall elapsed time (including switch processing for gathering data, data transmission to the controller, data parsing by the default data parsing step of controller, and the processing time of the proposed algorithm) for monitoring a hundred thousands of active flows is about a second. The experimental results also showed a small algorithm elapsed time, e.g., about 1.6 second for processing a million active flows per query. As the system elapsed time is a round a seconds for monitoring a hundred thousands of active flows, the proposed method satisfies the requirements of target networks and applications in terms of required number of active flows (i.e., thousands or a hundred thousands active flows) and required query time window (i.e., in an order of seconds or minutes). In addition, as the proposed method can server over a few tens of switches as discussed above, the requirement of target networks and applications about number of switches (i.e., a few or a few tens of switches) is also satisfied. Since the proposed monitoring method satisfies the specified requirements of targeted networks and applications (as outlined in section 4.1 of this chapter), it is considered to be capable of serving the targeted networks and applications.

Chapter 5

Discussion

5.1 Application examples

In this section, we discuss example traffic applications that can be developed on top of the SDN-MON. Since SDN-MON integrates fine-grained monitoring capability into OpenFlow-based SDN, it facilitates the implementation of traffic engineering applications since the monitoring and control are operated consistently as a whole system in a SDN controller. Typically, a traffic engineering application uses SDN-MON APIs to obtain fine-grained statistics of traffic flows in the network for its analysis. Once a routing decision or a traffic engineering policy is made, the application uses OpenFlow-based APIs to install flow entries into switches to enforce the new routing rules or policies. SDN-MON platform can be used for a variety of traffic engineering applications such as heavy-hitter detection, network bandwidth measurement and load balancing, traffic classification, traffic matrix estimation, routing optimization.

As an example, heavy-hitter detection and mitigation can be implemented on top of SDN-MON to detect traffic flows that have a very large number of packets or to find the set of flows contributing significant amounts of traffic to a link. Heavy-hitter detection benefits a number of network management applications, e.g., relieving link congestion [4], planning network capacity [19], or caching forwarding table entries [62] for efficient routing of traffic. Identifying heavy hitters at small time scales enables dynamic routing of heavy flows [12] and dynamic flow scheduling [66]. In order to detect heavy-hitters, fine-grained statistics of traffic flows are required for analysis and detection. The required flow statistics consists of number of packets and number of bytes in flows, where the required monitoring granularity of the flows is 5-tuple based granularity including specified header fields: source IP address, source port number, destination IP address, destination port number, and protocol number [46].

Rank	Source IP	Source Port	Destination IP	Destination Port	Protocol	Bytes	Percentage
1	192.168.2.104	1208	65.75.343.170	23560	TCP	929.4 MBytes	10%
2	62.146.10.41	80	192.168.2.104	3932	TCP	743.2 MBytes	8%
3	192.168.1.101	80	192.168.1.150	4497	TCP	504.3 MBytes	6%
4	217.6.164.162	80	192.168.2.104	4220	TCP	365.7 MBytes	5%
5	192.168.1.101	80	192.168.1.150	4497	TCP	235.2 MBytes	3%
...	...	...	...	...	...	...	...

Table 5.1 Example data of detected heavy-hitter flows.

As the required flow statistics of a heavy-hitter detection and mitigation application are exactly the flow statistics and monitoring granularity that SDN-MON supports, to implement the heavy-hitter detection, users or network operators can write a simple Python application/program at the controller that use SDN-MON APIs and initialize its monitoring process. By using the SDN-MON APIs, the application can set a suitable query time window (e.g., 5 seconds) by sending a Set query time window message to monitoring switches to get the fine-grained statistics. Periodically, SDN-MON monitoring modules at switches report the fine-grained flow statistics to the application. The receiving monitoring data at the controller will be parsed by SDN-MON controller module from the bytes format to usable format of monitoring entries, eliminating duplication, and storing flow statistics. Based on the monitoring data, the traffic engineering can simply calculate/count number of bytes or packets of the traffic flows to determine heavy-hitter flows in the network, and to determine available route of source-destination end hosts for rerouting traffic. As an example, detected heavy-hitter flows can also be ranked and visualized as Table 5.1. Once new routing rules are determined, the traffic engineering application simply use OpenFlow APIs to install corresponding flow entries in switches for traffic rerouting to mitigate/eliminate heavy-hitter flows. In case there is not other available link to reroute the flow from its source host to destination host, the application may drop incoming traffic of the flow to avoid link congestion. As the SDN-MON is integrated to SDN/OpenFlow as a same platform on controller, such application workflow can be implemented on SDN-MON platform via a few tens of lines of Python code. Algorithm 2 presents an example pseudo code of the application.

Since other traffic engineering applications such as routing optimization [21] [61], traffic classification [5], require the same flow statistics and monitoring granularity level, they can also be implemented using SDN-MON platform with a similar programming workflow within

```
Send Monitoring Mode On messages to SDN-MON switches;
continue-monitoring = True;
query-time-window = 5;
(set query time window of 5 seconds) network-topo = NetworkTopology();
network-topo.discover();
switch-IDs = network-topo.get-switches();
while continue-monitoring do
    for switch-Id in switch-IDs do
        send-SDNMON-Monitoring-Data-Request(switchId);
        sleep(query-time-window);
    end
end
heavy-hitters = [];
heavy-hitter-count = 5;
min-number-of-bytes = 0;
min-index = 0;
for switchId in switch-IDs do
    global-monitoring-table = global-monitoring-tables[switch-Id];
    for hash in global-monitoring-table.key() do
        if (global-monitoring-table[hash][byte-count] >= min-number-of-bytes) then
            heavy-hitters.remove(min-index);
            heavy-hitters.append(global-monitoring-table[hash]);
            update(min-number-of-bytes);
            update(min-index);
        end
    end
end
top-heavy-hitter = heavy-hitters[0];
network-links = network-topo.get-links();
reroute-link = find-reroute-link(top-heavy-hitter, network-links);
if (reroute-link) then
    install new flow entries to switches of the reroute link;
    remove old flow entries;
end
else
    send OF-Flow-Mod message to switches of the top heavy hitter flow to drop incoming
      traffic;
end
```

Algorithm 2: Example pseudocode of heavy hitter detection and mitigation application.

a few tens of lines of Python code. These example applications indicate that SDN-MON, with integrating fine-grained monitoring capability to OpenFlow-based software switch, facilitates implementation of common traffic engineering applications that benefit network management.

5.2 Applicability for designed networks and applications

The experimental results on monitoring performance at switch as showed in chapter 3 has demonstrated a low overhead of SDN-MON for monitoring fine-grained traffic flows. In addition, the experiment results for distributed scenarios in chapter 4 shows a reduction of monitoring load in each switch depending on flow duplication ratio, while monitoring load are balanced among switches. The evaluation also proves small processing times of the distributed monitoring mechanism and the monitoring system (e.g., for 100,000 active flows, switch processing and data transmitting time is less than 50 ms, and data transmission time is about 550 ms, so the overall elapsed time is about 600 ms). Specifically, the processing time of the distributed algorithm is small as around 0.3 second for processing 200,000 active flows and 1.6 second for processing a million flows correspondingly. Moreover, the evaluation in terms of scalability has showed that SDN-MON is capable of serving tens or even hundreds of switches depending on business of network (represented by the number of active flows in a certain time window). Together with requirements of different networks (i.e., LAN, Campus network, Backbone network, and smaller scaled networks) and applications (e.g., heavy-hitter detection, bandwidth monitoring, traffic classification, network diagnosis of troubleshooting) where the numbers of active flows is around or below 100,000 active flows in time windows of 5s or above, the evaluation results indicates that SDN-MON is considered to be capable of serving those networks and applications.

5.3 Limitations

The focus of this dissertation is proposing the monitoring method for OpenFlow-based SDN with small overhead so that SDN users or network operators can monitor traffic within the existing infrastructure (i.e., SDN switches and controller). Although our proposal has demonstrated its low overhead with reducing overhead per rule for switches, reducing overhead in switches and in the network by detecting and eliminating redundant monitoring rules and assigning rules to switches in a balancing fashion, the proposal and its current implementation has several limitations. Firstly, the proposal requires installing our extension modules for both switch and controller to use SDN-MON monitoring method. Secondly, the applicability of the proposal on hardware switch has not been investigated, thus the current implementation mostly works for software switch. We consider to investigate the applicability of our proposal on hardware switch for our future works. Thirdly, the current implementation instance of SDN-MON uses exact 5 tuple for match fields. With 5 tuple based match fields, network traffic is monitored with high granularity (i.e., fine-grained) in SDN-MON and the exact match also contributes to the processing speed of the monitoring process in switch. However, if SDN-MON introduces wildcard match by adding more sophisticated matching algorithm at switch, it causes high overhead due to the complicated flow matching algorithm and in the worst case, the performance may be similar to Lagopus. Also, if SDN-MON handles a wildcard by summing up individual monitoring entries matched to the wildcard at switch (then merge at controller), it doesn't scale for general rules. For example, in case of monitoring with one wildcard rule (i.e., one flow entry in switch), the switch performance of Lagopus is about 8 Mpps, while the corresponding performance for SDN-MON with 1000 rules without wildcard (i.e., 1000 monitoring entries) and one wildcard forwarding flow entry is about 7.4 Mpps, which is smaller than the original switch. Therefore, if wildcard is implemented in SDN-MON that can accept aggregations of the traffic flows (i.e., coarse-grained monitoring, such as origin-destination pair), the performance of the monitoring process in SDN-MON switch is decreased accordingly and in worse case it may be similar to performance of the original switch. Fourly, although our proposal has demonstrated its low overhead for monitoring in OpenFlow-based software switch, the limited resources at switches may limit its use cases, especially for busier networks (e.g., MAN, WAN), and applications that require monitoring larger number of active flows within smaller query time window (e.g, for anomaly detection, depending on the scale of network and number of hosts, if the number of active flows required to be monitored is over a hundred thousands flows in a time window of less than a second, more than one switch may be required to monitor the flows for the detection analysis). However, with the constraint of in-band monitoring within the existing SDN infrastructure, our proposal has showed its applicability for a number of common networks (i.e., LAN, Campus network, Backbone network, and smaller scaled networks) and a number of common traffic engineering applications (e.g., heavy-hitter detection, traffic classification, bandwidth monitoring, network diagnosis of troubleshooting) as discussed in the previous chapters.

Chapter 6

Conclusion

6.1 Dissertation summary

In this dissertation, we studied the traffic monitoring problem in SDN/OpenFlow for fine-grained traffic engineering. Since traffic engineering requires both monitoring traffic flow statistics and controlling network traffic, integrating monitoring and controlling in the same platform/APIs are essentially important. Therefore, in our proposal, we aim at integrating a fine-grained monitoring capability to OpenFlow-based SDN to facilitate the implementation of traffic engineering applications and benefit network management. We target our proposal to be capable of monitoring common networks (i.e., LAN, campus network, backbone network, and smaller scaled networks) and fine-grained traffic engineering applications (e.g., heavy hitter detection, traffic classification, bandwidth monitoring, routing optimization). Through an empirical study presented in chapter 2, we outline the requirements of the target networks and applications. The requirements consists of: (1) Capability of reducing overhead per monitoring rule in switch; (2) Capability of serving a few thousands or a hundred thousands of active flows; (3) Processing time of the monitoring method must be in an order of seconds for such number of active flows; (4) Capability to serve ten or a few tens of monitoring switches in the network for distributed monitoring scenarios.

As a switch typically has limited resources (e.g., throughput, CPU, memory) while fine-grained monitoring requires monitoring large number of flows, the first critical goal for our proposal is to reduce overhead per monitoring rule in OpenFlow based SDN switch. To meet this goal, we proposed a traffic monitoring method that can reduce overhead per monitoring rule in switch, thus reduce monitoring overhead in switch. We designed lightweight monitoring modules and process for switch that is operated independently from forwarding functionality to achieve the overhead reduction.

Furthermore, in most networks, multiple switches independently monitor flows that results in duplication of flows monitoring and produce overhead in switches and the network as analyzed in chapter 1. Specifically, when the number of flows traversing through the multiple monitoring switches is large, the duplication may become large accordingly that results in significant overhead in switches. As the resources (e.g., throughput, memory, CPU) in switches are limited, reducing overhead caused by the monitoring duplication is important to enable monitoring in distributed scenarios with reduced overhead. Therefore, we additionally propose a distributed monitoring method for traffic monitoring in distributed scenarios with multiple monitoring switches. We design a monitoring method that can detect and eliminate the duplicated monitoring rules and distribute monitoring load over multiple monitoring switches in a balancing fashion. We also design a dedicated protocol for communication between switches and controller and integrate it into OpenFlow protocol for exchanging monitoring data and instructions in the proposed monitoring platform.

We implemented the proposed methods as a systematic flows monitoring platform for OpenFlow-based SDN called SDN-MON. We conduct a number of experiments based on the implementation instance of the proposals, and demonstrate the effectiveness of our proposals with experimental results in both aspects: (1) monitoring performance at switch (in terms of throughput performance of the switch), and (2) performance of the distributed monitoring method (in terms of processing times of the distributed monitoring algorithm and the overall monitoring process). Experimental results demonstrated that SDN-MON achieves better performance at switch (i.e., higher throughput, or lower overhead) than the default OpenFlow based SDN monitoring mechanism. Therefore, the requirements on reducing overhead per monitoring rule is satisfied. Furthermore, experimental results on the distributed monitoring aspect show small algorithm elapsed times, e.g., about 1.6 second for processing a million active flows per query. As the system elapsed time is a round a seconds for monitoring a hundred thousands of active flows, the proposed method satisfies the requirements of target networks and applications in terms of required number of active flows (i.e., thousands or a hundred thousands active flows) and required query time window (i.e., in an order of seconds or minutes). In addition, as the proposed method can server over a few tens of switches as discussed above, the requirement of target networks and applications about number of switches (i.e., a few or a few tens of switches) is also satisfied. As the proposed monitoring method satisfies the specified requirements of targeted networks and applications (the requirements (1) to (4) as outlined above), it is considered to be applicable for targeted networks and fine-grained traffic engineering applications.

6.2 Future Work

These are categories of works needed to be done in the future.

Investigating the applicability of the proposal on hardware switches Our prototype implementation of the proposal was showed to work efficiently on software switch. The applicability of the proposal for hardware switch may require some refinement of the implementation to work on a hardware switch. This possibility has not tested. We consider this as a future direction of our work.

Deploying and experimenting the proposal on a physical network environment. The monitoring on multiple switches scenario was experimented in a simulation environment with multiple switches running in a physical machine due to a lack of physical devices for experiments. Although the evaluation results is considerably reliable due to the reliability of the simulation environment, it would be even more convincing to show the performance in that aspect on a physical deployment.

Making the implementation of the proposed framework more efficient. The experimental results on the performance of the current implementation of our proposal have showed a promising applicability to production networks such as LANs, CANs or even data centers. However, it still has room for performance improvement as the implementation might not be an optimal one in terms of programming and engineering. We consider this as another future direction of our work.

Making more programs and applications to be run on the proposed framework. Current applications written on top of the proposed monitoring framework is still simple. Therefore, making more programs and applications, even complicated ones, would show the applicability of the framework and interest community to using it better. This is considered as another future direction of our work.

References

[1] Davide Adami, Barbara Martini, Molka Gharbaoui, Piero Castoldi, Gianni Antichi, and Stefano Giordano. Effective resource control strategies using openflow in cloud data center. In *IFIP/IEEE IM 2013*, pages 568–574. IEEE, 2013.

[2] Ian F Akyildiz, Ahyoung Lee, Pu Wang, Min Luo, and Wu Chou. A roadmap for traffic engineering in sdn-openflow networks. *Computer Networks*, 71:1–30, 2014.

[3] Mohammad Banikazemi, David Olshefski, Anees Shaikh, John Tracey, and Guohui Wang. Meridian: an sdn platform for cloud network services. *IEEE Communications Magazine*, 51(2):120–127, 2013.

[4] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *ACM CoNEXT'11*, page 8. ACM, 2011.

[5] Laurent Bernaille, Renata Teixeira, Ismael Akodkenou, Augustin Soule, and Kave Salamatian. Traffic classification on the fly. *ACM SIGCOMM Computer Communication Review*, 36(2):23–26, 2006.

[6] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[7] Jeffrey D Case, Mark Fedor, Martin L Schoffstall, and James Davin. Simple network management protocol (SNMP). RFC1157, 1990.

[8] Marco Chiesa, Christoph Dietzel, Gianni Antichi, Marc Bruyere, Ignacio Castro, Mitch Gusat, Thomas King, Andrew W Moore, Thanh Dang Nguyen, Philippe Owezarski, et al. Inter-domain networking innovation on steroids: empowering ixps with sdn capabilities. *IEEE Communications Magazine*, 54(10):102–108, 2016.

[9] Shubhajit Roy Chowdhury, M Faizul Bari, Rizwan Ahmed, and Raouf Boutaba. Payless: A low cost network monitoring framework for software defined networks. In *IEEE NOMS'14*, pages 1–9, 2014.

[10] Benoit Claise. Cisco systems netflow services export version 9. RFC3954, 2004.

[11] Benoit Claise. Specification of the ip flow information export (ipfix) protocol for the exchange of ip traffic flow information. RFC5101, 2008.

[12] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. DevoFlow: scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 254–265, 2011.

[13] Anupam Das, Cristian Lumezanu, Yueping Zhang, Vishal K Singh, Guofei Jiang, and Curtis Yu. Transparent and flexible network management for big data processing in the cloud. In *HotCloud'13*, page 6. Usenix, 2013.

[14] Xenofontas Dimitropoulos, Paul Hurley, and Andreas Kind. Probabilistic lossy counting: an efficient algorithm for finding heavy hitters. *ACM SIGCOMM Computer Communication Review*, 38(1):5–5, 2008.

[15] DPDK: Data plane development kit. http://dpdk.org/.

[16] David Erickson. The beacon openflow controller. In *HotSDN'13*, pages 13–18. ACM, 2013.

[17] Cristian Estan, George Varghese, and Mike Fisk. Bitmap algorithms for counting active flows on high speed links. In *ACM IMC'03*, pages 153–166. ACM, 2003.

[18] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *ACM CoNEXT'14*, pages 75–88. ACM, 2014.

[19] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. Deriving traffic demands for operational ip networks: Methodology and experience. *IEEE/ACM Transactions on Networking*, 9(3):265–280, 2001.

[20] Floodlight Controller. http://www.projectfloodlight.org/floodlight/.

[21] Bernard Fortz, Jennifer Rexford, and Mikkel Thorup. Traffic engineering with traditional ip routing protocols. *IEEE communications Magazine*, 40(10):118–124, 2002.

[22] Éric Fusy and Frécéric Giroire. Estimating the number of active flows in a data stream over a sliding window. In *Proceedings of the Meeting on Analytic Algorithmics and Combinatorics*, pages 223–231. Society for Industrial and Applied Mathematics, 2007.

[23] Gagandeep Garg and Roopali Garg. Detecting anomalies efficiently in sdn using adaptive mechanism. In *IEEE ACCT'15*, pages 367–370. IEEE, 2015.

[24] Molka Gharbaoui, Barbara Martini, Davide Adami, Gianni Antichi, Stefano Giordano, and Piero Castoldi. On virtualization-aware traffic engineering in openflow data centers networks. In *IEEE NOMS'14*, pages 1–8. IEEE, 2014.

[25] Kostas Giotis, Christos Argyropoulos, Georgios Androulidakis, Dimitrios Kalogeras, and Vasilis Maglaris. Combining openflow and sflow for an effective and scalable anomaly detection and mitigation mechanism on sdn environments. *Computer Networks*, 62:122–136, 2014.

[26] Karyna Gogunska, Chadi Barakat, Guillaume Urvoy-Keller, and Dino Lopez-Pacheco. On the cost of measuring traffic in a virtualized environment. In *IEEE CloudNet'18*, pages 1–6. IEEE, 2018.

[27] Google Inc. Inter-datacenter WAN with centralized TE using SDN and OpenFlow. https://www.opennetworking.org/wp-content/uploads/2013/02/cs-googlesdn.pdf, 2012.

[28] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[29] Anton Gulenko, Marcel Wallschläger, and Odej Kao. A practical implementation of in-band network telemetry in open vswitch. In *IEEE CloudNet'18*, pages 1–4. IEEE, 2018.

[30] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean P Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. Sdx: A software defined internet exchange. *ACM SIGCOMM Computer Communication Review*, 44(4):551–562, 2015.

[31] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015.

[32] Hongxin Hu, Wonkyu Han, Gail-Joon Ahn, and Ziming Zhao. FLOWGUARD: building robust firewalls for software-defined networks. In *ACM HotSDN*, pages 97–102. ACM, 2014.

[33] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14, 2013.

[34] Abdesselem Kortebi, Luca Muscariello, Sara Oueslati, and James Roberts. Evaluating the number of active flows in a scheduler realizing fair statistical bandwidth sharing. In *ACM SIGMETRICS Performance Evaluation Review*, volume 33, pages 217–228. ACM, 2005.

[35] Ian Ku, You Lu, and Mario Gerla. Software-defined mobile cloud: Architecture, services and use cases. In *IEEE IWCMC'14*, pages 1–6. IEEE, 2014.

[36] Ian Ku, You Lu, Mario Gerla, Francesco Ongaro, Rafael L Gomes, and Eduardo Cerqueira. Towards software-defined vanet: Architecture and services. In *MED-HOC-NET'14*, pages 103–110. IEEE, 2014.

[37] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, et al. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 1–14. ACM, 2015.

[38] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. What you need to know about sdn flow tables. In *PAM'15*, pages 347–359. Springer, 2015.

[39] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *HotNets'10*, page 19. ACM, 2010.

[40] Jing Liu, Jie Li, Guochu Shou, Yihong Hu, Zhigang Guo, and Wei Dai. Sdn based load balancing mechanism for elephant flow in data center networks. In *IEEE WPMC'14*, pages 486–490. IEEE, 2014.

[41] MAWI Traffic Repository. http://mawi.wide.ad.jp.

[42] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[43] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *IEEE WoWMoM'14*, pages 1–6. IEEE, 2014.

[44] Wang Miao, Fernando Agraz, Shuping Peng, Salvatore Spadaro, Giacomo Bernini, Jordi Perelló, Georgios Zervas, Reza Nejabati, Nicola Ciulli, Dimitra Simeonidou, et al. Sdn-enabled ops with qos guarantee for reconfigurable virtual data center networks. *Journal of Optical Communications and Networking*, 7(7):634–643, 2015.

[45] Traffic monitoring using sFlow. http://www.sflow.org/sflowoverview.pdf.

[46] Tatsuya Mori, Tetsuya Takine, Jianping Pan, Ryoichi Kawahara, Masato Uchida, and Shigeki Goto. Identifying heavy-hitter flows from sampled flow statistics. *IEICE Transactions on Communications*, 90(11):3061–3072, 2007.

[47] Andrew C Myers and Andrew C Myers. Jflow: Practical mostly-static information flow control. In *ACM POPL'99*, pages 228–241. ACM, 1999.

[48] Ankur Kumar Nayak, Alex Reimers, Nick Feamster, and Russ Clark. Resonance: dynamic access control for enterprise networks. In *WREN'09*, pages 11–18. ACM, 2009.

[49] Open Networking Foundation . OpenFlow Switch Specification version 1.5.0. 2014.

[50] Open Networking Foundation. How OpenFlow-based SDN transforms private cloud. *ONF Solution Brief*, 2012.

[51] Open Networking Foundation. Software-defined networking: The new norm for networks. *ONF White Paper*, 2:2–6, 2012.

[52] Open Networking Foundation. SDN architecture overview. *ONF White Paper*, 2013.

[53] Open Networking Foundation. SDN in the campus environment. *ONF Solution Brief*, 2013.

[54] Open Networking Foundation. SDN security considerations in data center. *ONF Solution Brief*, 2013.

[55] Open vSwitch. http://www.openvswitch.org/.

[56] Peter Phaal and Marc Lavine. sflow version 5. *white paper*, 2004.

[57] Peter Phaal, Sonia Panchen, and Neil McKee. InMon Corporation's sFlow: A Method for Motoring Traffic in Switched and Routed Networks. RFC3176, 2001.

[58] Xuan Thien Phan and Kensuke Fukuda. SDN-Mon: Fine-grained traffic monitoring framework in software-defined networks. *Journal of Information Processings (JIP)*, 25: 182–190, Feb. 2017.

[59] POX Controller. http://www.noxrepo.org/pox/about-pox/.

[60] Reza Rahimi, Malathi Veeraraghavan, Yoshihiro Nakajima, Hirokazu Takahashi, S Okamoto, and N Yamanaka. A high-performance openflow software switch. In *IEEE HPSR'16*, pages 93–99. IEEE, 2016.

[61] Anton Riedl and Dominic A Schupke. Routing optimization in ip networks utilizing additive and concave link metrics. *IEEE/ACM transactions on networking*, 15(5): 1136–1148, 2007.

[62] Ori Rottenstreich and János Tapolcai. Optimal rule caching and lossy compression for longest prefix matching. *IEEE/ACM Transactions on Networking*, 25(2):864–878, 2017.

[63] Ryu SDN Framework. http://osrg.github.io/ryu/.

[64] Sakir Sezer, Sandra Scott-Hayward, Pushpinder-Kaur Chouhan, Barbara Fraser, David Lake, Jim Finnegan, Niel Viljoen, Mary Miller, and Neeraj Rao. Are we ready for SDN? Implementation challenges for software-defined networks. *IEEE Communications Magazine*, 51(7):36–43, 2013.

[65] Fernando Silveira, Christophe Diot, Nina Taft, and Ramesh Govindan. Astute: Detecting a different class of traffic anomalies. *ACM SIGCOMM Computer Communication Review*, 41(4):267–278, 2011.

[66] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *ACM SIGCOMM Computer Communication Review*, pages 44–57. ACM, 2016.

[67] Robin Sommer and Anja Feldmann. Netflow: Information loss or win? In *ACM IMW'02*, pages 173–174. ACM, 2002.

[68] John Sonchack, Adam J Aviv, Eric Keller, and Jonathan M Smith. Enabling practical software-defined networking security applications with OFX. In *NDSS'16*, pages 1–15, 2016.

[69] Anna Sperotto, Gregor Schaffrath, Ramin Sadre, Cristian Morariu, Aiko Pras, and Burkhard Stiller. An overview of ip flow-based intrusion detection. *IEEE communications surveys & tutorials*, 12(3):343–356, 2010.

[70] Zhiyang Su, Ting Wang, Yu Xia, and Mounir Hamdi. FlowCover: Low-cost flow monitoring scheme in software defined networks. In *IEEE GLOBECOM'14*, pages 1956–1961, 2014.

[71] José Suárez-Varela and Pere Barlet-Ros. Towards a netflow implementation for openflow software-defined networks. In *ITC'17*, volume 1, pages 187–195. IEEE, 2017.

[72] Junho Suh, Ted Taekyoung Kwon, Colin Dixon, Wes Felter, and Jenny Carter. OpenSample: A low-latency, sampling-based measurement platform for commodity SDN. In *IEEE ICDCS'14*, pages 228–237, 2014.

[73] Andrei M Sukhov, Dmitry I Sidelnikov, AP Platonov, MV Strizhov, and Aleksey A Galtsev. Active flows in diagnostic of troubleshooting on backbone links. *Journal of High Speed Networks*, 18(1):69–81, 2011.

[74] Arsalan Tavakoli, Martin Casado, Teemu Koponen, and Scott Shenker. Applying nox to the datacenter. In *ACM HotNets'09*, page 6. ACM, 2009.

[75] Tcpreplay. http://tcpreplay.synfin.net/wiki/tcpreplay.

[76] The CAIDA UCSD Anonymized Internet Traces 2016 - [18/02/2016]. http://www.caida.org/data/passive/passive 2016 dataset.xml.

[77] Amin Tootoonchian, Monia Ghobadi, and Yashar Ganjali. OpenTM: Traffic matrix estimator for OpenFlow networks. In *PAM'10*, pages 201–210. Springer, 2010.

[78] Niels LM Van Adrichem, Christian Doerr, and Fernando A Kuipers. OpenNetMon: Network monitoring in openflow software-defined networks. In *IEEE NOMS'14*, pages 1–8, 2014.

[79] Ronald van der Pol, Bart Gijsen, Piotr Zuraniewski, Daniel Filipe Cabaça Romão, and Marijke Kaat. Assessment of sdn technology for an easy-to-use vpn service. *Future Generation Computer Systems*, 56:295–302, 2016.

[80] Virtual Networks over Linux (VNX). http://web.dit.upm.es/vnxwiki/index.php.

[81] An Wang, Yang Guo, Fang Hao, TV Lakshman, and Songqing Chen. UMON: Flexible and fine grained traffic monitoring in Open vSwitch. In *ACM CoNEXT'15*, pages 1–7. ACM, 2015.

[82] Bing Wang, Yao Zheng, Wenjing Lou, and Y Thomas Hou. Ddos attack protection in the era of cloud computing and software-defined networking. *Computer Networks*, 81: 308–319, 2015.

[83] Guohui Wang, TS Ng, and Anees Shaikh. Programming your network at run-time for big data applications. In *HotSDN*, pages 103–108. ACM, 2012.

[84] Shao-Heng Wang, Patrick P-W Huang, Charles H-P Wen, and Li-Chun Wang. Eqvmp: Energy-efficient and qos-aware virtual machine placement for software defined data-center networks. In *ICOIN'14*, pages 220–225. IEEE, 2014.

[85] Abdulsalam Yassine, Hesam Rahimi, and Shervin Shirmohammadi. Software defined network traffic measurement: Current trends and challenges. *IEEE Instrumentation and Measurement Magazine*, 18(2):42–50, 2015.

[86] Soheil Hassas Yeganeh, Amin Tootoonchian, and Yashar Ganjali. On scalability of software-defined networking. *IEEE Communications Magazine*, 51(2):136–141, 2013.

[87] Curtis Yu, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Harsha V Madhyastha. FlowSense: Monitoring network utilization with zero measurement cost. In *PAM'13*, pages 31–41. Springer, 2013.

[88] Ying Zhang. An adaptive flow counting method for anomaly detection in SDN. In *ACM CoNEXT'13*, pages 25–30, 2013.

Publication List

Journal paper (reviewed)

[1] **X.T. Phan**, and K. Fukuda. Fine-grained traffic monitoring framework in Software-Defined Networks. *Journal of Information Processing*, vol. 25, pp.182–190, 2017.

Conference papers (reviewed)

[2] **X.T. Phan**, and K. Fukuda. Toward a flexible and scalable monitoring framework in Software-Defined Networks. In *IEEE Internetional Workshop on Network Management and Monitoring (NETMM), 2017*, pp.403–408, 2017.

[3] **X.T. Phan**, I.D.M. Casanueva, and K. Fukuda. Adaptive and distributed monitoring mechanism in Software-Defined Networks (short/poster paper). In *IFIP/IEEE International Conference on Network and Service Management (CNSM), 2017*, 5 pages, 2017.